Yugoslav Journal of Operations Research
9 (1999), Number 2, 285-300

SYMBOLIC IMPLEMENTATION OF THE HOOKE-JEEVES METHOD*

Predrag S. STANIMIROVIĆ, Milan B. TASIĆ, Miroslav RISTIĆ

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

Abstract: In this paper we describe the symbolic implementation of various modifications of the Hook-Jeeves method in the programming language MATHEMATICA. A few numerical results are reported as well as several graphical illustrations.

Keywords: Hook-Jeeves method, MATHEMATICA, symbolic computation.

1. INTRODUCTION

The Hooke-Jeeves method is a well-known method for unconstrained minimization. It does not use derivatives. Three modifications of this method are described in [2], [6], [8], [17].

Computational systems for the numerical implementation of optimization methods are known. They are written in procedural programming languages, mainly in FORTRAN [2], [6], [9], [14] and C [11]. But, procedural programming languages are not convenient for the symbolic implementation of optimization methods. Two modifications of the Hooke-Jeeves method can be used as typical examples.

In this paper we investigate the application of the *symbolic computation* of the programming language MATHEMATICA in the implementation of the Hooke-Jeeves method. The notion of the *symbolic implementation* of optimization methods assumes the possibility of using *symbolic computation* in the implementation of these methods. For example, symbolic computation in MATHEMATICA is described in [3], [5], [10],

---

*1991 *Mathematics Subject Classification*. 90C30, 68N15.

[15], [16]. From the symbolic computation available in MATHEMATICA we use mainly the following:

algebraic manipulations, which include computations with symbols and operations on algebraic expressions;

manipulating equations, which means the solution to various equations and the elimination of unknown variables;

symbolic differentiation;

the application of *transformation rules* in the following form:

expr/.lhs–>rhs apply a transformation rule to *expr*,
expr/.{lhs1–>rhsl, lhs2–>rhs2,...} apply a sequence of rules to *expr*.

MATHEMATICA is the world's only fully integrated environment for technical computing [15], [16]. It incorporate a range of programming paradigms, such as the following: procedural programming, list-oriented programming, functional programming, rule-based programming, object oriented programming, string-based programming, and mixed programming paradigms [3], [5], [10], [15], [16]. Hence, by means of MATHEMATICA, we can write every program in its most natural way.

The paper is organized as follows. In the second section we describe three varieties of the Hooke-Jeeves method. The third section describes the *symbolic implementation* of these methods. In the last section we develop several numerical examples and graphical illustrations.

2. DESCRIPTION OF THE METHOD AND MOTIVATION

For the sake of completeness, we shall briefly describe the Hooke-Jeeves method for unconstrained optimization [2], [6], [8], [17]. Generally, the Hooke-Jeeves algorithm consists of two major phases: an "exploratory search" around the base point and a "pattern search" in a direction selected for minimization.

The original Hooke-Jeeves method is introduced in [7] and is also described in [6] and [17].

Step 1. Initial values for all coordinates of some point $\mathbf{x}_0 = \mathbf{x}_B^0$ must be provided as well as the initial incremental change δ.

Step 2. Then a type I exploratory search is performed. Each variable is changed in rotation, one at a time, by incremental amounts, until the parameters have been properly changed.

More precisely, in the kth step, in the "basic point" $\mathbf{x}_B^k = (x_1^{(k)} \ldots\ldots x_n^{(k)})$ let $x_1^{(k)}$ be changed by some amount δ, so that $x_1^{(k+1)} = x_1^{(k)} + \delta$. If the value $Q(\mathbf{x})$ is improved, then $x_1^{(k)} + \delta$ is adopted as the new element in x_B^k. Otherwise, $x_1^{(k)}$ is changed by $-\delta$, and the value of the objective function $Q(\mathbf{x})$ is checked again. If the value of $Q(\mathbf{x})$ is not improved by either $\pm\delta$, then we keep the old value of the coordinate $x_1^{(k)}$. Then $x_2^{(k)}$ is changed by some amount $\pm\delta$, and so on, until all the independent variables have been changed and the exploratory search is completed. For each step or move in an independent variable, the value of the objective function is compared with the value at the previous point. If the objective function is improved for the given step, then the old value of the objective function is replaced by the new value of the objective function. Otherwise, if the step is unsuccessful, the old value is retained.

In this way, the type I exploratory search is finished when the search is performed using each of the independent variables. This produces a new basic point $\mathbf{x}_B^{k+1}$.

Step 3. After making one (or more) exploratory searches, a "pattern search" is made. A new point t_0^{k+2}, defined by

$$\mathbf{t}_0^{k+2} = \mathbf{x}_B^{k+1} + (\mathbf{x}_B^{k+1} - \mathbf{x}_B^k) \tag{2.1}$$

must be formed. Then the first successful point from t_0^{k+2} defines a new basic point $\mathbf{x}_B^{k+2}$.

Step 4. An exploratory search conducted after a pattern search is termed a type II exploratory search. The success or failure of a pattern search is established after the type II exploratory search is completed.

A. If $Q(\mathbf{x})$ is not improved after the type II exploratory search, the pattern search is said to fail. Then the step δ is reduced gradually.

B. If $Q(\mathbf{x})$ is improved after the type II exploratory search, the last point produced by the type II exploratory search is termed the new "basic point" $\mathbf{x}_B^{k+2}$.

Step 5. The process is terminated when the value of variable δ is less than a small prespecified number.

An alternative to equation (2.1) is to solve a one-dimensional problem in the direction $x_B^{k+1} - \mathbf{x}_B^k$ in order to generate a new basic point t_0^{k+2} [8]:

$$\mathbf{t}_0^{k+2} = \mathbf{x}_B^{k+1} + h_k(\mathbf{x}_B^{k+1} - \mathbf{x}_B^k), \tag{2.2}$$

where step h_k is defined by

$$h_k = \min_h F(h) = \min_h Q(x_B^{k+1} + h(\mathbf{x}_B^{k+1} - \mathbf{x}_B^k)) . \tag{2.3}$$

The optimal value h_k of step h can be computed using an arbitrary unidimensional optimization method. This is the essence of the first modification of the Hooke-Jeeves method.

