

SECOND ORDER OPTIMIZATION METHODS IN LISP*

Predrag STANIMIROVIĆ, Svetozar RANČIĆ

*University of Niš, Faculty of Philosophy,
Department of Mathematics, Ćirila i Metodija 2,
18000 Niš, Yugoslavia*

Abstract: We describe the implementation of main second-order gradient methods for unconstrained nonlinear optimization in the programming language LISP. This approach ensures significant improvements of the known procedures in procedural programming languages.

Keywords: Nonlinear programming, Newton's methods, variable metric methods, LISP.

1. INTRODUCTION

A general introduction to fundamental second-order derivative methods for unconstrained minimization is given in [2], [9], [12]. In Newton's method, the transition from the k th approximation $\bar{x}^{(k)}$ of the local minimum for the objective function $Q(\bar{x})$, to the new approximation $\bar{x}^{(k+1)}$ is defined by

$$\bar{x}^{(k+1)} = \bar{x}^{(k)} - [\nabla^2 Q(\bar{x}^{(k)})]^{-1} \nabla Q(\bar{x}^{(k)}) = \bar{x}^{(k)} - H^{-1}(\bar{x}^{(k)}) \nabla Q(\bar{x}^{(k)}),$$

where

$$\nabla Q(\bar{x}^{(k)}) = \text{grad } \nabla Q(\bar{x}^{(k)}) = \begin{bmatrix} \frac{\partial Q(\bar{x}^{(k)})}{\partial x_1} \\ \dots \\ \frac{\partial Q(\bar{x}^{(k)})}{\partial x_n} \end{bmatrix}$$

denotes the gradient of the objective function $Q(\bar{x})$ at point $\bar{x}^{(k)}$, and

* 1991 Mathematics Subject Classification. 90C30, 68N15

$$\nabla^2 Q(\bar{x}^{(k)}) = \mathbf{H}(\bar{x}^{(k)}) = \begin{bmatrix} \frac{\partial^2 Q(\bar{x}^{(k)})}{\partial x_1^2} & \dots & \frac{\partial^2 Q(\bar{x}^{(k)})}{\partial x_1 \partial x_n} \\ \dots & \dots & \dots \\ \frac{\partial^2 Q(\bar{x}^{(k)})}{\partial x_n \partial x_1} & \dots & \frac{\partial^2 Q(\bar{x}^{(k)})}{\partial x_n^2} \end{bmatrix}$$

represents the Hessian matrix of $Q(\bar{x})$.

We can suggest at least two advantages ensured by implementing this method in the functional programming languages [3], [7], [8], [10]:

1. Possibility of direct application of the implementation procedure to an arbitrary objective function, given as a formal parameter;
2. Possibility of the functional programming languages in symbolic derivation.

Also, we describe the implementation of the modified Newton method, defined by the parameter for the step length $h^{(k)}$:

$$\bar{x}^{(k+1)} = \bar{x}^{(k)} - h^{(k)} [\nabla^2 Q(\bar{x}^{(k)})]^{-1} \nabla Q(\bar{x}^{(k)}).$$

The search direction is given by $\bar{s}^{(k)} = -[\nabla^2 Q(\bar{x}^{(k)})]^{-1} \nabla Q(\bar{x}^{(k)})$, and the parameter $h^{(k)}$ is defined by

$$h^{(k)} = \min_h Q(\bar{x}^{(k)} - h[\nabla^2 Q(\bar{x}^{(k)})]^{-1} \nabla Q(\bar{x}^{(k)})).$$

In addition to the above - mentioned advantages, during the implementation of the modified Newton method in the LISP environment, we would mention the following:

3. It is considerably convenient problem for functional programming languages to form the new function, defined by

$$F(h) = \min_h Q(\bar{x}^{(k)} - h[\nabla^2 Q(\bar{x}^{(k)})]^{-1} \nabla Q(\bar{x}^{(k)})).$$

