Direct Optimizer
Technical Reference
DRAFT

Version: October 23, 2004.

http://www.directoptimizer.com

1. Introduction ____ *2*
2. Spreadsheet-program as an optimization platform ____ *3*
2.1. Built-in efficiency ____ **3**
2.2. Practical issues with Microsoft Excel ____ **7**
3. Hooke-Jeeves algorithm ____ *8*
3.1. The original algorithm ____ **8**
3.2. Suggested modifications ____ **10**
4. Berserk-mode ____ *11*
4.1. Description ____ **11**
4.2. The τ-parameter ____ **13**
5. Computational experiments ____ *14*
5.1. Extended Rosenbrock function (21) ____ **14**
5.2. Extended Powell Singular function (22) ____ **15**
5.3. Griewank's function ____ **16**
6. Discussion ____ *18*
6.1. Observed sublinearity in time complexity ____ **18**
6.2. Not a silver bullet ____ **18**
6.3. About nonlinear optimization in general ____ **19**
Appendix A: References ____ *21*
Appendix B: Installation Guide ____ *22*
Appendix C: Quick Start ____ *23*
Appendix D: Using Direct Optimizer from VBA programs ____ *24*
Accessing DOSolve via Application.Run ____ **24**
Accessing DOSolve directly ____ **25**
A simple multistart example ____ **26**
Known problems ____ **27**
Appendix E: Handling constraints ____ *28*

First version: September 1, 2004 released at http://www.directoptimizer.com as part of Direct Optimizer release 0.9.

1. Introduction

Direct Optimizer is a nonlinear optimization Add-In for Microsoft Excel, in many ways similar to the **Solver** Add-In shipped with Excel. They both optimize the contents of the target cell by varying other cells, which have been defined as variables for this purpose. The main differences to the Solver are:

1. Direct Optimizer can be (and has been) applied to problems even with million variables, but is often slow on small problems. Solver restricts the number of variables to 100, but is usually fast on small problems.

2. Direct Optimizer is based on modification of the Hooke-Jeeves direct search algorithm. Solver is based on dynamic quadratic models.

3. Direct Optimizer supports at the moment only simple bounds for the variables. Solver has sophisticated constraint support.

4. Direct Optimizer stores information about variable bounds to Data Validation of the variable cells. Solver has a proprietary method for storing constraint information.

5. Direct Optimizer has been designed for nonlinear optimization only. Solver can deal also with linear and integer problems.

As a convention, in the rest of this document we speak of “minimization” instead of “optimization". Our problem is to minimize a real-valued function $f(\mathbf{x})$ when $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in \mathbf{R}^n$.

2. Spreadsheet-program as an optimization platform

2.1. Built-in efficiency

The key idea behind a spreadsheet-program is to organize numerical data into a sheet of cells. The program keeps track of dependencies between cells. When some data is changed, the program computes automatically the cells, which contain formulas utilizing the recently changed data.

This approach leads to a highly intuitive user interface, which is now a "de facto" standard for various numerical tasks in a typical office. However, this approach allows also implementing many multivariate mathematical functions in such a way that when one variable is changed, calculating the effects of the change is often significantly less demanding than calculating the entire function from scratch.

As an example, consider the famous Extended Rosenbrock test function for n variables (n is even) [MGH81]:

$$f(\mathbf{x}) = \sum_{i=1}^{n} f_i^2(\mathbf{x})$$

where

$$\begin{aligned} f_{2i-1}(\mathbf{x}) &= 10(x_{2i} - x_{2i-1}^2) \\ f_{2i}(\mathbf{x}) &= 1 - x_{2i-1} \end{aligned}$$

A straightforward implementation of this test function with conventional C-programming language looks like this:

```
double ExtendedRosenbrock( const long n, const double* x)
{
      double result = 0;
      double temp1, temp2;
      long i;
      long uplimit = n-1;
      for (i = 0; i < uplimit; i += 2)
            result += 100*(temp1 *= (temp1 = (x[i+1] - x[i]*x[i])))
                  + (temp2 *= (temp2 = 1-x[i]));
      return result;
}
```

Now, every time we wish to explore the effect of changing one variable, as when estimating the gradient of the function with finite differences for example, we need to compute the entire function. This results in a lot of repeated computation.

