Acta Mathematica et Informatica Universitatis Ostraviensis

Predrag Stanimirović; Svetozar Rančić

Implementation of penalty function methods in LISP

Acta Mathematica et Informatica Universitatis Ostraviensis, Vol. 7 (1999), No. 1, 119--141

Persistent URL: http://dml.cz/dmlcz/120543

Implementation of penalty function methods in LISP

Predrag Stanimirović
Svetozar Rančić

Abstract: We describe implementation of several varieties of penalty function methods for constrained nonlinear optimization in programming languages LISP and MATHEMATICA. Our main goal is application of symbolic processing in implementation of constrained optimization programs, which is so far considered only numerically. Developed routines are simpler and more universal than the corresponding in procedural programming languages. A few numerical examples are given.

Key Words: Constrained optimization, LISP, MATHEMATICA, symbolic processing

Mathematics Subject Classification: 90C30, 68N15

1. Introduction

Consider the following general nonlinear programming problems:

$$\begin{aligned} &\text{Minimize: } Q(\vec{x}), \quad \vec{x} \in \mathbb{R}^n \\ &\text{Subject to: } f_i(\vec{x}) \leq 0, \quad i \in \mathcal{P} = \{1, \ldots, p\} \\ &\qquad\qquad h_j(\vec{x}) = 0, \quad j \in \mathcal{Q} = \{1, \ldots, q\}. \end{aligned} \tag{1.1}$$

$$\begin{aligned} &\text{Minimize: } Q(\vec{x}), \quad \vec{x} \in \mathbb{R}^n \\ &\text{Subject to: } f_i(\vec{x}) \leq 0, \quad i \in \mathcal{P} = \{1, \ldots, p\}. \end{aligned} \tag{1.2}$$

The essence of the penalty function methods is to replace a constrained nonlinear programming problem to a sequence of unconstrained problems, whose solutions in the limit approximate the minimum of the nonlinear constrained problem [1], [6], [19].

In the literature are known computational systems for implementation of numerical optimization, which are written in procedural programming languages, mainly in FORTRAN [1], [2], [9], [14] and C [8]. But, procedural programming languages are not completely convenient for implementation of the optimization methods, which will be investigated in this paper.

On the other hand, in the programming package MATHEMATICA [17], [18] are available a few functions for numerical optimization. The function *FindMinimum*

starts at the specified points, and follows the path of steepest descent on the surface in approximation of the local minimum.

The functions *ConstrainedMin* and *ConstrainedMax* allow you to specify a linear objective function to minimize or maximize, together with a set of linear inequality constrains on variables. In all cases it is assumed that the variables are constrained to have non-negative values. More precisely, only the linear programming is implemented in MATHEMATICA.

`ConstrainedMin[f, {inequalities}, {x, y, ...}]` find the global minimum of f, in the region specified by *inequalities*;

`ConstrainedMax[f, {inequalities}, {x, y, ...}]` find the global maximum of f, in the region specified by *inequalities*.

Maximal precision of these functions is 14 digits.

This is an incomplete system in consideration of numerous optimization methods.

Main purpose of this paper is to describe implementation of penalty function methods for nonlinear constrained programs, using possibilities of the symbolic processing in functional programming languages LISP and MATHEMATICA.

Although LISP is evolved primarily as a support tool for artificial intelligence research and applications, its becomes increasingly popular among non-artificial intelligence programmers. SCHEME - dialect of LISP [11] is one of the most versatile programming language available today, useful for a variety of programming projects, available in both symbolic and numeric processing.

Additionally, we investigate application of the functionality of the programming language MATHEMATICA in implementation of the same penalty function methods.

This gives a basis for application of an arbitrary functional programming language, or an arbitrary functional technique, in implementation of constrained minimization methods.

This paper is first attempt to unite both symbolic and numeric processing in implementation of constrained optimization methods.

We suggest the following advantages which arise during the implementation of the constrained optimization methods in functional programming languages.

1. It is possible to apply an objective function for unconstrained optimization to some arguments under the program control.
2. It is possible, on an elementary way, to take the objective function and given constrains in the list of formal parameters of an arbitrary implementation procedure.
3. Simple implementation of symbolic differentiation in functional programming languages [4], [5], [10], [13], [17], [18]. Furthermore, the functions representing the partial derivative of the objective function can be easily applied to a list of arguments and incorporated as elements in the gradient of the objective function.
4. Possibility of symbolic processing in transformation of the internal form of given constrained nonlinear programming problem into the internal form of the corresponding unconstrained problem, according to the principles of

a few varieties of the penalty function methods. Generated internal form, applicable in unconstrained optimization, can be placed in the list of formal parameters of a selected procedure for unconstrained optimization.

5. It is possible to include symbolically generated *slack variables* and weighting coefficients in the internal form of the *transformed function*.
6. Simple extension of *parameter list* of the objective function by the list of slack variables.
7. Possibility to generate a vector or a list whose elements are selected functions. In the programming language LISP, elements of such a vector can be later transformed into the corresponding lambda-expressions and applied. In the package MATHEMATICA, functions placed in a list can be used directly.

The paper is organized as follows: In Section 2 we briefly describe the internal representation of nonlinear constrained programs and necessary declarations. Also, we point out a few advantages of the symbolic processing. In section 3 we describe transformations of a given nonlinear constrained problem into the corresponding unconstrained nonlinear problem. In section 4 we are concerned with the main implementation details for nonlinear constrained programs. In Section 5 a few numerical results are reported, and relations with the corresponding results, obtained by means of the procedural programming languages, are discussed.

2. Internal representations and preliminaries

In programs written in the procedural programming languages, only the set of objective functions defined in subroutines (so called test-functions) can be used [1], [8], [9], [14]. For example, in [2], the objective function is rewritten as a sequence of calls to subroutines. Also, in gradient methods, the computer subroutines that specify the partial derivatives of the objective functions are also defined.

For example, the function $f(x,y,z) = (x-1)^2 + (y-1)^2 + (z-1)^2$ is given by [8]

```
float func(x)
float x[];
{ int i; float f=0.0;
   for(i=1; i<=3; i++) f+=(x[i]-1.0)*(x[i]-1.0);
   return f;
}
```

The sequence of its partial derivatives is generated in the following subroutine [8]

```
void dfunc(x,df)
float x[], df[];
{ int i;
   for(i=1; i<=3; i++) df[i]+=2.0*(x[i]-1.0);
}
```

If the objective function is changed, an user must to change both of these routines. The same principle is used in the programming language FORTRAN [1], [2], [9], [14].