From the class of methods termed *variable metric*, we implemented the Davidon-Fletcher-Powell method [2], [9], [12]. In this method the inverse of the Hessian matrix $H^{-1}(\bar{x}^{(k)})$ is approximated by the matrix H_k , computed using information from only first-order derivatives. A new \bar{x} from the preceding stage can be computed by means of

$$\bar{x}^{(k+1)} = \bar{x}^{(k)} - h^{(k)} H_k \nabla Q(\bar{x}^{(k)}),$$

where

$$h^{(k)} = \min_h Q(\bar{x}^{(k)} - hH_k \nabla Q(\bar{x}^{(k)})) .$$

For the implementation language we selected SCHEME [8], a dialect of LISP, which is one of the most versatile programming languages available today, useful for a variety of programming projects. Standard SCHEME is applicable in symbolic processing as well as in numerical processing. The largest possible integer varies among implementations, but is typically very large. The precision of the results given by expressions involving real number operations is typically quite supportive for scientific computation.

For this purpose, we preferred the programming language LISP when implementing the methods of unconstrained nonlinear programming. As far as we know, such an approach has not been, employed before.

Note that in the programming package MATHEMATICA [5], [11] a few functions for numerical optimization are available. The function *FindMinimum*, for finding a local minimum, starts at the specified points, then follows the path of steepest descent on the surface.

FindMinimum[*f*, {*x*, *x*₀}] search for a local minimum of the function *f*, starting from the point $x = x_0$;

FindMinimum[*f*, {*x*, *x*₀}, {*y*, *y*₀}, ...] search for a local minimum in a function of several variables;

FindMinimum[*f*, {*x*, {*x*₀, *x*₁}}] search for a local minimum using *x*₀ and *x*₁ as the first two values of *x* (this form must be used if symbolic derivatives of *f* cannot be found);

FindMinimum[*f*, {*x*, *x*_{start}, *x*_{min}, *x*_{max}}] search for a local minimum, stopping the search if *x* ever gets outside of the range [*x*_{min}, *x*_{max}].

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

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*.

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

The paper is organized as follows: In Section 2 we briefly describe implementation of Newton's method, the modified Newton method and the Davidon-

Fletcher-Powell method in LISP environment. In Section 3 a few computer results are reported, and relations with the corresponding results obtained by means of the procedural programming languages are discussed.

2. IMPLEMENTATION DETAILS

In this section we describe the main details of implementations procedures.

2.1. Internal form of the objective function

The selected real objective function $Q(x_1, \dots, x_n) = Q(\vec{x})$ is represented by a list, denoted q , of two elements:

```
((<function>) (<largs>))
```

The first element is the selected PC SCHEME arithmetic function and the second represents its argument list. Consequently, the *function* contained in the internal form q can be selected by the expression $(car\ q)$, and the corresponding *parameter list* by the expression $(cadr\ q)$. This internal representation q of a given objective function can be transformed into the corresponding *lambda-expression*:

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

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

```
(apply fun v)
```

The argument list can be transformed into the corresponding vector $varg$:

```
(set! varg (list->vector (cadr q)))
```

We mentioned before that in the built-in functions for numerical optimization, available in MATHEMATICA, the arguments of the objective function must also be specified.

2.2. Symbolic differentiation and declarations

The symbolic differentiation in our package is implemented in the function *deriv*, extending the corresponding functions in [1], [4], [7]. The call of the function *deriv* is of the form

```
( deriv <function> <var> )
```

where $\langle function \rangle$ denotes the internal representation of the objective function and $\langle var \rangle$ is any selected variable from the parameter list.