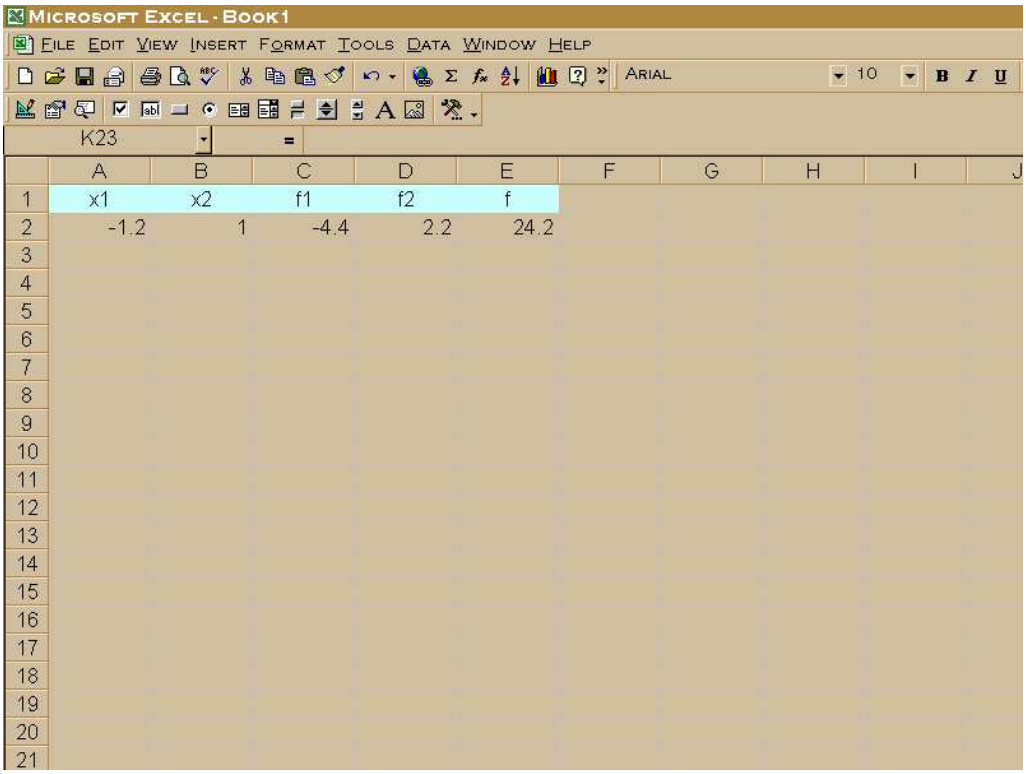
There are some possibilities to speed up the computation. The main alternatives are:

- Not estimating the effects of varying the variables with this function at all, but by implementing the gradient-function (or the "explorer"-function) separately.

- Keeping track of "computation history", and when requesting a new value, first checking what actually needs to be computed.

The first approach increases problem preparation effort, and is not always feasible. The second approach requires us first to compare the new variable values to the stored history - a check, which typically requires $O(n)$ time, and does not necessarily result in actual savings in computation time. We do not want to change the interface of the function for example by giving special parameters indicating the variables that have been changed (because this would not allow other, existing optimization algorithms to utilize the modified interface).

On the other hand, consider implementing the Extended Rosenbrock test function in a spreadsheet program. An experienced programmer might implement it in a way similar to the C-function approach above, but a non-programmer would most likely start with an approach like below, first implementing the function for 2 variables only:

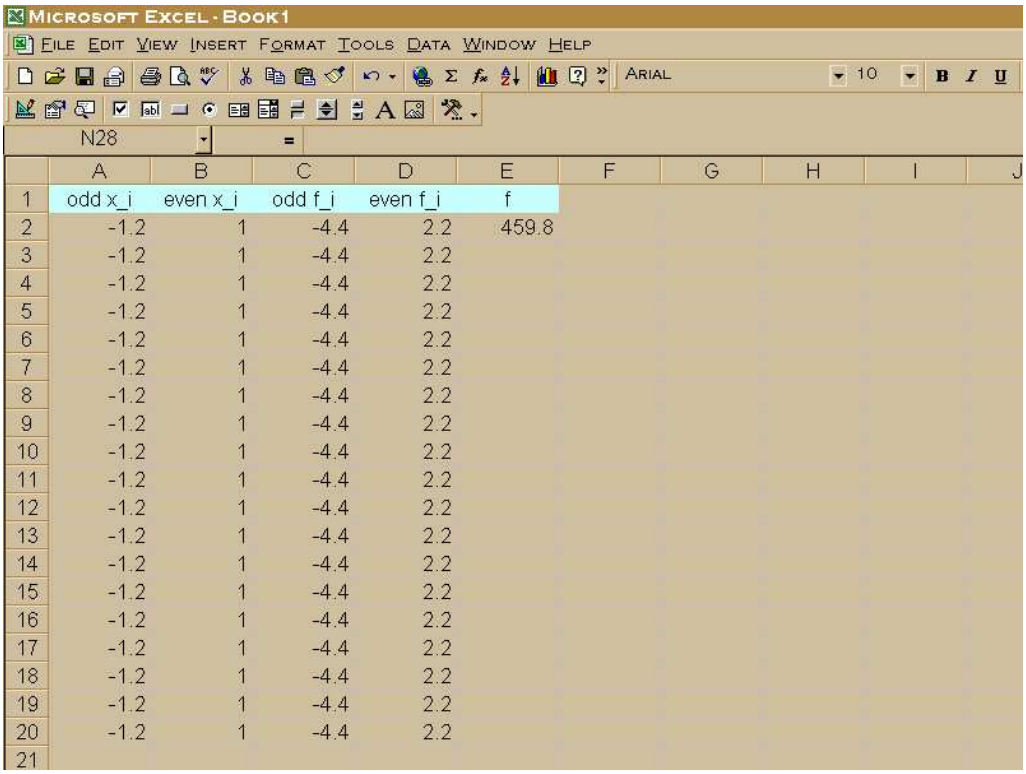


Here the first, coloured row contains only textual headlines for the columns. The contents of the other cells are:

A2: -1.2	(initial value for variable x_1)
B2: 1	(initial value for variable x_2)
C2: =10*(B2-A2*A2)	(formula for calculating f_1)
D2: =1-A2	(formula for calculating f_2)
E2: =C2*C2 + D2*D2	(formula for calculating f)

Updating this spreadsheet so that it handles more, say for example 38 variables instead of 2, is done like this:

- select cells A2:D2
- drag and drop the selection to row 20 with the computer's mouse
- update the formula in cell E2



	A	B	C	D	E	F	G	H	I	J
1	odd x_i	even x_i	odd f_i	even f_i	f					
2	-1.2	1	-4.4	2.2	459.8					
3	-1.2	1	-4.4	2.2						
4	-1.2	1	-4.4	2.2						
5	-1.2	1	-4.4	2.2						
6	-1.2	1	-4.4	2.2						
7	-1.2	1	-4.4	2.2						
8	-1.2	1	-4.4	2.2						
9	-1.2	1	-4.4	2.2						
10	-1.2	1	-4.4	2.2						
11	-1.2	1	-4.4	2.2						
12	-1.2	1	-4.4	2.2						
13	-1.2	1	-4.4	2.2						
14	-1.2	1	-4.4	2.2						
15	-1.2	1	-4.4	2.2						
16	-1.2	1	-4.4	2.2						
17	-1.2	1	-4.4	2.2						
18	-1.2	1	-4.4	2.2						
19	-1.2	1	-4.4	2.2						
20	-1.2	1	-4.4	2.2						
21										

The formula in cell E2 is updated to:

$$=\text{SUMSQ(C:C)+SUMSQ(D:D)}$$

so that it does not need to be changed if further changing the number of variables.

Now, consider what happens if we change the contents of the cell A2 in order to explore the effect of changing this variable. The spreadsheet program recalculates cells C2, D2 and E2, but values of cells C3:C20 and D3:D20 are used without re-computation.

In this simple example the amount of computation required when one variable changes is still $O(n)$, but by adding handling of partial sums it can be reduced as shown below. It is then assumed that internally in the spreadsheet-program, utilizing the dependencies between cells does not require a walk-through through all the cells. This is not true with all spreadsheet-programs in all situations.

Consider the case when n variables and computed data from them are organized into rows as follows:

$$\frac{n}{m}\text{ rows}:\begin{cases} x_1 & x_2 & \cdots & x_m & f_1(x_1, x_2, \cdots, x_m) \\ x_{m+1} & \cdots & & & f_2(x_{m+1}, x_{m+2}, \cdots, x_{2m}) \\ \vdots & & & & \vdots \\ x_{n-m+1} & x_{n-m+2} & \cdots & x_n & f_m(x_{n-m+1}, x_{n-m+2}, \cdots, x_n) \end{cases}$$

Each row has *m* variables and a function defined on those variables, the computation of which is assumed to take *O(m)* time. We wish to minimize the sum of the squares of the functions, but instead of calculating the sum of the squares directly, we calculate partial sums first, grouping the functions into groups of *k* functions each:

$$p_j = \sum_{i=j}^{j+k} f_i^2$$

and then computing the function to be minimized:

$$f = \sum_{j=1}^{\frac{n}{m}/k} p_j$$

Now, when one variable changes, we need to compute the function f_i for that row (*O(m)* time), one new partial sum (requiring *k* numbers to be squared and summed) and the new value for f (requiring $\frac{n}{m}/k$ numbers to be squared and summed).

Therefore the total number of operations required is

$$h(k) = am + b + 2k + \frac{2n}{mk}$$

with suitably selected constants *a* and *b*. If we wish to select *k* so that the number of these operations is minimized, we differentiate this formula with respect to *k* obtaining

$$h'(k) = 2 - \frac{2n}{mk^2}$$

thus finding out that $h(k)$ is minimized when $k = \sqrt{\frac{n}{m}}$ and $h(k)$ is then $O(\sqrt{n})$.

This can still be improved by arranging the data into a tree with more depth than above. That way it is possible to achieve a theoretical time complexity of $O(\log n)$ for examining the effect of varying one variable, but in practice with current spreadsheet-programs this might not allow greater problems to be solved (as discussed in the next chapter).

It must be emphasized that this technique exploits the structure of the function to be optimized in order to implement its computation in a more efficient way. Sometimes this is possible (perhaps more often so with artificial test functions), sometimes not. However, it imposes no changes to the interface between the function to be optimized and the optimization algorithm - the optimization routine still needs to know only the variables and the function value.

In addition, a spreadsheet program typically offers extensive support for graphics, multiple file formats, exotic mathematical functions - all appealing features for an optimization platform.

2.2. Practical issues with Microsoft Excel

In this discussion we consider only Microsoft Excel versions 97 and later.

Excel only tracks 65536 dependencies to unique references for automatic re-calculation. After the workbook has passed this limit, Excel no longer attempts to recalculate only the changed cells. It recalculates all the cells at each calculation instead. For more information on this limitation, see the discussion at http://www.decisionmodels.com/. This limit was apparently not reached with the tests described later in this document, even with one million variables on the Extended Rosenbrock test function.