In the language FORTRAN, it is possible to use any selected objective function in the list of formal parameters of an arbitrary optimization procedure, only by means of the lexical and syntax analysis of the entered expression.

In the language C, an arbitrary objective function can be used in the list of formal parameters by means of the pointer to this function.

In the literature there are derived a lot of methods for automatic differentiation in the procedural programming languages, mainly in FORTRAN, C or C++ (see for example [2], [3], [7]). Automatic differentiation in the procedural programming languages is divided to numerical differentiation (which produces an approximation for $f'(x)$) and symbolic differentiation which produces a formula for $f'(x)$. Numerical evaluation of the derivatives by difference schemes can be implemented in an arbitrary programming language, and we leave it out of consideration. Symbolic differentiation is more convenient problem for functional programming languages. In the procedural programming languages symbolic algebraic expressions are parsed, i.e. converted into the reverse Polish notation and into a binary tree form [7]. Since parsing is done, according to differentiation rules, tree derivative is made. After differentiation, the binary tree is simplified to remove useless operations [7]. In [2] is stated that automatic differentiation (in the procedural languages) is suited to languages such as C, PASCAL or ADA which permit the introduction of data types and additional definitions of the operator symbols to manipulate such types. In [2] the problem of automatic differentiation is solved using a set of ordered triplets.

On the other hand, these problems can be easily solved in functional programming languages. Firstly, in functional programming languages an arbitrary function can be used as argument during unconstrained and constrained optimization.

In the functions implementing unconstrained optimization methods in LISP [12], [13], an arbitrary real objective function $Q(x_1, \ldots x_n) = Q(\vec{x})$ is represented by the list of two elements in the form

'(Q($\vec{x}$) ($\vec{x}$))

The first element of this list is a selected PC SCHEME arithmetic function and the second represents its argument list.

The internal representation, denoted by q, of a given objective function, used in unconstrained optimization, can be transformed into the corresponding *lambda-function* [12], [13]:

```
(set! fun (eval (list 'lambda (cadr q) (car q)))).
```

This lambda function can be applied to an argument list v:

```
(apply fun v)
```

Assume that $q\_$ is the parameter denoting the objective function Q in MATHEMATICA. Let the variable *var* denotes the parameter list of Q and $x0$ is the list representing a given point. Then the expression $q[x0]$ can be computed as follows

```
q0=q;    Do[q0=q0/.var[[i]]->x0[[i]], {i,n}];
```

This is essence of the advantage 1.

Furthermore, symbolic differentiation in functional programming languages is an easily solvable problem [4], [5], [10], [15], [17], [18].

In the programming language LISP, we need to modify the known routines for symbolic differentiation [10], [15], to be applicable to an arbitrary LISP arithmetic expression. Also, it is desirable to implement several routines for simplification of the resulting partial derivatives [15].

A relationship between the numerical and symbolic differentiation is studied in [3]. Symbolic differentiation is free from function error, but the naive formula manipulation for differentiation will require far more time and space than numerical differentiation [3]. There has been a lot of progress in formula manipulators to make improvement on those disadvantages by incorporating techniques similar to those employed in automatic differentiation [3]. Also, an application of the symbolic differentiation in computation of the formula for the gradient or Jacobian is well known problem [3]. We now investigate possibilities of the languages LISP and MATHEMATICA in symbolic construction of the gradient and Jacobian.

If the function making the symbolic partial derivative of the function $f(\vec{x})$, depending on the independent variable x_i, $1 \leq i \leq n$ is defined by the function $(deriv \quad f \quad x_i)$, then the gradient of the internal form

$$((Q(\vec{x}))\,(\vec{x})) = ((Q(x_1, \ldots, x_n))\,(x_1, \ldots, x_n))$$

can be represented by the following list

$$\nabla\,((Q(\vec{x}))\,(\vec{x})) = ((deriv \quad Q\ x_1) \cdots (deriv \quad Q\ x_n))\,.$$

Elements of the list, representing the value of gradient in a given vector $x0 = \left(x_1^{(0)}, \ldots x_n^{(0)}\right)$, can be obtained forming the lambda-expressions from the corresponding expressions representing partial derivatives of the objective function, and applying these expressions on the successive elements of the vector $x0$.

```
(define (gradfor q x0)
  (let ((f (car q)) (p (cadr q)) (p1 p) (nl nil))
     (do (fc)
         ( (null?  p1) nl)
         (set!  fc (eval (list 'lambda p (deriv f (car p1)))))
         (set!  p1 (cdr p1))
         (set!  nl (append nl (list (apply fc x0))))
) )   )
```

Gradient of a given objective function q in a given point $x0$ in MATHEMATICA can be formed using the differentiation operator D, as follows, where $var\_$ denotes the list of variables.

```
gradfor[q_,var_List,x0_List]:=
  Block[{n=Length[var],i,dqdx={}},
     Do[dqdx=Append[dqdx,D[q,var[[i]]]],{i,n}];
```

```
    Do[dqdx=dqdx/.var[[i]]->x0[[i]],{i,n}];
    dqdx
]
```

In this way, we just describe the advantage 3.

Consider now implementation of constrained programs in the procedural programming languages. In the procedural programming languages it is an inconvenient problem to place an arbitrary objective function and constrains in the list of parameters of the subroutine which implements an optimization problem. Usually, the selected objective function and constrains are defined by a subroutine, and placed into an array [1], [14]. Consequently, application of a new objective function and constrains are conditioned by these definitions. For example, in [1] each equality constraint, inequality constraint, and the objective function are identified by the subscripted variable - $R[i]$. In this way, program (1.1) is stored as follows:

$$\begin{aligned} R(1) = f_1(\vec{x}), \ldots, R(p) = f_p(\vec{x}), \\ R(p+1) = h_1(\vec{x}), \ldots, R(p+q) = h_q(\vec{x}), \\ R(p+q+1) = Q(\vec{x}). \end{aligned}$$

Each application of the constrained minimization procedure requires a modification of the array R. Moreover, size of the problems is limited by the dimension of R.

For example, the following constrained program

$$\begin{aligned} &\text{Minimize: } -x_1 - x_2 \\ &\text{Subject to: } f_1(x_1, x_2) = x_1^2 + x_2^2 - 1 \leq 0, \\ &\qquad\qquad h_1(x_1, x_2) = -x_1 + x_2^2 = 0 \end{aligned}$$

is represented as follows:

$$\begin{aligned} R(1) &= X(1) * *2 + X(2) * *2 - 1, \\ R(2) &= -X(1) + X(2) * *2, \\ R(3) &= -X(1) - X(2). \end{aligned}$$

Consequently, definition of the objective function and the constrains are strictly bounded with the values of the global array X.