The second modification of the Hooke-Jeeves method is described in [2]. Let $d_1,\ldots,d_n$ be any selected appropriate vectors. Then the major steps in this modification are as follows:

Step 1. Initial values: Select a real number $\varepsilon > 0$, initial point x_1 and set the initial values $k = 0$, $y_1 = x_1$.

Step 2. Exploratory search: For each $i = 1,\ldots,n$ compute

$$y_{i+1} = y_i + \lambda_i d_i \text{, where } \lambda_i = \min_\lambda G(\lambda) = \min_\lambda Q(y_i + \lambda d_i) . \tag{2.4}$$

Step 3. Set $x_{k+1} = y_{n+1}$. If $\| x_{k+1} - x_k \| < \varepsilon$ then stop. Otherwise, go to Step 4.

Step 4. Compute

$$y_1 = x_{k+1} + h_k(x_{k+1} - x_k), \tag{2.5}$$

where

$$h_k = \min_h H(h) = \min_h Q(x_{k+1} + h(x_{k+1} - x_k)) . \tag{2.6}$$

Go to Step 2.

What is the need of the *symbolic implementation* of the Hooke-Jeeves method? Above all, the programming language used must be capable of generating formulas which define functions $F(h)$, $G(\lambda)$ and $H(h)$. This requires a language which is able to process arbitrary formulas, known only at the run time.

Also, it is desirable to use a language that is powerful in numerical computations in order to avoid truncation errors in numerical computations.

Finally, we also need good graphical illustrations of the generated results.

Therefore, we have just reported at least three reasons to use the MATHEMATICA language in the implementation of the Hooke-Jeeves method.

3. IMPLEMENTATION

We suggest the following advantages which appear during the symbolic implementation of the Hooke-Jeeves method and its modifications in functional programming languages.

1. It is possible to take an arbitrary objective function, which is not defined by subroutines, as a formal parameter under the program control.
2. For a given point $xm = \{x_1, \ldots, x_n\}$, it is possible to construct the following function

$$P(\delta) = Q(x_1, \ldots, x_{i-1}, x_i \pm \delta, x_{i+1}, \ldots, x_n) \tag{2.7}$$

without previous definition of the function Q.

3. We can construct, in a natural way, new objective functions $F(h)$, $G(\lambda)$ and $H(h)$, which are defined in (2.3), (2.4) and (2.6), respectively.

Let us first investigate Advantage 1. In the languages FORTRAN or C, the objective function is usually rewritten as a sequence of calls to subroutines [4], [11], [12]. In FORTRAN, an arbitrary objective function, which is not defined by subroutines, can be placed as an argument to other functions only after a lexical and syntactical analysis of the entered expression. In the language C, it is allowed to pass functions as arguments to other functions using function pointers as arguments. But, even in this case, corresponding subroutines which define the objective function must be written by the user. In the programming package MATHEMATICA we represent an arbitrary objective function Q in the internal form which contains the following two parts:

- the first part, denoted by $q\_$, is an arbitrary arithmetic expression in MATHEMATICA;
- the second part, denoted by $var\_$, is the list of variables. Assume that $q\_$ is the parameter denoting the objective function Q and the parameter $var\_$ denotes the parameter list of Q. Let xm be the list representing a given point. Then the value $q[xm]$ can be computed by means of the following transformation rules:

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

Shortly, we can write

```
q0=q;  q0=q0/.var ->xm;
```

An arbitrary arithmetic expression, representing the objective function, can be written as the actual parameter, instead of the formal parameter $q\_$. Consequently, the program is able to uses objective functions known only at the run time.

This is a verification of Advantage 1.

The exploratory search is implemented in the function *ExpSearch*. It is assumed that an arbitrary objective function is given in the internal form $q\_$, $var\_$. More precisely, a new "basic point" is formed in the function *ExpSearch* and the new value of the objective function is compared with the old one.

Assume that $xm = \{xm[[1]],\ldots,xm[[n]]\}$ represents a given point. Then the function $P(\delta)$, defined in (2.7), can be implemented using the following sequence of transformation rules:

```
qm=q;
Do[qm=qm/.var[[j]]->xm[[j]], {j, i-1}];
qm=qm/.var[[i]]->(xm[[i]]-delta);
Do[qm=qm/.var[[j]]->xm[[j]], {j, i+1, n}];
```

The symbol *delta* in the second replacement is used instead of the parameter δ .

This is a verification of Advantage 2.

The function *ExpSearch* is written as follows:

```
ExpSearch [q_, var_List , t_List, delta_] : =
    Block [{q,qm,i,j,n=Length[var], success=False, xm=t},
        q0=qDo[q0=q0/.var[[j]]->xm[[j]],{j,n}];
        For[i=1,i<=n, i++,
            qm=q;
            Do[qm=qm/.var[[j]]->xm[[j]], {j,i-1}];
            qm=qm/.var[[i]]->(m[[i]]+delta);
            Do[qm=qm/.var[[j]]->xm[[j]],{j,i+1,n}];
            If [qm<q0,
                xm[[i]]+=delta;  success=True,
                    qm=q;
                    Do[qm=qm/.var[[j]]->xm[[j]], {j, i-1}];
                    qm=qm/.var[[i]]->(xm[[i]] -delta);
                    Do[qm=qm/.var[[j]]->xm[[j]],{j,i+1,n}];
                    If [qm<q0,
                        xm[[i]]-=delta;   success=True
                    ]
            ]
        ];
        {xm, success}
]
```