The vector $grad = \nabla Q(\bar{x}^{(k)})$ must be declared by means of the following initialization:

```
(grad (make-vector (length (cadr q)) 0))
```

The vectors $hest = \nabla^2 Q(\bar{x}^{(k)})$ and $hesti = [\nabla^2 Q(\bar{x}^{(k)})]^{-1}$ are specified by the following initializations:

```
(hest (make-vector (length (cadr q)) 0))
```

```
(hesti (make-vector (length (cadr q)) 0))
```

Moreover, each element of these vectors can be defined as a vector of unspecified elements:

```
(do ((j 0) (n (length (cadr q))))
    ((= n j) ())
    (vector-set! hest j (make-vector n))
    (vector-set! hesti j (make-vector n))
    (set! i (+ i 1)))
```

2.3. Elementary matrix algebra in LISP

We begin implementation of Newton's method by describing a few useful operations from the matrix algebra. The addition, subtraction and multiplication (and other dyadic operations) on two given vectors x, y can be implemented in the following common functional, which uses the function op as argument:

```
(define (vecop x y op)
  (do ((i 0) (n (vector-length x)))
      ((= n i) vcp)
      (vector-set! vcp i (op (vector-ref x i)
                             (vector-ref y i)))
      (set! i (+ i 1))))
```

where $op \in \{+, -, *\}$.

In a similar way the function $(matop a b op)$ can be written for the multiplication, addition and subtraction of two matrices a and b , where $op \in \{*, +, -\}$. The function implementing product of two matrices in LISP is written in [3].

The multiplication of a given matrix *mat* and a given vector *vec* is implemented as follows:

```
(define (matvec mat vec)
  (do ((i 0) (n (vector-length vec))(m(vector-length mat))
      (vcp (make-vector (vector-length mat))) (s 0))
      ((= m i) vcp)
      (set! s 0)
      (do ((j 0)
          ((= n j) ()))
          (set! s (+ s (* (vector-ref (vector-ref mat i) j)
                          (vector-ref vec j))))
          (set! j (+ j 1)))
      (vector-set! vcp i s)
      (set! i (+ i 1))))
```

2.4. Computation of gradient and Hessian

The gradient of Q at a given point *pt* can be computed as follows:

```
(define (gradientin q pt)
  (do ((i 0)(var (list->vector (cadr q)))(n (length (cadr q)))
      (nabl (make-vector (length (cadr q))))
      ((= n i) nabl)
      (vector-set! nabl i (apply (eval (list 'lambda (cadr q)
      (deriv (car q) (vector-ref var i)))) (vector->list pt)))
      (set! i (+ i 1))))
```

Then the vector $grad = \nabla Q(\bar{x}^{(k)}) = \nabla Q(vx0)$ is equal to

```
(set! grad (gradientin fun vx0))
```

In a similar way the Hessian matrix of the function Q can be formed at point *pt*

```
(define (hessin q pt)
  (let ((i 0)(var(list->vector (cadr q)))(n(length (cadr q)))
      (hes (make-vector (length (cadr q)) 0))(f 1))
    (do ((i 0)
        ((= n i) ()))
```



```

      ((= n j) ())
      (vector-set! (vector-ref hesti i) j (if (= i j) 1 0))
      (set! j (+ j 1)))
    (set! i (+ i 1)))

```

Step 2. Perform the extended Gauss-Jordan transformation

$[hest | I_n] \rightarrow [I_n | hesti]$.

Step 2.1. A for loop, determined by the index $i = 0, \dots, n-1$:

```

(do ((i 0)
    ((= n i) ()))

```

In the body of the cycle perform steps A and B.

A. Compute

```

  for j ← 0 to n-1
    hest(i,j) ← hest(i,j)/hest(i,i)
    hesti(i,j) ← hesti(i,j)/hest(i,i)

  (set! el (vector-ref (vector-ref hest i) i))
  (do ((j 0)
      ((= n j) ()))
      (vector-set! (vector-ref hest i) j
                  (/ (vector-ref (vector-ref hest i) j) el))
      (vector-set! (vector-ref hesti i) j
                  (/ (vector-ref (vector-ref hesti i) j) el))
      (set! j (+ j 1)))

```

B. Compute

```

  for j ← 0 to n-1
    hest(j,k) ← hest(j,k)-hest(j,k)*hest(j,i)/hest(i,i)
    hesti(j,k) ← hesti(j,k)-hesti(j,k)*hest(j,i)/hest(i,i)

  (do ((j 0)
      ((= n j) ()))
      (set! elp (vector-ref (vector-ref hest i) i))
      (set! elk (vector-ref (vector-ref hest j) i))
      (set! koef (/ elk elp))