However, it is possible to avoid this problem in the language C, using linked lists generated by means of dynamic memory allocation.

On the other hand, we describe algorithm which makes possible to use the objective function and constrains in the role of formal parameters in LISP and MATHEMATICA. For this purpose, in this paper we introduce the *internal form* appropriate for nonlinear constrained problems. The nonlinear constrained problem (1.1) is transformed in the following LISP's form:

```
'(   Q(x⃗)   (x⃗)
     ( f1(x⃗) ··· fp(x⃗) )
```

($h_1(\vec{x})$ $\cdots$ $h_q(\vec{x})$)

)

Analogous internal form in MATHEMATICA is

{ Q($\vec{x}$) , {$\vec{x}$},
{ $f_1(\vec{x})$ $\cdots$ $f_p(\vec{x})$ }
{ $h_1(\vec{x})$ $\cdots$ $h_q(\vec{x})$ }

}

If one of the inequality or equality constrains absent, the corresponding list is empty.

Example 2.1. The nonlinear constrained programming problem

$$\text{Minimize: } -x_1 - x_2$$
$$\text{Subject to: } x_1^2 + x_2^2 - 1 = 0$$

is represented in the internal following LISP's form

```
' (  (- 0 (+ x1 x2))  (x1 x2)
     ()
     (  (- (+ (* x1 x1) (* x2 x2)) 1) )
)
```

Corresponding internal form in MATHEMATICA is

```
{  -x1-x2, {x1,x2},
     {},
     {x1^2+x2^2-1}
}
```

The first element of the *internal form* is an arbitrary PC SCHEME or MATHEMATICA arithmetic function, the second represents its argument list, the third element is a list of functions forming the inequality constrains, and the fourth is interpreted as a list of functions contained in the set of equality constrains. Consequently, the *function* contained in the internal form q of an arbitrary constrained nonlinear program can be selected by the expression (*car* q), and the corresponding *parameter list* by the expression (*cadr* q). Similarly, the list of functions forming the inequality constrains can be extracted by the expression (*caddr* q), and the list of functions forming the inequality constrains using (*cadddr* q). Analogous expressions in MATHEMATICA are q[[1]], q[[2]], q[[3]] and q[[3]].

Note that the user must to set task of the constrained optimization problem into the described internal form. Our motivation for application of such an internal form is the similar internal form for linear constrained programs in MATHEMATICA (see the function *ConstrainedMin*.)

In this way, we show the advantage 2.

The vector vf containing the functions in equality constrains in a PC SCHEME internal form q can be constructed as follows:

```
(set! vf (make-vector (set! lf (length (caddr q)))))
(if (< 0 lf) (set! vf (list->vector (caddr q))))
```

In a similar way we generate the vector vh of inequality constrains:

```
(set! vh (make-vector (set! lh (length (cadddr q)))))
(if (< 0 lh) (set! vh (list->vector (caddr q))))
```

The list of inequality and equality constrains in MATHEMATICA can be selected by the following expressions, respectively:

```
vf=q[[3]],   vh=q[[4]]
```

In these routines we use possibility of functional programming languages to place arbitrary selected functions in a list or a vector, which is an inconvenient problem in procedural programming languages. This is a part of the advantage 7.

3. Construction of the corresponding unconstrained problem

In this section we describe symbolic transformation of a given internal form of a constrained problem into the internal form of the corresponding unconstrained problem, i.e. the advantage 4. As far as we know, this problem is not employed before. Assume that a constrained program is given by the internal form $(Q\ (\vec{x})\ (\vec{f})\ (\vec{h}))$ or $\{Q\ \{\vec{x}\}\ \{\vec{f}\}\ \{\vec{h}\}\}$ where $\vec{f} = f_1, \ldots, f_p$ and $\vec{h} = h_1, \ldots, h_q$ denote given inequality and equality constrains, respectively. We develop procedures in PC SCHEME and MATHEMATICA for transformation of a given constrained program into the formula, which represents the objective function Q_u of the corresponding unconstrained program. Then the internal form of the unconstrained program is equal to the list $(Q_u\ (\vec{x}))$ or $\{Q_u\ \{\vec{x}\}\}$.

In one of the exterior point methods, the nonlinear constrained problem (1.1) is converted into the following sequence of unconstrained problems [1], [6], [19]:

$$\begin{aligned} \min_{(\vec{x})} F(\vec{x}) &= \min_{(\vec{x})} \left(Q(\vec{x}) + \frac{1}{\rho^{(k)}} P(\vec{x}) \right) \\ &= \min_{(\vec{x})} \left(Q(\vec{x}) + \frac{1}{\rho^{(k)}} \left\{ \sum_{i=1}^{p} [f_i(\vec{x})]_+{}^{\alpha} + \sum_{j=1}^{q} |h_j(\vec{x})|^{\beta} \right\} \right), \end{aligned} \tag{3.1}$$

where $[f_i(\vec{x})]_+ = \max\{0, f_i(\vec{x})\}$, $\rho^{(k)}$ is strongly decreased sequence of positive numbers, and α, $\beta \geq 1$ are two integers.

In the LISP's internal form Q_u of the objective function of the corresponding unconstrained program, the strings "ro", "$alpha$" and "$beta$" are used instead of the parameters $\rho^{(k)}$, α, β. The internal form $(Q_u\ (\vec{x}))$ can be symbolically generated by means of the following routine.

*PROCEDURE GOAL*1.

Step 1. To form the formula Q_u, called $fgoal$, which has the form

$$G(\vec{x}, "ro", "alpha", "beta") = Q(\vec{x}) + \frac{1}{"ro"} \left\{ \sum_{i=1}^{p} [f_i(\vec{x})]_+{}^{"alpha"} + \sum_{j=1}^{q} |h_j(\vec{x})|^{"beta"} \right\}$$

```
(set!  fgoal 0)
(do ((i 1))
  ((= i lf))
  (set!  fgoal
   (list '+
        (list 'expt (list 'max 0 (vector-ref vf i)) "alpha")
        fgoal
  ))
   (set!  i (+ i 1)))
(do ((i 1))
   ((= i lh))
   (set!  fgoal
      (list '+
            (list 'expt (abs (vector-ref vh i)) "beta")
            fgoal
   )  )
   (set!  i (+ i 1)))
(set!  fgoal (list '+ (car q)(list (list '/' 1 "ro") fgoal)))
```

Step 2. Append the parameter list to the list Q_u:

```
(set!  fgoal (list fgoal (cadr q)))
```

We now describe an analogous routine in MATHEMATICA.