The original Hooke-Jeeves method (originated in [7]) can be implemented as follows.

```
HookeJeves [q_, var_List , xb_List, delta_, eps] : =
    Block [{n=Length [var], x0=xl=xb, t=xb , succ=False, q0, del=delta, pn, it=0},
        q0=q; Do[q0=q0/.var[[i]]->x0[[i]],{i,n}];
        While [Abs [del] >=eps,
            (* Type I exploratory search *)
            pn=ExpSearch[q,var,t,del];
            succ=pn[[2]];
            If[succ, xl=pn[[1]], del/=2];
            (* Pattern search *)
            t=xl+(xl-x0);
            (* Type II exploratory search *)
            pn=ExpSearch[q,var,t,del];
            succ=pn[[2]];
            If[succ, x0=xl; xl=pn[[1]], del/=2];
            q0=q; Do[q0=q0/.var[[i]]->xl[[i]] ,{i,n}];
            it+=1
        ];
        q0=q; Do[q0=q0/.var[[i]]->xl[[i]] ,{i,n}];
        {xl ,q0}
]
```

We will now describe the implementation of the first modification of the Hooke-Jeeves method and Advantage 3. The critical point in this implementation is the construction of the new function $F(h)$, depending on parameter h. According to (2.3), the function $F(h)$ is defined by

$$F(h) = Q(\mathbf{x}_B^{k+1} + h(\mathbf{x}_B^{k+1} - \mathbf{x}_B^k)) = Q(x1 + h(x1 - x0)) \tag{2.8}$$

A universal algorithm for automatic construction of the function $F(h)$ is not developed in procedural programming languages. Even if the function $Q(x)$ is defined by subroutines, it is difficult to generate the function $F(h)$ symbolically in procedural programming languages. If the function $Q(\mathbf{x})$ is given by a set of subroutines, then the corresponding functions which define the function $F(h)$ must also be written. This problem can be easily solved in MATHEMATICA using the following algebraic transformation

```
t=xl+h*(xl-x0);
```

and the following set of transformation rules:

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

Now, the unidimensional minimization $\min_h F(h)$ can be performed using the internal representation $f\_\{h\}$ of the objective function $F(h)$. In this transition of

parameters it is assumed that available functions for unidimensional optimization use the internal representation of the objective function as a formal parameter. This is another verification of Advantage 1.

The first modification of the Hooke-Jeeves method is implemented in the following function.

```
HookeJevesh [q_, var_List , xb_List , delta_, eps_] : =
Block [{n=Length[var], x0=xl=xb ,t=xb , succ=False ,q0, del=delta, pn, f, hk, it=0},
    q0=q; Do[q0=q0/.var[[i]]->x0[[i]], {i,n}];
    (* Form a new basic point *)
    While[del>=eps,
        (* Type I exploratory search *)
        pn=ExpSearch [q, var, t ,del];   succ=pn[[2]];
        If[succ, x1=pn[[1]], del/=2];
        (* Pattern search *)
        (* Form a new function F(h) symbolically *)
        t=x1+h*(x1-x0);
        f=q;
        Do[f=f/.var[[i]]->t[[i]], {i,n}];
        (* Unidimensional optimization hk = min F(h) *)
                                                  h
                    ... ...
        t=x1+hk*(xl-x0);
        (* Type II exploratory search *)
        pn=ExpSearch[q,var,t,del];
        succ=pn[[2]];
        If[succ, x0=xl; xl=pn[[1]], del/=2];
        q0=q; Do[q0=q0/.var[[i]]->x1[[i]], {i,n}];
        it+=1
    ];
    q0=q; Do[q0=q0/.var[[i]]->x1[[i]], {i,n}];
    {x1 ,q0}
]
```

In this way, we verify a part of Advantage 3.

Finally, the second modification of the Hooke-Jeeves method can be implemented symbolically as follows. The function *ExpDel*, performing an exploratory search, contains n unidimensional optimizations. The optimization directions $d_1,\ldots,d_n$ are defined such that each direction $d_i, 1 \le i \le n$ contains the ith coordinate identical to 1 and zeros in all other positions.

For an arbitrary $i \in \{1,\ldots,n\}$ the function

$$G(\lambda) = G(lam) = Q(y_i + \lambda d_i) = Q(xm + lam * d_i)$$

defined according to (2.4), can be effectively formed using algebraic manipulations with an arbitrary symbol *lam*, two sequences of transformation rules of the form

$$var[[j]] -> xm[[j]],\ j = 1,\ldots,i-1,\ var[[j]] -> xm[[j]],\ j = i+1,\ldots,n$$

and a transformation rule of the form

$$var[[i]] -> xm[[i]] + lam\ .$$

This is achieved in the following code:

```
qm=q;
Do[qm=qm/.var[[j]]->xm[[j]] ,{j,i-1}];
qm=qm/.var[[i]]->(xm[[i]]+lam);
Do[qm=qm/.var[[j]]->xm[[j]],{j, i+1, n}];
```

Now, the unidimensional optimization

$$\lambda_i = lambda = \min_{\lambda} G(\lambda) == \min_{lam} qm(lam)$$

can be performed using the internal form $qm.\{lam\}$ of the function $G(\lambda)$.

This is also a verification of Advantage 3.

```
ExpDel [q_,var_List ,t_List ,met_, eep_]: =
Block [{Lis={}, q0, qm, i, j, n=Length[var], eps=eep, lambda,
        xm=t, del=1, nor=0, metod=met},
q0=q; Do[q0=q0/.var[[j]]->xm[[j]],{j,n}];
Lis=Append[Lis,xm];
For [i=1, i<=n, i++,
    (* Form a new function G(lam) *)
    qm=q;
    Do[qm=qm/.var[[j]]->xm[[j]] ,{j,i-1}];
    qm=qm/.var[[i]]->(xm[[i]]+lam);
    Do[qm=qm/.var[[j]]->xm[[j]],{j,i+1,n}];
    (* Unidimensional optimization lambda = min G(lam) *)
                                          lam
        ... ...
        qm=qm/. lam->lambda;
        If [qm<q0, xm[[i]]+=lambda;
            nor+=lambda^2;
            Lis=Append [Lis, xm];
        ]
    ];
    del=Sqrt[nor];
    {xm,del,Lis}
]
```