```

```

(if (not (= i j))
  (begin
    (do ((k 0)
        ((= n k) ())
        (vector-set! (vector-ref hest j) k
                     (- (vector-ref (vector-ref hest j) k)
                        (* (vector-ref (vector-ref hest i) k) koef))))
      (vector-set! (vector-ref hesti j) k
                   (- (vector-ref (vector-ref hesti j) k)
                      (* (vector-ref (vector-ref hesti i) k) koef))))
      (set! k (+ k 1))))
    (set! j (+ j 1)))
  (set! i (+ i 1)))

```

Finally, the transition from point $vx0 = \bar{x}^{(k)}$ to the next approximation $vx1 = \bar{x}^{(k+1)}$ is implemented as follows:

Step 1. Compute

$$vt = [\nabla^2 Q(\bar{x}^{(k)})]^{-1} \nabla Q(\bar{x}^{(k)}) = hesti \cdot grad :$$

```
(set! vt (matvec hesti grad))
```

Step 2. Compute $vx1 = vx0 - vt$.

```
(set! vx1 (vecop vx0 vt -))
```

2.6. Implementation of the modified Newton method

The crucial point in the implementation of the modified Newton method is determination of the new function

$$f(\theta) = Q(\bar{x}^{(k)} - \theta[\nabla^2 Q(\bar{x}^{(k)})]^{-1}[\nabla Q(\bar{x}^{(k)})]^T) = Q(vx0 - \theta * g).$$

This goal can be achieved substituting the vector $vx0 = \bar{x} = (x_1, \dots, x_n)$ by the vector $vx0 - h * g$, i.e. by means of the substitutions

$$x_i \leftarrow (vx0 - h * g), \quad i = 0, \dots, n-1$$

in the expression $(car q)$. The function performing these substitutions is defined in the standard way [3], [10]:

```
(define (subst x y p)
  (cond ((equal? p y) x)
        ((atom? p) p)
        (t (cons (subst x y (car p)) (subst x y (cdr p)))
  ) )
```

The routine which forms the function $f(\theta)$ is defined in the following code:

```
(set! f (car q))
(do ((i 0)
    ((= n i) ())
    (set! argstari (vector-ref varg i))
    (set! vt (vector-ref vx0 i))
    (set! vx (vector-ref g i))
    (set! argnovi (list '- vt (list '* 'teta vx)))
    (set! f (subst argnovi argstari f))
    (set! i (+ i 1)))
```

Now, an arbitrary function implementing unidimensional optimization can be invoked using the parameter representing the internal form for $f(\theta)$:

```
(list f (list 'teta))
```

in the place of the internal form of the objective function. It is recommended that the precision in the unidimensional optimization be at least equivalent to that returned for termination of the main algorithm [2].

2.7. Davidon-Fletcher-Powell method

During the implementation of the Davidon-Fletcher-Powell method we use the following function, performing multiplication of the vector-column $x \in \mathbf{C}^{m \times 1}$ and the vector-row $y \in \mathbf{C}^{1 \times n}$.

```
(define (vecvec x y)
  (let ((i 0) (m (vector-length x)) (n (vector-length y))
        (a (make-vector (vector-length x) 0)))
    (do ((i 0)
        ((= m i) ())
        (vector-set! a i (make-vector n))
        (set! i (+ i 1)))
```

```

(do ((i 0)
    ((= m i) a)
    (do ((j 0)
        ((= n j) ())
        (vector-set! (vector-ref a i) j
                     (* (vector-ref x i)(vector-ref y j)))
        (set! j (+ j 1)) )
    (set! i (+ i 1))))))

```

It is not difficult to write the following functions:

(skalpr x y), for computing the scalar product of the vectors x and y ;
 (numvec r v), which multiplies elements of given vector v by a number r ;
 (nummat r a), which multiplies elements of given matrix a by a number r .

```

(define (skalpr x y)
  (let ((r 0) (i 0) (n (vector-length x)))
    (do ()
        ((<= n i) r)
        (set! r (+ r (* (vector-ref x i)(vector-ref y i))))
        (set! i (+ i 1))))))

```

```

(define (numvec r v)
  (let ((i 0) (n (vector-length v)))
    (do ()
        ((<= n i) v)
        (vector-set! v i (* r (vector-ref v i)))
        (set! i (+ i 1))))))

```

```

(define (nummat r a)
  (let ((i 0) (m (vector-length a))(n 0))
    (set! n (vector-length (vector-ref a 1)))
    (do ()
        ((= m i) a)
        (do ((j 0)
            ((= n j) ())
            (vector-set! (vector-ref a i) j
                         (* r (vector-ref (vector-ref a i) j)))
            (set! j (+ j 1))

```

(set! i (+ i 1))))))

Now, the main steps in the Davidon-Fletcher-Powell method can be implemented as follows.

Step 1. Select an internal form *fun* of a function, an initial approximation $vx0 = \bar{x}^{(0)}$ and compute the following values:

$grad = g0 = \nabla Q(\bar{x}^{(0)})$ by means of

(set! g0 (gradientin fun vx0))

and $h = H_0 = I_n$, using the known technique.

The stopping criterion is given by the expression

(< (sqrt (skalpr vx0 vx0)) eps)

where *eps* is a small real number.

Step 2. Compute $d = d_k = H_k g_k = H_k \nabla Q(\bar{x}^{(k)})$:

(set! d (matvec h g0))

Step 3. Perform the unidimensional optimization

$$\theta_k = \min_{\theta} Q(\bar{x}^{(k)} + \theta d_k) = \min_{\theta} Q(vx0 + \theta d)$$

using the known principles from the modified Newton algorithm.

Step 4. Compute $z_k = \theta d_k = \text{teta} * d$ and $\bar{x}^{(k+1)} = vx1 = \bar{x}^{(k)} + z_k = vx0 + \text{teta} * d$.

(set! z (numvec teta d))

(set! vx1 (vecop vx0 z +))

Step 5. Compute the new approximation H_{k+1} by means of

$$g_{k+1} = \nabla Q(\bar{x}^{(k+1)}), \quad w_k = g_{k+1} - g_k$$

$$A_k = \frac{z_k z_k^T}{(z_k, w_k)}, \quad B_k = \frac{H_k w_k w_k^T H_k^T}{(z_k, H_k w_k)}$$

$$H_{k+1} = H_k + A_k - B_k.$$

(set! g1 (gradientin fun vx1))

(set! w (vecop g1 g0 -))

```
(set! a (nummat (/ 1 (skalpr z w)) (vecvec z z))
(set! p (matvec h w))
(set! b (nummat (/ 1 (skalpr w p)) (vecvec p p))
(set! h (matop h a +)) (set! h (matop h b -))
```

2.8. Formal parameters

The formal parameters of the described procedures are:

q: the internal form of the objective function;
x0: the list containing initial values for variables;
eps: a small real number, defining precision.

The initial point $vx0 = \bar{x}^{(0)}$ is equal to

```
(list->vector x0)
```

3. COMPUTATIONAL EXPERIENCE

Example 3.1. Newton-like methods and the Davidon-Fletcher-Powell method are illustrated for the problem

$$\text{minimize : } x_1^4 + x_1^3 - x_1 + x_2^4 - x_2^2 + x_2 + x_3^2 - x_3 + x_1 x_2 x_3 ,$$

using the starting point given by the list $x0 = (1 \ -1 \ 1)$. The internal form *q* of the objective function is

```
((- (+ (expt x1 4)(expt x1 3)(expt x2 4) x2 (expt x3 2)
(* x1 x2 x3) )
x1 (expt x2 2) x3)
(x1 x2 x3) )
```