Step 1. Decreased sequence ρ^k can be determined by means of the following recursive definition

```
ro[1]=1;
ro[n_]:=ro[n-1]/2
```

Then the function *fgoal* in the ith iteration can be formed as follows:

```
fgoal=q[[1]]+1/ro[[i]](Sum[Max[0,vf[[i]]^alpha,{i,Length[vf]}]
                + Sum[(Abs[vh[[i]])^beta, {i,Length[vh]});
```

Also, we can use the symbol *ro* instead of the function *ro*, and write

```
fgoal=q[[1]]+1/ro(Sum[Max[0, vf[[i]]^alpha, {i,Length[vf]}]
         + Sum[(Abs[vh[[i]])^beta, {i,Length[vh]});
```

Step 2. The internal form $\{Q_u \ \{\vec{x}\}\}$ is formed using the built-in function *List*.

```
fgoal:=List[fgoal, q[[2]]];
```

Example 3.1. The LISP's internal form of the objective function Q_u, corresponding to the nonlinear programming problem stated in Example 2.1, is equal to

```
( (+ (- 0 (+ x1 x2))
     (* (/ 1 "ro")
        (expt (- (+ (* x1 x1) (* x2 x2)) 1) "beta")
)  )
(x1 x2)
)
```

Corresponding internal form in MATHEMATICA is

```
-x1-x2 + 1/ro (Abs[-1+x1^2-x2^2])^beta .
```

One of the most popular interior point methods converts the nonlinear programming problem (1.2) into the following sequence of unconstrained problems [1], [6], [19]:

$$\min_{(\vec{x})} F(\vec{x}) = \min_{(\vec{x})} \left(Q(\vec{x}) + \rho^{(k)} P(\vec{x})\right) = \min_{(\vec{x})} \left(Q(\vec{x}) + \rho^{(k)} \sum_{i=1}^{p} \frac{1}{[f_i(\vec{x})]^2}\right), \tag{3.2}$$

where the weighting factors $\rho^{(k)}$ are positive and form a monotonically decreasing sequence of values.

The internal form *fgoal* of the function Q_u is the formula

$$G(\vec{x}, ro) = Q(\vec{x}) + "ro" \sum_{i=1}^{p} \frac{1}{[f_i(\vec{x})]^2}$$

where the string "*ro*" is placed in positions of the parameter ρ. Now, the internal form of the generated constrained program can be formed applying the following code in PC SCHEME:

```
(set!  fgoal 0)
(do ((i 0))
   ((= i lf))
   (set!  fgoal
      (list '+
            (list '/ 1 (list 'expt (vector-ref vf i) 2))
            fgoal
   )  )
   (set!  i (+ i 1)))
   (set!  fgoal (list '+ (car q) (list '* "ro" fgoal)))
   (set!  fgoal (list fgoal (cadr q)))
```

The internal form of the corresponding unconstrained minimization problem can be produced using only the following two expressions in MATHEMATICA:

```
fgoal=q[[1]]+ro Sum[1/vf[[i]]^2, {i,Length[vf]}];
fgoal:=List[fgoal, q[[2]]];
```

To generalize the technique of Lagrange multipliers, the inequality constrains must be treated as equations by introduction of appropriate slack variables, one for each inequality constraint [1], [6], [19]. In this way, the general constrained optimization program (1.1) is converted to the following equivalent program, which uses only equality constraints:

$$\begin{aligned} &\text{Minimize: } Q(\vec{x}), \quad \vec{x} \in \mathbb{R}^n \\ &\text{Subject to: } g_i(\vec{x}) = f_i(\vec{x}) + z_i^2 = 0, \quad i \in \mathcal{P} \\ &\qquad\qquad h_j(\vec{x}) = 0, \quad j \in \mathcal{Q}. \end{aligned} \tag{3.3}$$

Finally, the program (3.3) is transformed into the following sequence of unconstrained minimization problems:

$$\begin{aligned} &\min_{(\vec{x},\ \vec{z})} L(\vec{x},\ \vec{z}) = \min_{(\vec{x},\ \vec{z})} (Q(\vec{x}) + P(\vec{x},\ \vec{z})) \\ &= \min_{(\vec{x},\ \vec{z})} \left(Q(\vec{x}) + \sum_{i=1}^{p} \mu_i (f_i(\vec{x}) + z_i^2) + \sum_{j=1}^{q} \mu_{j+p+1} |h_j(\vec{x})| \right), \end{aligned} \tag{3.4}$$

where $\mu_i, \quad i = 1, \ldots, p+q$ are nonnegative weighting factors independent of $\vec{x}$, identifiable as the Lagrange multipliers, and the vector $\vec{z}$ contains the *slack variables* $z_i, \quad i = 1, \ldots, p$.

We now describe possibility of symbolic processing to generate the *slack variables* z_i and nonnegative weighting factors μ_i, as well as their incorporation in the internal form of transformed function for unconstrained optimization. In this manner, we justify the advantage 5.

Let $lf = p + 1$ denotes the number of inequality constrains, and $lh = q + 1$ represents the number of equality constrains. The extended Lagrange's function (3.4) requires lf auxiliary variables z_i, i.e. the corresponding symbolic expression in PC SCHEME requires lf symbols. For this purpose, we use the symbols given in a list, denoted by $alzi$, and externally declared by

$$alzi = (z0\ z1\ z2\ z3\ z4\ z5\ z6\ z7\ z8\ z9).$$

Now, the vector `(set! vecz (list->vector alzi))` contains the symbols zi.

The list of used μ_i coefficients, denoted by $listm = \mu = (\mu_0, \ldots, \mu_{p+q+1})$, can be formed by means of the following routine which generates the lists whose elements are strings:

```
(define (gen-str-list pf begin n)
   (let ((vec (make-vector n)) (i 1) (iv 0) )
    (do ((i begin) (iv 0))
        ((= n iv) (vector->list vec))
        (vector-set!  vec iv (string-append pf
                              (number->string i '(int))))
        (set!  iv (+ iv 1))
        (set!  i (+ i 1)))))
```

Now, the value of the expression

```
(set! listm (list->vector (gen-str-list "m" 0 (+ lf lh 1))))
```

is the list of strings corresponding to weighting μ coefficients:

$$vecm = (\text{"}m0\text{"}\ \text{"}m1\text{"}\ \cdots\ \text{"}mt\text{"}), \quad t = lf + lh + 1.$$

Now, the internal form of the objective function Q_u is generated using the pattern

$$G(\vec{x}, vecz, vecm) = Q(\vec{x}) + \sum_{i=1}^{p} vecm_i \left(f_i(\vec{x}) + vecz_i^2\right) + \sum_{j=1}^{q} vecm_{j+p} |h_j(\vec{x})|$$

The internal form (Q_u ($\vec{x}$)) is generated in the following procedure.