We will now describe the implementation of the second modification of the Hooke-Jeeves method. According to (2.6), the function $H(h)$ is defined by

$$H(h) = Q(x_{k+1} + h(x_{k+1} - x_k)) \tag{2.9}$$

This function can be formed in an analogous way as the function $F(h)$, defined in (2.8). Assume that $x0$ and $x1$ denote two successive approximations x_k and x_{k+1}, respectively. Then the function $H(h)$ can be generated in the following way:

```
t=x1+h*(x1-x0);
qexp=q;
Do[qexp=qexp/.var[[i]]->t[[i]],{i,n}];
```

The second modification of the Hooke-Jeeves method is implemented in the function *HookeJeevesDel*.

```
HookeJevesDel [q_,var_List ,xb_List , epsi_]: =
    Block [{Lista={}, n1, n=Length[var], x0=x1=dd=xb, t=xb, succ=False, q0,
            delta=eps, pn, qexp, hk, it=0},
        Lista=Append[Lista,x0];
        Do[q0=q0/.var[[i]]→x0[[i]],i,n];
        pn=ExpDel[q,var,t,metod,eps];
        delta=pn[[2]];
        it+=1;x1=pn[[1]];
        Lista=Join[Lista,pn[[3]]];
        dd=x1-x0;
        While[Abs [delta] >=eps,
            (* Form a new function H(h) symbolically *)
            t=x1+h*(x1-x0);
            qexp=q;
            Do[qexp=qexp/.var[[i]]->t[[i]],{i,n}];
            t=x1+hk*(x1-x0);
            pn=ExpDel [q,var,t,metod,eps];
            delta=pn[[2]];
            Lista=Join[Lista,pn[[3]]];
            x0=x1; x1=pn[[1]]; Lista=Append[Lista,x1];
            dd=x1-x0;
            q0=q; Do[q0=q0/.var[[i]]→x1[[i]],{i,n}];
            it+=1;
];
q0=q; Do[q0=q0/.var[[i]]→x1[[i]] ,{i,n}];
{x1,q0,Lista}
]
```

4. NUMERICAL ILLUSTRATION

This section is devoted to a comparison of the above described modifications of the Hooke-Jeeves method. Graphical illustrations are developed using two plots by means of the functions *ListPlot* and *ContourPlot*, in a similar way as in [1] and [13].

Example 4.1. Consider the objective function $Q(x,y)=x^2+y^2-3\sin|x-y|$. The original Hooke-Jeeves method used in the expression

```
In[1]:= HookeJeves[x^2+y^2-3*Sin[x-y],{x,y},{0.5,0.8},0.1,0.001]
```

produces a divergent process:

x1={0.8, 0.8} q=1.28 delta=0.1
x1={1.1, 0.8} q=0.963439 delta=0.1
x1={1.3, 0.8} q=0.891723 delta=0.1
...
x1={0.9, $2.22045\ 10^{-16}$} q=-1.53998 delta=0.1
x1={0.9, -0.1} q=-1.70441 delta=0.1
...
x1={0.7, -0.4} q=-2.02362 delta=0.025
x1={0.575, -0.6} q=-2.07744 delta=0.025
...
x1={- 1.325, -3.95} q= 15.8764 delta=0.025
x1={-1.3, -3.925} q=15.6139 delta=0.025
...
x1={0.5, -0.725} q=-2.04679 delta=0.025
x1={0.625, -0.5} q=-2.06618 delta=0.025
...
x1={1.45, 1.225} q=11.8603 delta=0.025
x1={0.15, 3.5} q=11.6518 delta=0.025
...
x1={1.975, 13.65} q=187.889 delta=0.025
Out[1]= $Aborted

Under the same assumptions, the first modification of the Hooke-Jeeves method converges for an arbitrary unidimensional search optimization method. Consider the expression

```
In[2]:= HookeJevesh[x^2+y^2-3*Sin[x-y],{x,y},{0.5,0.8},0.1,0.001]
```

For example, in the starting point, symbolic function $f(h)$ is equal to

$$f = (0.8 + 0.\,h)^2 + (0.6 + 0.1\,h)^2 + 3\,\mathrm{Sin}[0.2 - 0.1\,h]$$

The golden section method after minimization $h_k = \min_h f(h)$ gives $h_k = 0.999959$ which implies

$$x1=\{0.799996,0.8\},\ q=1.28001,\ delta=0.1$$

In the 22th iteration we get

f = (-0.584927 - 5.03952 10^{-12} h)^2 + (0.585755 + 0.00312474 h)^2 -
> 3 Sin[1.17068 + 0.00312474 h]

hk = 0.0000408563 and

$x1$ = {0.585755, -0.584927}, q = -2.0778, $delta$ = 0.00078125.

Consider now the expression

In[3]:= HookeJevesh[x^2+y^2-3*Sin[x-y],{x,y},{0.5,0.8},0.1,0.00000000001]

Using the fixed step search during the unidimensional optimizations $h_k = \min_h F(h)$, we get the result after 64 iterations.

The obtained results are illustrated in the following picture.