3. Hooke-Jeeves algorithm

3.1. The original algorithm

The original Hooke-Jeeves algorithm has two phases, *pattern search* and *exploratory search.* First a relatively large step size δ is selected. Then *exploratory search* tries if the value of the function can be decreased by increasing or decreasing each variable by δ. If increasing a variable by δ has the desired effect, it is left to its new value and the effect of decreasing a variable is not examined anymore. It is indeed possible that *exploratory search* leaves all the variables unchanged.

Based on the current iteration $\mathbf{x}^i$ and previous iteration $\mathbf{x}^{i-1}$ *pattern search* establishes a "guess" $\mathbf{x}^i + (\mathbf{x}^i - \mathbf{x}^{i-1})$ even if the function value would increase there. After an exploratory search is performed from the point $\mathbf{x}^i + (\mathbf{x}^i - \mathbf{x}^{i-1})$, the value of the function in the resulted point is compared to $f(\mathbf{x}^i)$. If the value of the function in the new point is less than in $\mathbf{x}^i$, then it is accepted as a next iteration point $\mathbf{x}^{i+1}$ and the computation is continued with a new *pattern search*. Otherwise, an *exploratory search* is performed in the point $\mathbf{x}^i$. If this also fails in decreasing the function value, then the step size is decreased from δ to a new value δ / ρ, where $\rho > 1$ (typically $\rho = 2$) [HoJ61] [WiB67, p. 307-313] [Him72, p. 142-148].

The algorithm has been visualized in figure 1. The numbering of the points indicates the order in which the function evaluations are done. First an *exploratory search* is done, but all trials are abandoned. Then step size is reduced, leading next *exploratory search* to point 7, which is the second iteration point. From here a *pattern search* step is taken to point 8, and an exploratory search from there produces point 11 to be the third iteration point. Finally, at the point 18 function value is greater than in the previous iteration point 14, so *pattern search* step is cancelled, and *exploratory search* from point 14 produces point 20.

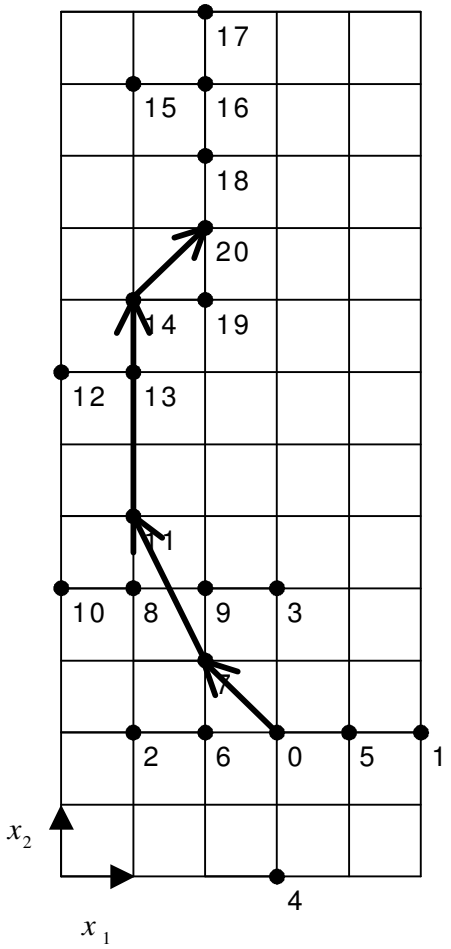


Figure 1: Example iterations 0-4 of the Hooke-Jeeves algorithm

More formally, *exploratory search* creates from a point $\mathbf{x}$ given to it as a parameter a new point $\mathbf{y}$ as:

$$y_j = \begin{cases} x_j + \delta, & \text{if } f(y_1,\ldots,y_{j-1},x_j+\delta,x_{j+1},\ldots,x_n) < f(y_1,\ldots,y_{j-1},x_j,x_{j+1},\ldots,x_n) \\ x_j - \delta, & \text{if } f(y_1,\ldots,y_{j-1},x_j+\delta,x_{j+1},\ldots,x_n) \geq f(y_1,\ldots,y_{j-1},x_j,x_{j+1},\ldots,x_n) \\ & \text{and } f(y_1,\ldots,y_{j-1},x_j-\delta,x_{j+1},\ldots,x_n) < f(y_1,\ldots,y_{j-1},x_j,x_{j+1},\ldots,x_n) \\ x_j, & \text{otherwise} \end{cases}$$

for each $j = 1,\ldots,n$. Let us assume that a function $\mathbf{E}(\mathbf{x})$ returns as its value the point, which is created from point $\mathbf{x}$ by applying *exploratory search*. Iterations of the algorithm are marked as $\mathbf{x}^i$, $i \geq 0$. We define two points, $\mathbf{e1}^i$ and $\mathbf{e2}^i$ for each iteration as:

$$\mathbf{e1}^i = \begin{cases} \mathbf{e2}^i & \text{when } i = 0 \\ \mathbf{E}(\mathbf{x}^i + (\mathbf{x}^i - \mathbf{x}^{i-1})) & \text{when } i > 0 \end{cases}$$

$$\mathbf{e2}^i = \begin{cases} \mathbf{E}(\mathbf{x}^i) & \text{if } f(\mathbf{E}(\mathbf{x}^i)) < f(\mathbf{x}^i) \\ \mathbf{e2}^i \text{ calculated after an assignment } \delta = \delta / \rho & \text{otherwise} \end{cases}$$

Note the recursive definition of $\mathbf{e2}^i$. The iterations of the Hooke-Jeeves algorithm are then:

$$\mathbf{x}^{i+1} = \begin{cases} \mathbf{e1}^i & \text{if } f(\mathbf{e1}^i) < f(\mathbf{x}^i) \\ \mathbf{e2}^i & \text{otherwise} \end{cases}$$

The algorithm is stopped when the step length δ becomes less than a pre-set value (typically square root of the machine epsilon) when reducing it. Point $\mathbf{e2}^i$ is not calculated on every iteration of the algorithm, but only when it is needed.