PROCEDURE GOAL2.

Step 1. Form the first part (i.e. the function Q_u) of the internal form:

```
(set!  fgoal 0)
(do ((i 1))
   ((= i lf))
   (set!  fgoal
     (list '+ fgoal
           (list '* (vector-ref vecm i)
                 (list '+ (vector-ref vf i)
                       (list 'expt (vector-ref vecz i) 2)
   ) )     )     )
(do ((i 1))
   ((= i lh))
   (set!  fgoal
      (list '+ fgoal
            (list '* (vector-ref vecm (+ i lf 1))
                     (list 'abs (vector-ref vh i))
   ) )     )
   (set!  i (+ i 1))
)
(set!  fgoal (list '+ (car q)  fgoal))
```

Step 2. Append the list of $\vec{x}$ and $\vec{z}$ parameters to the first element of the internal form:

```
(set!  fgoal (list fgoal (append (cadr q) (alzi)))
```

In this way, the parameter list ($\vec{x}$) of the initial constrained problem is extended to the new parameter list ($\vec{x}$ $\vec{z}$), where the vector $\vec{z}$ contains unbounded slack variables. This is the essence of the advantage 6. Note that in [13] we describe transformation of the multiargument objective function into the function of one argument. Also, the advantages 1. and 3. are used during implementation of unconstrained optimization (see [13].)

The *slack variables* z_i and weighting coefficients μ_i in MATHEMATICA can be implemented using the symbols of the form $z[i]$ and $vecm[i]$, where i denotes a variable used in the cycle.

The internal form $fgoal$ of the objective function Q_u, in MATHEMATICA is formed as follows:

```
fgoal=q[[1]]+Sum[vecm[i] (vf[[i]]+z[i]^2), {i,Length[vf]}]
        +Sum[vecm[i+Length[vf]] Abs[vh[[i]]],{i,Length[vh]}]
```

The list of parameters can be extended by the *slack variables* and appended to the internal form $fgoal$ in the following way:

```
Do[q[[2]]=Append[q[[2]],z[i]],{i,Length[vf]}];
```

Note that $z[i]$ are symbols whose values are not defined. In this way, an unlimited number of slack variables $z[i]$ is included into the internal form of the just constructed unconstrained optimization program. More precisely, we are in a position to include slack variables into the analytic expression of the objective function, as well as in its argument list.

Similar principles are valid for implementation of the Rockafellar's extension of the Lagrange's function, which is defined by:

$$L(\vec{x},\lambda,\mu)=Q(\vec{x})+\frac{1}{4a}\sum_{i\in\mathcal{P}}\{[\lambda_i+2af_i(\vec{x})]_+^2-\lambda_i^2\} - \sum_{j\in\mathcal{Q}}\mu_j h_j(\vec{x})+\frac{a}{2}\sum_{j\in\mathcal{Q}}[h_j(\vec{x})]^2, \tag{3.5}$$

where $a > 0$ is sufficiently large real number. The constrained nonlinear programming problem (3.3) is converted into the following sequence of unconstrained problems [19]:

$$\begin{aligned}&\lambda_0 \geq 0, \mu_0 \text{ are arbitrary}\\ &\text{Minimize } L(\vec{x},\lambda_i^{(k)},\mu^{(k)}),\\ &\text{where } \lambda_i^{(k+1)}=[\lambda_i^{(k)}+2af_i(\vec{x}^{(k)})]_+, \quad i\in\mathcal{P}\\ &\mu_j^{(k+1)}=[\mu_j^{(k)}-h_j(\vec{x}^{(k)})], \quad j\in\mathcal{Q}, \quad k=0,1,\ldots\end{aligned} \tag{3.6}$$

The internal representation of the Rockafellar's extended lagrangian, denoted by

$$L(\vec{x},vecl,vecm)=Q(\vec{x})+\frac{1}{4a}\sum_{i\in\mathcal{P}}\{[vecl_i+2af_i(\vec{x})]_+^2-vecl_i^2\} - \sum_{j\in\mathcal{Q}}vecm_j h_j(\vec{x})+\frac{a}{2}\sum_{j\in\mathcal{Q}}[h_j(\vec{x})]^2$$

is formed by means of the following algorithm:

Step 1. The vectors of used λ_i and μ_i coefficients, denoted by $vecl = \lambda = (\lambda_0,\ldots,\lambda_p)$ and $vecm = \mu = (\mu_0,\ldots,\mu_q)$ are symbolically formed as follows:

```
(set!  vecl (list->vector (gen-str-list "l" 0 lf)))
(set!  vecm (list->vector (gen-str-list "m" 0 lh)))
```

Step 2. The internal form of the function

$$\frac{1}{4a}\sum_{i\in\mathcal{P}}\{[\lambda_i+2af_i(\vec{x})]_+^2-\lambda_i^2\}=\frac{1}{4a}\sum_{i\in\mathcal{P}}\{[vecl_i+2af_i(\vec{x})]_+^2-vecl_i^2\}$$

can be generated as follows:

```
(define (plus x)  (if (< x 0) 0 x))

(set!  fgoal 0)
```

```
(do ((i 1))
   ((= i lf))
   (set!  fgoal (list '+ fgoal
                  (list '-
                      (list 'expt
                         (list 'plus
                              (list '+ (vector-ref vecl i)
                                    (list '* 2 a
                                          (vector-ref vf i)
                              )     )
                              ) 2
                         )
                         (list 'expt (vector-ref vecl i) 2)
   )           )  )
   (set!  i (+ i 1))
)
(set!  fgoal (list '/ fgoal (list '* 4 a)))
```

Step 3. To form the internal representation of the parts

$$\sum_{j\in Q}\mu_j h_j(\vec{x}) = \sum_{j\in Q} vecm_j h_j(\vec{x}), \quad \text{and} \quad \frac{a}{2}\sum_{j\in Q}[h_j(\vec{x})]^2 :$$

```
(set!  hgoal 0) (set!  hhgoal 0)
(do ((i 1))
      ((= i lh))
      (set!  hgoal (list '+ hgoal
                       (list '* (vector-ref vecm i)
                             (vector-ref vh i)
      )           )     )
      (set!  hhgoal (list '+ hhgoal
                          (list '* (vector-ref lh i)
                                (vector-ref lh i)
      )            )     )
      (set!  i (+ i 1))
)
   (set!  hhgoal (list '* a hhgoal 0.5))
```

Step 4. Internal representation of the function $L(\vec{x}, vecl, vecm)$.

```
(set!  fgoal (list '- (list '+ fgoal  hhgoal) hgoal))
(set!  fgoal (list '+ (car q) fgoal))
(set!  fgoal (list fgoal (cadr q)))
```

4. Evaluation of nonlinear constrained methods

The general algorithm used in symbolic implementation of nonlinear constrained methods can described as follows:

Step 1. Select the corresponding values of the formal parameters:
- State the internal form of a selected objective function and given constrains, as it is described in Section 2.
- Select a starting point $vx = \vec{x}^{(0)}$, satisfying given constrains.
- Select a small real number, which determines the stopping criterion.