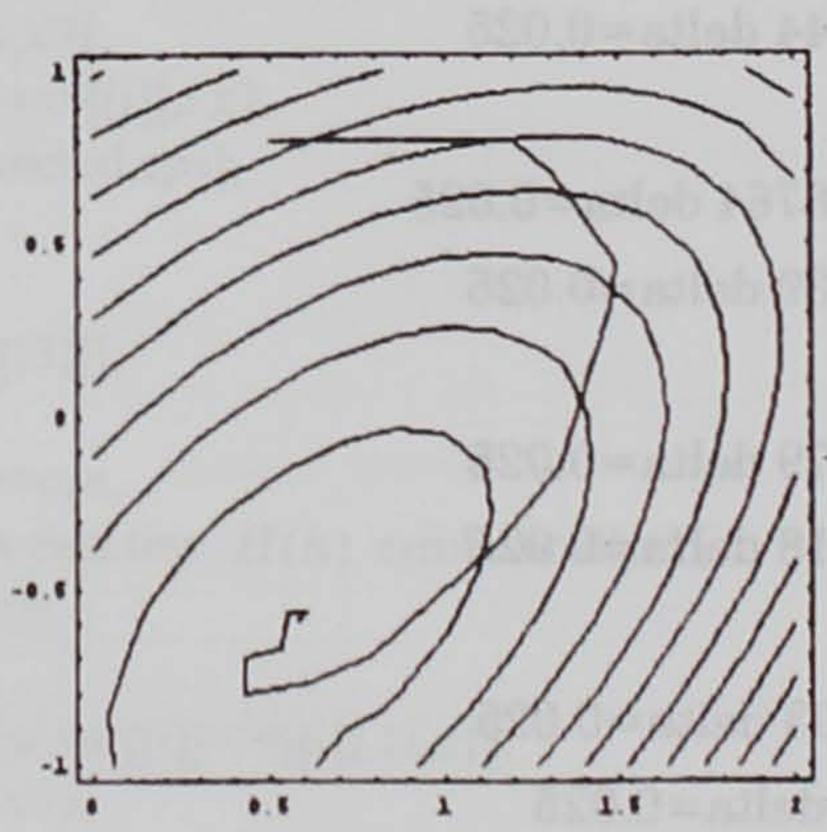

Figure 1: The first modification converges in 64 iterations

Under the same assumptions, the second modification of the method converges in 3 iterations.

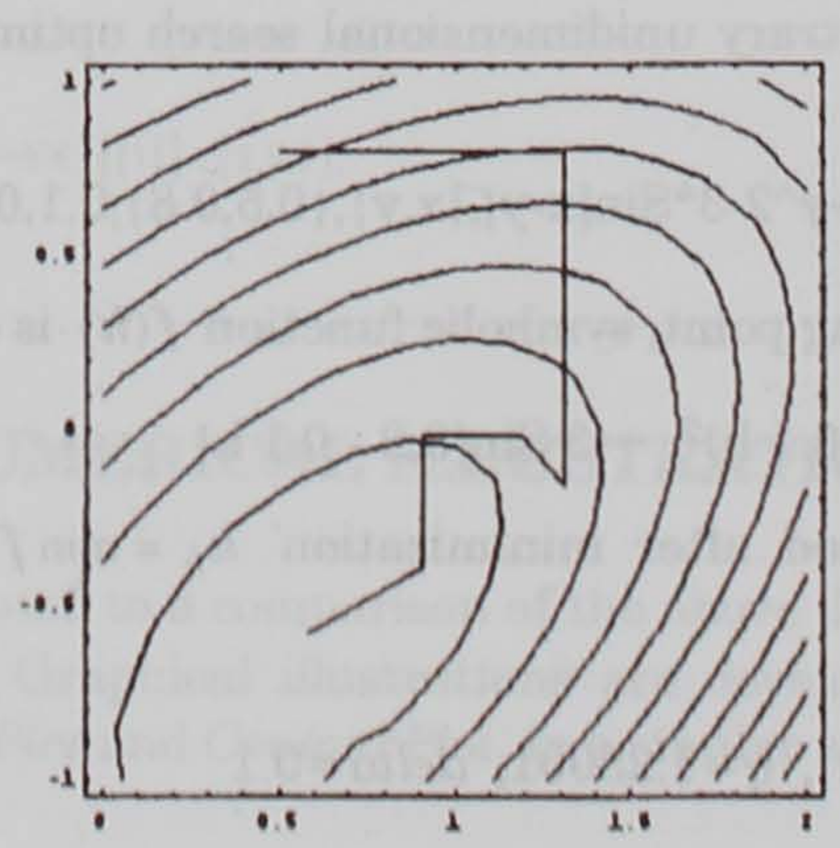

Figure 2: The second modification converges in 3 iterations

Example 4.2. The resulting list of the expression

HookeJevesh[x^2+y^2,{x,y},{2,3},0.1,0.00000000001]

is represented in the following figure:

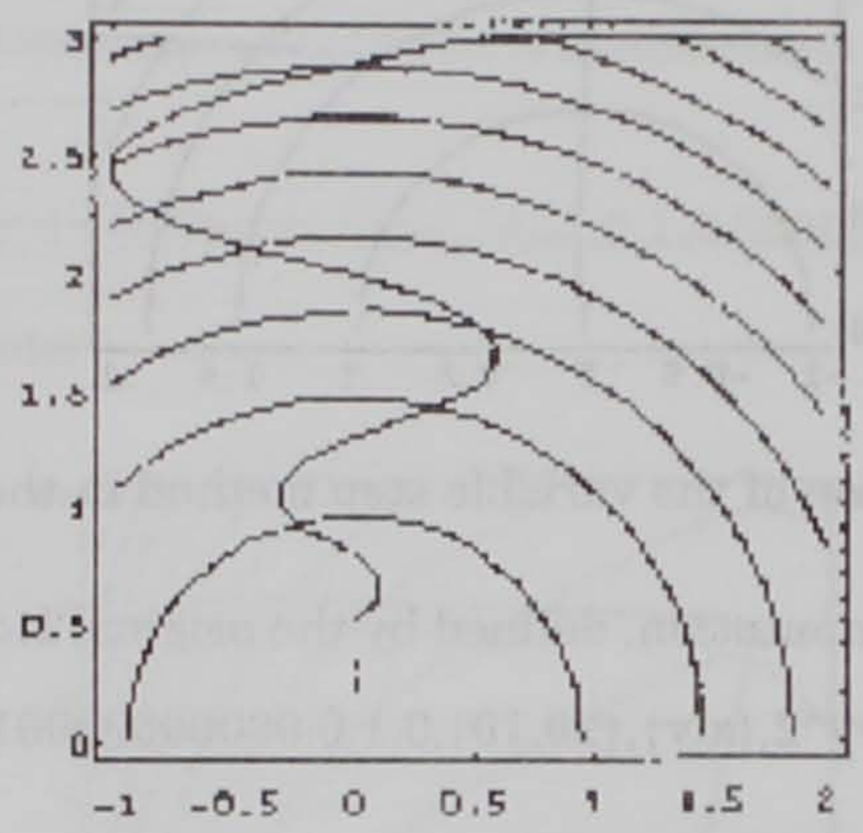

Figure 3: Convergence of the original Hooke-Jeeves method

The first modification of the modified Hooke-Jeeves method

HookeJevesh[x^2+y^2,{x,y},{2,3},0.1,0.00000000001]

improves convergence rate and accuracy. The following figure illustrates the trajectory generated by the fixed step method with the precision 10^{-11}:

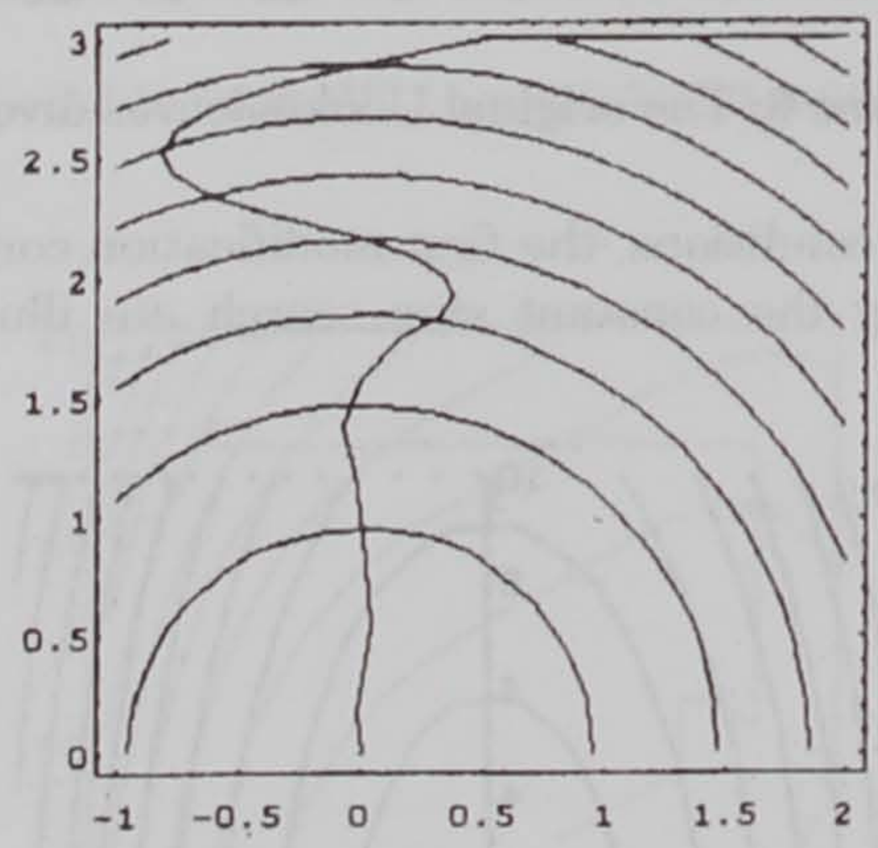

Figure 4: Application of the fixed step method in the first modification

The following figure illustrates the trajectory generated by the variable step method with the precision 10^{-11}:

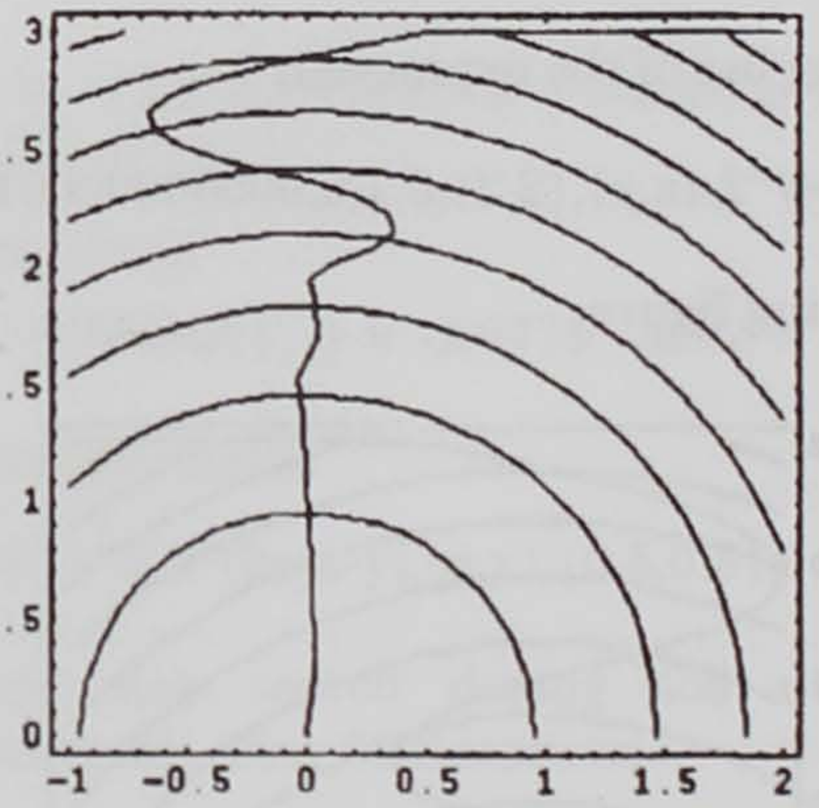

Figure 5: Application of the variable step method in the first modification

The following computation, defined by the original Hooke-Jeeves method

HookeJeves[x^2+y^2,{x,y},{20,10},0.1,0.00000000001]

diverges:

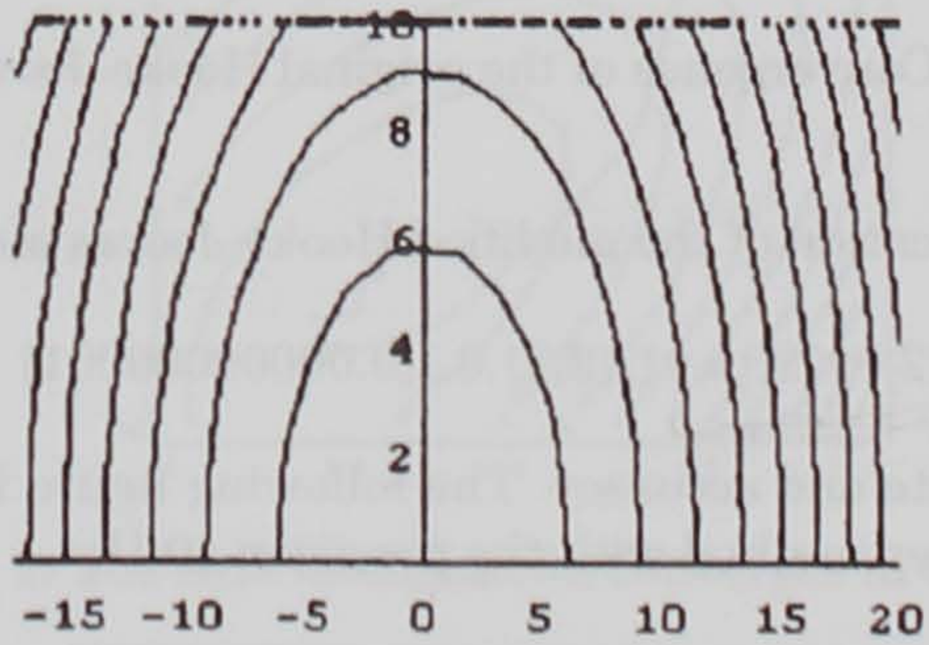

Figure 6: The original Hooke-Jeeves diverges

Under the same conditions, the first modification converges. For example, the results obtained applying the constant step search are illustrated in the following figure:

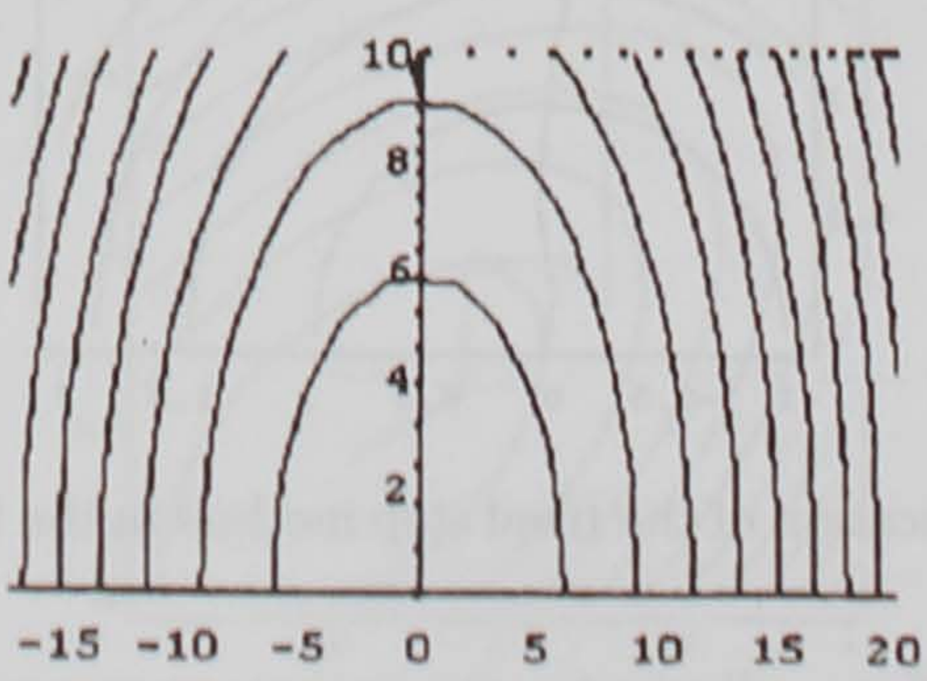

Figure 7: The first modification converges

Example 4.3. Consider the objective function $Q(x,y)=(x-2)^4+(x-2y)^2$. The original Hooke-Jeeves method, used in the expression

HookeJeves[(x-2)^4+(x-2y)^2],{x,y},{0,3},0.1,0.00000001]

diverges. Under the same assumptions, the first modification of the method, denoted by the expression

In[1]:=HookeJevesh[(x-2)^4+(x-2y)^2],{x,y},{0,3},0.1,0.00000001]

converges after 51 iterations:

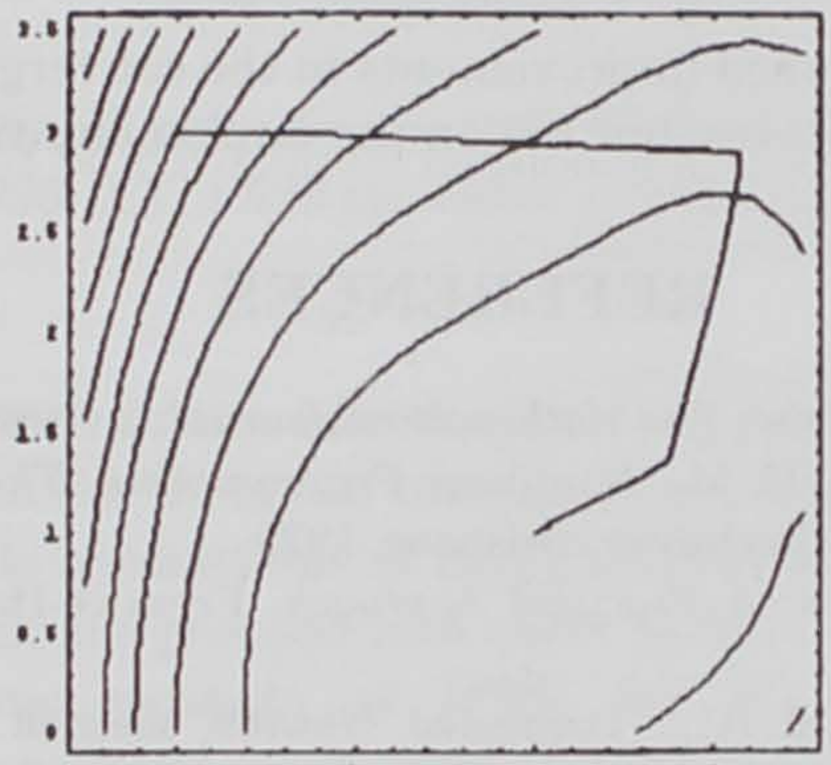

Figure 8: The first modification converges in 51 iterations

Finally, the second modification of the method, used in the expression

HookeJevesDel[(x-2)^4+(x-2y)^2],{x,y},{0,3},0.1,0.00000001]

converges in 4 iterations.

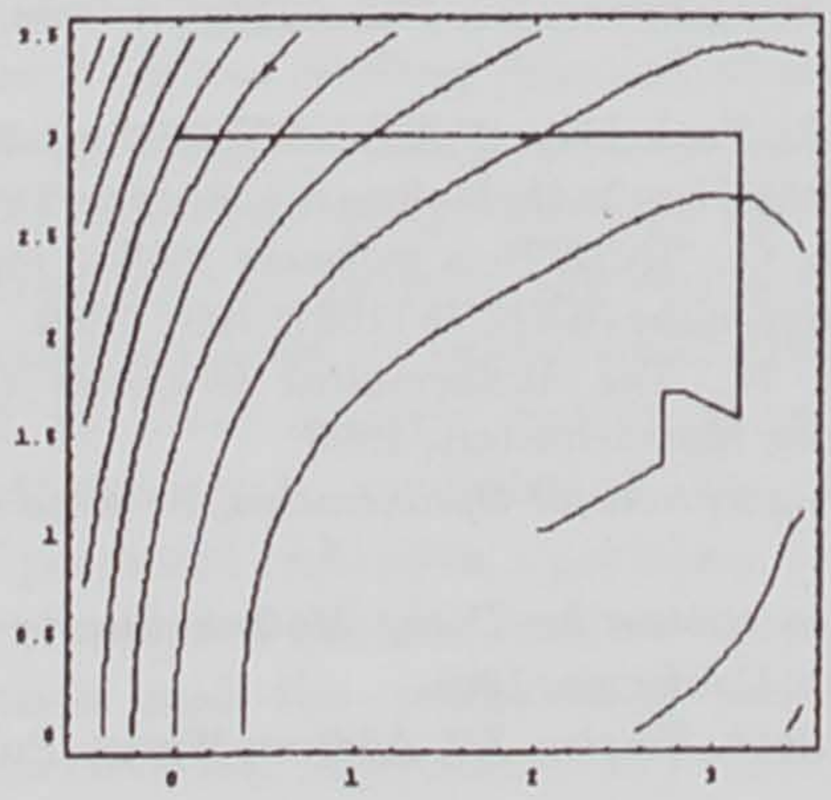

Figure 9: The second modification converges in 4 iterations

5. CONCLUSIONS

It is known that formula manipulation by a computer requires much more time and memory space than traditional implementation in the procedural programming languages. But, as shown in the examples, improvements in the convergence ensured by modifications of the Hooke-Jeeves method are significant. This is a compensation for the great memory space and time requirements for the above described symbolic implementation of these modifications. The second compensation for the symbolic implementation of the method is the simple implementation of algorithms for symbolic manipulations in the package MATHEMATICA. Moreover, the possibility of the software to process an arbitrary objective function makes it generally applicable. The user is released from further modifications in the program.

Note that the mentioned improvements in the convergence depend only on the modifications of the Hooke-Jeeves, but not on the implementation language.

REFERENCES

[1] Abbot, P., "Tricks of the trade", *The Mathematica Journal*, 3 (1993) 18-22.

[2] Bazaraa, M. S., and Shety, C. M., *Nonlinear Programming, Theory and Algorithms,* John Wiley and Sons, New York, Chichester, Brisbane, 1979.

[3] Blackman, N., *Mathematica: A Practical Approach*, Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[4] 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.

[5] Gray, T., and Glynn, J., *Exploring Mathematics in Mathematica*, Adisson-Wesley, Redwood City, California, 1991.

[6] Himellblau, D. M., *Applied Nonlinear Programming,* McGraw-Hill Book Company, 1972.

[7] Hooke, R., and Jeeves, T. A., "Direct search solution of numerical and statistical problems", *J. Assoc. Comput. Mash.*, 8 (1961) 212-229.

[8] Jacoby, S. L. S., Kowalik, J. S., and Pizzo, J. T., *Iterative Methods for Nonlinear Optimizotion Problems*, Prentice-Hall, Inc, Englewood, New Jersey, 1977.

[9] Krčevinac, S., Čupić, M., Petrić, J., and Nikolić, I., *Algorithms and Programs from Operations Research,* Naučna Knjiga, Beograd, 1983. (in Serbian)

[10] Maeder, R., *Programming in Mathematica, Third Edition*, Addison-Wesley, Redwood City, California, 1996.

[11] 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.

[12] 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.

[13] Smith, C., and Blackman, N., *The Mathematica Graphics Guidebook,* Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

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

[15] Wolfram, S., *Mathematica: a System for Doing Mathematics by Computer,* Addison-Wesley Publishing Co, Redwood City, California, 1991.

[16] Wolfram, S., *Mathematica Book, Version 3.0,* Addison-Wesley Publishing Co, Redwood City, California, 1997.

[17] Zlobec, S., and Petrić, J., *Nonlinear Programming*, Naučna knjiga, Beograd, 1989. (in Serbian)