Some distinct features of the algorithm are:

- Only a step is taken to the search directions. No line searching is done.
- It is adaptive in the sense that the distance between successive iterations (when measured with infinity norm) may become even thousands of times larger than the step size used in the *exploratory search.*
- While gradient-based algorithms always explore the effect of varying the variables with an *infinitesimal* amount, Hooke-Jeeves algorithm explores the effect of varying the bits of a floating-point number in the direction *from most significant to least significant.*
- Because the algorithm takes the *pattern search* –step even when the value of the function increases, it is efficient in following a curved route towards optimum [Pow70, p.86]
- No multiplications or divisions between floating-point numbers are done (when δ is an integer), so it is easy to implement the algorithm in a machine-independent way with no danger of a floating-point overflow. The algorithm is numerically very stable.
- Its convergence has been proven [Céa71] [Yu79] [Tor97] [CoP01].

3.2. Suggested modifications

The *exploratory search* performed after the *pattern search* may produce the current iteration point, but because of rounding errors it may seem better than the current iteration point. This may even cause the algorithm to loop forever, but it can be avoided by verifying that the algorithm advances at least half the step size used in the *exploratory search* in every iteration [BeP66, p. 685]:

$$\mathbf{x}^{i+1} = \begin{cases} \mathbf{e1}^i & \text{if } f(\mathbf{e1}^i) < f(\mathbf{x}^i) \text{ and } \left\| \mathbf{e1}^i - \mathbf{x}^i \right\|_\infty > \delta / 2 \\ \mathbf{e2}^i & \text{otherwise} \end{cases}$$

and by adding a check into the computation of the vector $\mathbf{e2}^i$:

$$\mathbf{e2}^i = \begin{cases} \mathbf{E}(\mathbf{x}^i) & \text{if } f(\mathbf{E}(\mathbf{x}^i)) < f(\mathbf{x}^i) \text{ and } \left\| \mathbf{E}(\mathbf{x}^i) - \mathbf{x}^i \right\|_\infty > \delta / 2 \\ \mathbf{e2}^i \text{ calculated after an assignment } \delta = \delta / \rho & \text{otherwise} \end{cases}$$

After these modifications the algorithm is in practice immune to rounding errors.

A summary of other suggested modifications has been presented in [Kuh93]. In short, the key ideas have been

- Various methods for adjusting the step size δ (often individually for each variable).
- Adding "curvature estimation" by using more information from the previous iteration points.
- Adding randomness to the algorithm, for example by performing the exploratory search into random directions.
- Adding line minimizations to *exploratory search* or to *pattern search*.

No single modification has gotten very popular among practitioners. Typically the modifications are also supported with very few experimental results and without proofs of convergence.

Simple constant bounds for the variables can be incorporated into the algorithm by simply setting a variable to its bound whenever a bound is violated. However, this leads to a small problem when combined with the numerical stableness modification: when projecting a variable to the boundary, it might well cause the algorithm to progress less than the required amount of half the step size used (when measured with infinity norm), so this step would then be rejected. As a result, when the optimum occurs at the boundary, the algorithm may sometimes stop to a distance less than half of the final step size used from the boundary. This should not be a big problem in practice.

4. Berserk-mode

4.1. Description

In [Kuh93] the Hooke-Jeeves algorithm was applied to all (except one) such Moré-Garbow-Hillstrom test problems, which were generalized to n-dimensions. One notably difficult problem was the test function 25 *Variably dimensioned function*. No explanation could be given there why the problem was so difficult for the Hooke-Jeeves algorithm. However, when implementing Direct Optimizer in Microsoft Excel and watching the computation in action on computer screen led to a intuitive explanation: when optimizing this function, typically many variables remain unchanged for a long period of time. *Exploratory search* spends a lot of time examining the effect of varying variables, which have remained unchanged for even hundreds of iterations.

This same problem was later discovered even more dramatically when examining the behaviour of the Hooke-Jeeves algorithm in the extended version of the well-known Rosenbrock "banana valley" test function (Moré-Garbow-Hillstrom test function 21) using a non-conventional starting point (1, 1, 2, 4, 3, 9, 4, 16, ...). This seems to be pathologically difficult starting point for the original Hooke-Jeeves algorithm. The greater the initial values of the variable pairs are, the smaller the step size δ has to be in order to start changing the variables at all.

It would seem at first natural to increase the efficiency of the computation in cases like this by individual step size control for each variable. It would indeed keep more variables changing in early stages of the computation, but it would also add a lot of unnecessary computation to the late stages of the computation for example in the case of the extended Rosenbrock test function.