Step 2. Declare the local variables and construct the vectors vf and vh, arising from the given inequality and equality constrains, respectively, using the routines described in Section 2.

Step 3. Make the internal form of the type $(Q_u(\vec{x})\ \ (\vec{x}))$ in PC SCHEME or the type $\{Q_u(\vec{x}),\ \{\vec{x}\}\}$ in MATHEMATICA. These internal forms are applicable in the procedures implementing a set of unconstrained optimization methods, described in [12], [13]. The corresponding algorithms are described in Section 3.

Step 4. To form the initial values for adjustable parameters. In the problems (3.1) and (3.2) the adequate values for the symbols *ro*, *alpha*, *beta* must be given:

```
(set!  ro (read))  (set!  alpha (read)) (set!  beta (read)).
```

```
ro=Input[];  alpha =Input[]; beta =Input[];.
```

The problems (3.4) and (3.5) use the vectors vm and vl, containing the values for the elements of the vectors *vecm* and *vecl*, respectively. The corresponding initial values can be selected by means of the following procedure, which generates values contained in a vector $v = (v_0, \ldots, v_n)$.

```
(define (vecv n)
   (do ((i 1) (v (make-vector n)))
       ((= i n) v)
       (vector-set!  (vector-ref v i) (read))
       (set!  i (+ i 1))
)   )
```

For the problem (3.4) we use the starting values for μ coefficients and the initial point $vx = \vec{x}^{(0)}$, generated by:

```
(set!  vm (vecv (+ lf lh 1)))
(set!  vx (vecv (length (cadr q))))
```

Also, the initial values of the slack variables $vz = \vec{z}^{(0)}$ can be computed using the values of the entered starting point $vx = \vec{x}^{(0)}$, according to (3.3), as follows:

$$z_i^{(0)} = \sqrt{-f_i(\vec{x}^{(0)})}, \quad i \in \mathcal{P}:$$

In PC SCHEME we write

```
(do ((i 1))
   ((= i lf))
   (vector-set!  vz i
```

```
        (* (sqrt (apply (eval (list 'lambda (cadr q) (car q)))
                        (vector->list vx)))) -1)
    (set!  i (+ i 1)))
```

In MATHEMATICA we suggest the following code

```
vz=Input[];
Do[vz=Append[Sqrt[-sub[vf[[i]],q[[2]],vz]],{i,Length[vf]}];
```

where *sub* is a function which substitutes each variable x_i from $var1$ by the corresponding value $x0_i$:

```
sub[equation_,var1_List,x_List]:=
Block[{eq=equation,i,var=var1,x0=x},
     Do[eq=eq/.var[[i]]->x0[[i]],{i,Length[var]}];
     Return[eq];
];
```

The problem (3.5) uses initial values for the symbol a and the vectors vf, vh:

```
(set!  a (read)) (set!  vl (vecv lf)) (set!  vm (vecv lh))
```

Step 5. A do cycle which terminates when a selected stopping criterion is satisfied. In the cycle perform the following:

Step 5.1. Substitute each string or symbol, corresponding to one of the adjustable parameters with the corresponding value, in the internal form of the corresponding unconstrained problem, formed in Step 3. In PC SCHEME this can be done by means of the function *subst*, which is described in [5], [16]. Evaluation of the expression `(subst npart oldpart list)` substitutes *oldpart* with *npart* in all levels of *list*.

For the problem (3.1) we can write

```
(set!  fgoal (subst ro "ro" fgoal))
(set!  fgoal (subst alpha "alpha" fgoal))
(set!  fgoal (subst beta "beta" fgoal))
```

The problem (3.2) uses

```
(set!  fgoal (subst ro "ro" fgoal))
```

The problem (3.4) uses the following:

```
(do ((i 1))
   ((= i (+ lf lh)))
   (set!  fgoal (subst (vector-ref vm i)
                       (vector-ref vecm i) fgoal))
   (set!  i (+ i 1))
)
(do ((i 1))
   ((= i lh))
   (set!  fgoal (subst (vector-ref vz i)
                       (vector-ref vecz i) fgoal))
   (set!  i (+ i 1))
```

```
)
```

and in the problem (3.5) we write

```
(do ((i 1))
   ((= i lf))
   (set!  fgoal (subst (vector-ref vl i)
                       (vector-ref vecl i) fgoal))
   (set!  i (+ i 1))
)
(do ((i 1))
   ((= i lh))
   (set!  fgoal (subst (vector-ref vm i)
                       (vector-ref vecm i) fgoal))
   (set!  i (+ i 1))
)
```

In MATHEMATICA this goal can be achieved in the same way, using the *replacement operator* /.

Step 5.2. Perform the unconstrained minimization, using one of the methods presented in [12], [13]. In this way, we give the new approximation of the optimal point.

Step 5.3. Select new values for the adjustable parameters. In the Rockafellar's method the adjustable parameters must be generated, according to (3.6). The new values in the vectors vm and vl can be computed by means of the following code, where vx denotes the vector of values for x_i

```
;To form the vectors of the functions
;contained in vf and vg
(do ((i 1))
   ((= i lf))
   (vector-set!  vffun i (eval (list 'lambda (cadr q)
                                     (vector-ref vf i))))
   (vector-set!  vhfun i (eval (list 'lambda (cadr q)
                                     (vector-ref vh i))))
   (set!  i (+ i 1))
)

;using the formed functions to set the new values
;for λ and μ
(do ((i 1))
   ((= i lf))
   (vector-set!  vl i
            (plus (+ (vector-ref vl i)
                     (* 2  a
                         (apply (vector-ref vffun i)
                             (vector->list vx))
   )        )     )  )
   (set!  i (+ i 1))
```

```
)
(do ((i 1))
   ((= i lh))
   (vector-set!  vm i
    (- (vector-ref vm i)
       (* a (apply (vector-ref vhfun i) (vector->list vx)
   )) )    )
   (set!  i (+ i 1)))
)
```

In the above presented code, functions contained in the vectors $vffun$ and $vhfun$ can be extracted, transformed into the corresponding lambda-expressions, and applied to the supplied argument list. In this way we describe the advantage 7.