The functions implementing the Newton methods and the Davidon-Fletcher-Powell method, applied with the precision $eps = 10^{-5}$, give results analogous with the results derived in [12] for these methods:

$$x_1^{(4)} = 0.57086 , x_2^{(4)} = -0.93956 , x_3^{(4)} = 0.76818 , F(\bar{x}^{(4)}) = -1.91177$$

Moreover, we can obtain results with greater precision. Using Newton's method with the precision $eps = 10^{-12}$, we obtain the following results:

	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$	$Q(\bar{x}^{(k)})$
0.	1	-1	1	-1
1.	0.710365853658	-0.954268292682	0.832317073170	-1.836689396285
2.	0.592551924079	-0.941653842008	0.778246318947	-1.910144342363
3.	0.571502823844	-0.939615161558	0.768474902884	-1.911770770299
4.	0.570856568381	-0.939555960428	0.768175826556	-1.911772189070
5.	0.570855968375	-0.939555906203	0.768175548339	-1.911772189071
6.	0.570855968375	-0.939555906203	0.768175548339	-1.911772189071

Application of the modified Newton method with the precision $eps = 10^{-6}$ gives the following:

	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$	$Q(\bar{x}^{(k)})$
0.	1	-1	1	-1
1.	0.571451	-0.932334	0.751892	-1.911337
2.	0.570853	-0.939589	0.768111	-1.911772
3.	0.570855	-0.939555	0.768175	-1.911772

Application of the Davidon-Fletcher-Powell method with the precision $eps = 10^{-6}$ produces the following approximations:

	$x_1^{(k)}$	$x_2^{(k)}$	$x_3^{(k)}$	$Q(\bar{x}^{(k)})$
0.	1	-1	1	-1
1.	0.593889	-0.916553	0.916553	-1.886319
2.	0.582650	-0.938019	0.768282	-1.911233
3.	0.571404	-0.940625	0.769439	-1.911766
4.	0.570865	-0.939551	0.768185	-1.911772
5.	0.570855	-0.939555	0.768175	-1.911772

Note that in the unidimensional optimization the Davies-Swan-Campey method [2], [9] is used, with the precision $eps/5$.

4. CONCLUSIONS

Our tendency is primarily to improve the implementation of second-order (and other) optimization methods, without changing their essence. The improvements are ensured primarily applying the possibility of symbolic processing of the functional programming language PC SCHEME. Of course, similar principles are valid for the

other functional programming languages. But, we prefer PC SCHEME because of its 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.

REFERENCES

- [1] Abelson, H., and Sussbam, G.J., *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, Massachusetts, 1985.
- [2] David, M.H., *Applied Nonlinear Programming*, Mc-Graw-Hill Book Company, 1972.
- [3] Henessey, L.W., *Common LISP*, McGraw-Hill Book Company, 1989.
- [4] Hyvönen, E., and Seppänen, J., *Introduction to LISP and Functional Programming*, Mir, Moskva, 1990 (in Russian).
- [5] Krejić, N., and Herceg, Dj., *Mathematics and MATHEMATICA, Računari u univerzitetskoj praksi*, Novi Sad, 1993 (in Serbian).
- [6] Milovanović, G.V., *Numerical Analysis, Part I*, Naučna Knjiga, Beograd, 1985 (in Serbian).
- [7] Richard, W.S., *LISP, Lore and Logic*, Springer-Verlag, 1990.
- [8] Smith, J.D., *An Introduction to Scheme*, Prentice Hall, Englewood Cliffs, New Jersey 1988.
- [9] Stojanov, S., *Methods and Algorithms for Optimization*, Drzavno izdatelstvo, Tehnika, Sofija, 1990 (in Bulgarian).
- [10] Wilensky, R., *Common LISPcraft*, Norton, New York, 1986.
- [11] Wolfram, S., *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley Publishing Co, Redwood City, California, 1991.
- [12] Zlobec, S., and Petrić, J., *Nonlinear Programming*, Naučna Knjiga, Beograd, 1989 (in Serbian).