Therefore a new strategy was designed in this research based on the idea that if a variable has not changed for many iterations, the effect of changing it is not examined in the *exploratory search* which is done after the *pattern search*. If *pattern search* is then cancelled and *exploratory search* is done in the previous iteration point, we again ignore the "non-interesting" variables at first. Only if the function value is not decreased with this search we examine also the rest of the variables. Whenever the step size δ is reduced, the "change history" of each variable is reset.

More formally, let us assume a constant τ for indicating the number of iterations a variable must remain unchanged before we start ignoring it (the "temper"). Let also κ indicate the number of iterations that have passed since the last change of the step size δ.

We'll define a function $\mathbf{B}(\mathbf{x})$ to return a point $\mathbf{y}$ based on an iteration point $\mathbf{x}^i$ as follows:

$$y_j = \begin{cases} x_j^i & \text{if } \kappa \geq \tau \text{ and } x_j^i = x_j^{i-1} = \ldots = x_j^{i-\tau} \\ y_j \text{ as in the computation of } \mathbf{E} & \text{otherwise} \end{cases}$$

for each $j = 1, \ldots, n$. We also define a function $\tilde{\mathbf{B}}(\mathbf{x})$ which "explores the rest", returning a point $\mathbf{y}$ based on a point $\mathbf{x}$ (not necessarily an iteration point) as follows:

$$y_j = \begin{cases} y_j \text{ as in the computation of } \mathbf{E} & \text{if } \kappa \geq \tau \text{ and } x_j^i = x_j^{i-1} = \ldots = x_j^{i-\tau} \\ x_j & \text{otherwise} \end{cases}$$

The motivation for functions $\mathbf{B}(\mathbf{x})$ and $\tilde{\mathbf{B}}(\mathbf{x})$ is that in this way we can explore the variables in two phases: first by ignoring the variables, which have not been recently changing (by applying function $\mathbf{B}(\mathbf{x})$), and then by exploring the rest with $\tilde{\mathbf{B}}(\mathbf{B}(\mathbf{x}))$.

We now define points $\mathbf{b1}^i$ and $\mathbf{b2}^i$ for each iteration as:

$$\mathbf{b1}^i = \begin{cases} \mathbf{b2}^i & \text{when } i = 0 \\ \mathbf{B}(\mathbf{x}^i + (\mathbf{x}^i - \mathbf{x}^{i-1})) & \text{when } i > 0 \end{cases}$$

$$\mathbf{b2}^i = \begin{cases} \mathbf{B}(\mathbf{x}^i), & \text{if } f(\mathbf{B}(\mathbf{x}^i)) < f(\mathbf{x}^i) \\ \tilde{\mathbf{B}}(\mathbf{B}(\mathbf{x}^i)) & \text{if } f(\mathbf{B}(\mathbf{x}^i)) \geq f(\mathbf{x}^i) \text{ and } f(\tilde{\mathbf{B}}(\mathbf{B}(\mathbf{x}^i))) < f(\mathbf{x}^i) \\ \mathbf{e2}^i \text{ calculated after an assignment } \delta = \delta / \rho & \text{otherwise} \end{cases}$$

Note the usage of (recursive) definition of $\mathbf{e2}^i$ on the definition of $\mathbf{b2}^i$. The iterations of the Hooke-Jeeves algorithm utilizing Berserk-mode are then:

$$\mathbf{x}^{i+1} = \begin{cases} \mathbf{b1}^i & \text{if } f(\mathbf{b1}^i) < f(\mathbf{x}^i) \\ \mathbf{b2}^i & \text{otherwise} \end{cases}$$

Proof of convergence is not presented in this version of this paper.

It is not self-evident that this modification increases the performance of the algorithm, as seen in the next chapter.

4.2. The τ-parameter

In the tables below, the effect of value of τ is seen on various test functions. The value "infinity" refers to the original Hooke and Jeeves algorithm (no variables are ever ignored). These tests are run on a 266 MHz Pentium II machine with Direct Optimizer version 0.9, Windows 98 and Microsoft Excel 2000. Default parameter values for Direct Optimizer (other than τ) were used. The measured time is only approximative "calendar time", not the CPU-time used.

τ	time (hh:mm:ss)	iterations	func evals	func val at result
1	15:51:24	1752144	34164258	6.53E-11
2	11:42:16	1362429	25531894	3.50E-11
3	10:17:49	1193167	22450088	4.06E-11
4	5:07:46	682587	12130558	2.65E-11
5	3:29:39	481288	8196858	3.90E-11
6	1:16:00	205975	2919183	2.83E-11
7	6:34	10934	250464	3.00E-11
8	6:35	10934	251232	3.00E-11
9	6:35	10934	252000	3.00E-11
10	6:36	10934	252772	3.00E-11
20	6:47	10934	260464	3.00E-11
30	6:55	10934	268124	3.00E-11
40	7:03	10934	274948	3.00E-11
50	7:10	10934	279592	3.00E-11
60	7:14	10934	283672	3.00E-11
70	7:18	10934	286600	3.00E-11
80	7:22	10934	289020	3.00E-11
90	7:24	10934	291060	3.00E-11
100	7:26	10934	293100	3.00E-11
1000	11:02	10934	465824	3.00E-11
infinity	18:53	10934	847572	3.00E-11