5. Computational experience

Example 5.1. Consider the following nonlinear programming problem:

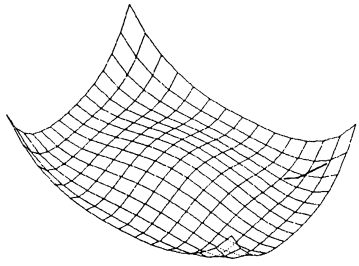
$$\text{Maximize: } -x_1 - x_2$$
$$\text{Subject to: } f_1(x_1, x_2) = x_1^2 + x_2^2 - 1 = 0.$$

Using the exterior point method, for the case $\beta = 1$, $\rho_0 = 1$, $\rho^{(k)} = \frac{1}{2^k}$, and using the Newton's method with the precision 10^{-14} in the generated unconstrained optimization problem, we obtain the following results:

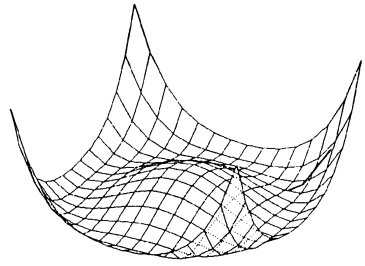
k	$\rho^{(k)}$	$x_1^{(k)}$	$x_2^{(k)}$	$Q(\vec{x}^{(k)})$
0	1.	1.	1.	-2.
1	1	0.80901699437494	0.80901699437494	-1.5225424859373
2	0.2	0.73089310318622	0.73089310318622	-1.4383869376311
3	0.04	0.71205472555989	0.71205472555989	-1.4191786979486
4	0.008	0.70810466782927	0.70810466782927	-1.4152121521447
5	0.0016	0.70730669639767	0.70730669639767	-1.4144135058365
6	3.2e-4	0.70714677779294	0.70714677779294	-1.4142535601106
7	6.4e-5	0.70711478105078	0.70711478105078	-1.4142215622825
8	1.28e-5	0.70710838118111	0.70710838118111	-1.4142151623694
9	2.56e-6	0.70710710118633	0.70710710118633	-1.4142138823729
10	5.12e-7	0.70710684518653	0.70710684518653	-1.4142136263730
15	1.6384e-10	0.70710678120702	0.70710678120702	-1.4142135623935
20	5.24288e-14	0.70710678118655	0.70710678118655	-1.4142135623731
21	1.048576e-14	0.70710678118654	0.70710678118654	-1.4142135623731
22	2.097152e-15	0.70710678118654	0.70710678118654	-1.4142135623731
23	4.194304e-16	0.70710678118654	0.70710678118654	-1.4142135623731

Table 1.

The following figures show trajectories which are formed using computed approximations of the local minimum:



$\rho = 1.0$



$\rho = 10^3$

The analogous results, for $\rho_k = \frac{1}{10^k}$, given by means of traditional implementation of the exterior point method are arranged in Table 2. [19]:

k	$\rho^{(k)}$	$x_1^{(k)}$	$x_2^{(k)}$	$Q(\vec{x}^{(k)})$
1	1	0.809017	0.809017	-1.618034
2	10^{-1}	0.719290	0.719290	-1.438580
3	10^{-2}	0.708354	0.708354	-1.416708
4	10^{-3}	0.707232	0.707232	-1.414464
5	10^{-4}	0.707119	0.707119	-1.414238
6	10^{-5}	0.707108	0.707108	-1.414216
7	10^{-6}	0.707107	0.707107	-1.414214
8	10^{-7}	0.707107	0.707107	-1.414214

Table 2.

Example 5.2. Consider the following nonlinear programming problem:

$$\begin{aligned} &\text{Minimize: } -x_1 - x_2 \\ &\text{Subject to: } f_1(x_1, x_2) = x_1^2 + x_2^2 - 1 \leq 0, \\ &\qquad\qquad f_2(x_1, x_2) = -x_1 + x_2^2 \leq 0. \end{aligned}$$

Using the interior point method, for the case $\rho^{(k)} = \frac{1}{2^k}$, and the DFP method with the precision 10^{-7} in the generated unconstrained optimization, we obtain the following results:

k	$x_1^{(k)}$	$x_2^{(k)}$	$Q(\vec{x}^{(k)})$
0	0.5	0.5	-1.
1	0.63085851	0.09223888	2.56860712
2	0.70733854	0.67997758	-1.35642624
3	0.70670616	0.70666650	-1.41252669
4	0.70709348	0.70709348	-1.41416037
5	0.70710636	0.70710636	-1.41421188
6	0.70710676	0.70710676	-1.41421350
7	0.70710677	0.70710677	-1.41421352

Table 3.

On the other hand, by means of the well-known traditional implementation, the interior point method produces the following results [19]:

k	$x_1^{(k)}$	$x_2^{(k)}$	$Q(\vec{x}^{(k)})$
2	0.6884721	0.3952927	-1.0837648
3	0.7106337	0.3713147	-1.0819484
4	0.7349830	0.4535905	-1.1785735
5	0.7276601	0.5228195	-1.2504796
6	0.7251426	0.5758051	-1.3009477
7	0.7203736	0.6143570	-1.3358093
8	0.7151591	0.6446030	-1.3597621
9	0.7105652	0.6567412	-1.3763306
10	0.7072276	0.6803946	-1.3876222
14	0.7048593	0.7027896	-1.4076489
18	0.7063525	0.7062206	-1.4125731
30	0.7070938	0.7070938	-1.4141876
53	0.7071067	0.7071067	-1.4142134

Table 4.

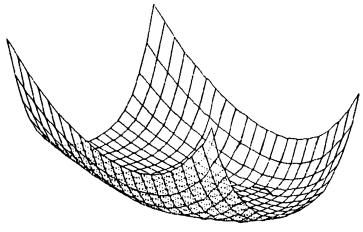
It is evident that the functional implementation requires less iterations with respect to a traditional implementation. For example, precision 10^{-6} in Table 3. is achieved in 5th iteration, and in Table 4 in 53th iteration.