Table 1: Extended Rosenbrock test function from starting point (1,1,2,4,3,9,4,16,....) for various τ when n=40

τ	time (mm:ss)	iterations	func evals	func val at result
1	5:41	13387	203277	3.10E-10
2	7:19	13801	271674	1.16E-10
3	8:21	14723	314307	3.02E-10
4	10:58	18811	414245	3.23E-10
5	10:13	16377	388082	1.78E-10
6	10:05	15734	384875	6.52E-11
7	11:09	16916	428882	2.85E-10
8	10:42	15548	408054	3.09E-10
9	12:04	17740	468346	1.73E-10

10	10:44	15320	418916	3.36E-10
20	13:45	16315	544686	2.85E-10
30	18:57	19721	761113	2.20E-10
40	17:43	16168	718266	3.80E-10
50	18:33	15348	740862	2.85E-10
60	19:59	14798	804224	1.73E-10
70	23:05	17411	931060	5.56E-11
80	20:30	13866	828238	1.09E-10
90	22:43	15238	922183	2.33E-10
100	24:08	16287	984053	2.08E-10
1000	30:19	15420	1263829	1.25E-10
infinity	29:35	15420	1263829	1.25E-10

Table 2: Variably Dimensioned test function for various τ when n=40

As these two test functions react to the value of τ in very different ways, we select a conservative default value of 100 for τ for the Direct Optimizer.

5. Computational experiments

The test functions are implemented in the file "testfunctions.xls" distributed with Direct Optimizer. Only three test functions have been chosen for presentation here. The main reasons for this limited experimentation are:

- Testing is very time-demanding and ongoing at the moment. This document is upgraded gradually.
- An overview how the Hooke-Jeeves algorithm behaves on those Moré-Garbow-Hillstrom test problems which have been generalized to n dimensions has already been given in [Kuh93].

5.1. Extended Rosenbrock function (21)

The function to be minimized is [MGH81]:

$$f(\mathbf{x}) = \sum_{i=1}^{n} f_i^2(\mathbf{x})$$

where

$$\begin{aligned} f_{2i-1}(\mathbf{x}) &= 10(x_{2i} - x_{2i-1}^2) \\ f_{2i}(\mathbf{x}) &= 1 - x_{2i-1} \end{aligned}$$

and the starting point is (-1.2, 1, -1.2, 1, ...). A typing error in [MGH81] in the definition of this function has been corrected.

The minimum is $f(\mathbf{x}) = 0$ at (1, ..., 1).

The results are:[1]

n	time	iters	f evals	f val	dist	dif
100	0:11	67	20048	1.85E-13	2.98E-08	1.85E-13
1000	1:58	67	197798	1.85E-12	2.98E-08	1.85E-12
10000	20:56	67	1975298	1.85E-11	2.98E-08	1.85E-11
100000	5:01:40	67	19750298	1.85E-10	2.98E-08	1.85E-10
1000000	170:16:32	67	197500298	1.85E-09	2.98E-08	1.85E-09

During the computation, Berserk-mode was allowed with the default temper-value of 100, but did not become active. The columns are:

n: number of variables
time: calendar time used in hours:minutes:seconds
iters: number of iterations of the algorithm
f evals: number of function evaluations
f val: function value at returned point
dist: distance of returned point from the minimum point as infinity-norm
dif: difference of returned function value to the minimum value

The number of function evaluations required (shown in the "f evals" column) depends on the number of variables as shown with the difference equation :

$$F(10n) = 10F(n) - 2682$$

The goodness of the solution when measured with the distance of the returned point from the minimum point as infinity norm remains the same.

5.2. Extended Powell Singular function (22)

The function to be minimized is [MGH81]:

$$f(\mathbf{x}) = \sum_{i=1}^{n} f_i^2(\mathbf{x})$$

where

[1] Direct Optimizer 1.0 on an Intel Celeron 2.4GHz PC with 512 MB of memory, Windows 98 SE and Excel 2000 SP-3.

$$
\begin{aligned}
f_{4i-3}(\mathbf{x}) &= x_{4i-3} + 10x_{4i-2} \\
f_{4i-2}(\mathbf{x}) &= \sqrt{5}(x_{4i-1} - x_{4i}) \\
f_{4i-1}(\mathbf{x}) &= (x_{4i-2} - 2x_{4i-1})^2 \\
f_{4i}(\mathbf{x}) &= \sqrt{10}(x_{4i-3} - x_{4i})^2
\end{aligned}
$$

and the starting point is $x_0 = (\xi_0, \ldots, \xi_n)$, where

$$
\begin{aligned}
\xi_{4j-3} &= 3 \\
\xi_{4j-2} &= -1 \\
\xi_{4j-1} &= 0 \\
\xi_{4j} &= 1
\end{aligned}
$$

The minimum is $f(\mathbf{x}) = 0$ at (0, ..., 0).