Using the Rockafellar's method with $\lambda_0 = 1$, $a = 10$, we obtain the following results:

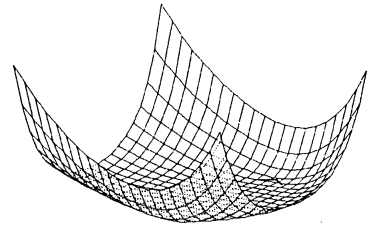
k	$x_1^{(k)}$	$x_2^{(k)}$	$Q(\vec{x}^{(k)})$
0	1.	1.	-2.
1	0.70705940	0.70705056	-1.41410996
2	0.70711190	0.70710163	-1.41421353
3	0.70710735	0.70710626	-1.41421361
4	0.70711125	0.70710225	-1.41421351
5	0.70710709	0.70710651	-1.41421361
6	0.70710707	0.70710648	-1.41421356
7	0.70710679	0.70710676	-1.41421356

Table 5.

Note that in the generated unconstrained minimizations are used search method (DSK-Powell) with the precision 10^{-7}. trajectories generated by $\lambda_0 = 1$ and two different values of parameter a are illustrated in the following figures:



$\rho = 1.0$



$\rho = 10$

Analogous results, given by means of traditional implementation are presented in Table 6 (see [19]):

k	$x_1^{(k)}$	$x_2^{(k)}$
1	0.70200159	0.702000159
2	0.70701744	0.70701746
3	0.70710524	0.70710522
4	0.70710676	0.70710675
5	0.70710678	0.70710678
6	0.70710678	0.70710680
7	0.70710678	0.70710679

Table 6.

Observed decreasing of the number of iterations is implied by symbolic computations of derivatives, with respect to their inexact numerical computations.

A smaller number of iterations implies also a smaller number of function and gradient calls. But, each iteration in the functional implementation requires a

transformation of a constrained optimization problem, given in an appropriate internal form into the internal form of the corresponding unconstrained problem. Also, interpreter-based languages are slower than the languages implemented by compilers.

6. Conclusions

Our tendency is primarily to improve implementation of the constrained optimization methods, which are written in procedural programming languages. Also, our motivation is the absent of functions for nonlinear constrained optimization which are available in MATHEMATICA. The improvements are ensured primarily applying possibility of symbolic processing of the functional programming languages PC SCHEME and MATHEMATICA. Of course, similar principles are valid for the other functional programming languages. But, we prefer PC SCHEME and MATHEMATICA, because of their ability in symbolic processing as well as in numeric processing. The main purpose is to point out that the proper selection of the programming language in nonlinear optimization is not FORTRAN, but a language applicable in symbolic processing and powerful in numerical computations.

We improve greater part of the criteria used as the best in evaluating a nonlinear programming algorithm [1]:

1. Size (dimensionality, number of inequality and/or equality constrains) of the problem. The inequality and equality constrains are stored in the list, so that its number is not limited in advance. On the other hand, in FORTRAN and C, if the set of constrains is stored in the array $X(1),\ldots,X(N)$, then allowance has been made for a N-dimensions problem. In the programming language C it is possible to implement a list of constrains by means of linked list, using dynamic memory allocations.

2. Simplicity of use (time required to introduce data and functions into the computer program). The objective function is given as an arbitrary PC SCHEME arithmetic expression, incorporated in the internal form of the problem.

3. Simplicity of computer program to execute the algorithm. Using the functional programming languages we ensure the following.

- Possibility to use the objective function and constrains, without a lexical or syntax analysis.

- Simple implementation of the partial derivatives of the objective function.

- An elegant method of transformation of a given internal form which represents a constrained program into the internal form of the corresponding unconstrained program.

This paper is a contribution toward a new approach to implementation of constrained optimization methods. This approach can be called *symbolic implementation.*

References

[1] David, M.H., *Applied Nonlinear Programming*, McGraw-Hill Book Company, 1972.

[2] Dixon, L.C. and Price, R.C., *Truncated Newton method for sparse unconstrained optimization using automatic differentiation*, J. Optimiz. Theory Appl. **60** (1989), 261–275.

[3] Griewank, A. and Corliss, G.F., *Automatic differentiation of algorithms*, Proceedings of the first SIAM Workshop on Automatic Differentiation, SIAM, Philadelphia, 1991.

[4] Henessey, L.W., *Common LISP*, McGraw-Hill Book Company, 1989.

[5] Hyvönen, E. and Seppänen, J., *Introduction to LISP and functional programming*, Moskva, "Mir", 1990. (In Russian)

[6] Jacoby, S.L.S., Kowalik, J.S. and Pizzo, J.T., *Iterative methods for nonlinear optimization problems*, Prentice-Hall, Inc, Englewood, New Jersey, 1977.

[7] Parker, T.S. and Chua, L.O., *INSITE- a software toolkit for the analysis of nonlinear dynamic systems*, Proceedings of the IEEE **75** (1987), 1081–1089.

[8] Press, W.H.; Flannery, B.P.; Teukolsky, S.A. and Vetterling, W.T., *Numerical recipes in C*, Cambridge University Press, New York, Melbourne, Sydney, 1990.

[9] Press, W.H.; Flannery, B.P.; Teukolsky, S.A. and Vetterling, W.T., *Numerical recipes*, Cambridge University Press, New York, Melbourne, Sydney, 1986.

[10] Richard, W.S., *LISP, Lore and Logic*, Springer-Verlag, 1990.

[11] Smith, J.D., *An Introduction to Scheme*, Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[12] Stanimirović, P. and Rančić, S., *Unidimensional search optimization in LISP*, Proceedings of the II Mathematical Conference in Priština (1996), 253–262.

[13] Stanimirović, P. and Rančić, S., *Unconstrained optimization in LISP*, in: XI Conference on Applied Mathematics, Budva (1996), 355–362.

[14] Stojanov, S., *Methods and Algorithms for Optimization*, Drzavno izdatelstvo, Tehnika, Sofija, 1990. (In Bulgarian)

[15] Sussman, G.J., *Structure and interpretation of computer programs*, MIT Press, Cambridge, Massachusetts, 1985.

[16] Wilensky, R., *Common LISPcraft*, Norton, New York, 1986.

[17] Wolfram, S., *Mathematica: a system for doing mathematics by computer*, Addison-Wesley Publishing Co, Redwood City, California, 1991.

[18] Wolfram, S., *Mathematica Book, Version 3.0*, Wolfram Media and Cambridge University Press, 1996.

[19] Zlobec, S. and Petrić, J., *Nonlinear programming,*, Naučna Knjiga, Beograd, 1989. (In Serbian)

Author's address: University of Niš, Faculty of Philosophy, Department of Mathematics, Ćirila i Metodija 2, 18000 Niš, Yugoslavia

Received: July 7, 1997