Initial step size for the algorithm was changed from the default value of 1 to 0.31. The results are:[2]

n	time	iters	f evals	f val	dist	dif
100	0:46	494	102114	4.22E-10	1.72E-03	4.22E-10
1000	07:40	494	1006614	4.22E-09	1.72E-03	4.22E-09
10000	1:24:18	494	10051614	4.22E-08	1.72E-03	4.22E-08
100000	22:05:50	494	100501614	4.22E-07	1.72E-03	4.22E-07

During the computation, Berserk-mode became active in every run. The columns are as with the Rosenbrock test function.

The number of function evaluations required (shown in the "f evals" column) depends on the number of variables as shown with the difference equation :

$$F(10n) = 10F(n) - 14526$$

The goodness of the solution when measured with the distance of the returned point from the minimum point as infinity norm remains the same.

5.3. Griewank's function

The function to be minimized (as presented in [ALR03]) is:

$$f(\mathbf{x}) = 1 + \sum_{i=1}^{n} \frac{x_i^2}{d} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right)$$

[2] Direct Optimizer 1.0 on an Intel Celeron 2.4GHz PC with 512 MB of memory, Windows 98 SE and Excel 2000 SP-3.

where $d = 10$. The starting point is chosen by selecting a random number uniformly distributed between -100 and 100 for each x_i. The global minimum is $f(\mathbf{x}) = 0$ at (0, ..., 0).

This function has several local minima, and is often used when testing global optimization software. The results are:[3]

n	time	iters	f evals	f val	dist	dif
100	0:26	35	19269	3.77E-15	4.04E-08	3.78E-15
1000	4:12	42	204087	4.75E-14	4.50E-08	4.75E-14
10000	1:09:38	80	2814244	4.80E-13	4.56E-08	4.80E-13
100000	86:07:30	191	50018481	4.83E-12	4.62E-08	4.83E-12

During the computation, Berserk-mode became active only in the run with 100000 variables. The columns are as with the Rosenbrock test function.

[3] Direct Optimizer 0.9c on an Intel Pentium II 266 MHz PC with 328 MB of memory, Windows 98 SE and Excel 2000 SP-3.

6. Discussion

6.1. Observed sublinearity in time complexity

It requires less function evaluations to solve the Extended Rosenbrock or the Extended Powell Singular test function with *10n* variables than solving ten times the same test function with *n* variables only.

This can be explained as follows. Let us assume that the algorithm has been implemented in such a way that *exploratory search* receives as its parameter the function value in the starting point. Therefore *pattern search* step computes the function value once. Let us further assume that when optimizing a function $f(\mathbf{x})$ from a starting point $\mathbf{x}^0 = (x_1^0, x_2^0, \ldots, x_n^0) \in \mathbf{R}^n$ we need $1 + p_f + e_f$ function value evaluations in total. Here p_f is the total number of function value evaluations done in *pattern search* steps, e_f is the total number of function value evaluations done in *exploratory searches,* and one evaluation is done in the beginning of the algorithm. Let us now construct a new function $F(\mathbf{z}), \mathbf{z} \in \mathbf{R}^{2n}$ as $F(\mathbf{z}) = f(\mathbf{x}) + f(\mathbf{y})$. Optimizing the function $F(\mathbf{z})$ from a starting point $\mathbf{z}^0 = (x_1^0, x_2^0, \ldots, x_n^0, x_1^0, x_2^0, \ldots, x_n^0) \in \mathbf{R}^{2n}$ requires $1 + p_f + 2e_f$ function value evaluations, which is always less than $2(1 + p_f + e_f)$ (even if p_f was zero). If rounding errors are absent, the distance of the returned point from the optimum when measured with $\| \|_\infty$ -norm remains the same.

When applying this analysis to the results of Extended Rosenbrock test function, it would seem that the optimization algorithm has performed 297 pattern search steps. This is counter-intuitive, as the algorithm has performed only 67 iterations. This can be explained as follows:

- Direct Optimizer counts every function evaluation, even if it would result only in returning the value of the target cell (without any calculation).
- Exploratory search in Direct Optimizer has been implemented so that it does not get the function value in the starting point as its parameter, therefore it evaluates it there.
- Also the result of the exploratory search is found out by evaluating the function in the main body of the algorithm.

This is insignificant from performance point of view, but the above analysis needs to be modified a bit when applied to the results of Direct Optimizer. As this is considered to be an uninteresting implementation detail, it is not presented here.

6.2. Not a silver bullet

The success of Direct Optimizer may seem almost phenomenal on selected problems, but even the Berserk-mode modification did not solve the difficulties it faces for example with the Variably dimensioned function (25) of More et al [MGH81]. At the time of writing this, Direct Optimizer has

spent 430 hours solving it on a 300 MHz Pentium II PC for one thousand variables applying Berserk-mode with the temper-value of 1, and is almost ready.

Although Berserk-mode helps on problems like this, further research is needed in order to understand why these problems are so difficult for it and if the algorithm can be modified to solve also these in an efficient way.

6.3. About nonlinear optimization in general

Nonlinear optimization has been dominated with the "dynamic quadratic models" paradigm. This, in turn, is based on the well-known results on how to optimize quadratic functions with no constraints on the variables. Several methods exist, which give exact solutions assuming the rounding errors have no effect. Some features of typical algorithms are summarized in the table below. We assume

- rounding errors have no effect on computation
- evaluating the gradient requires $O(n)$ function evaluations
- there are no constraints on the variables

Algorithm:	**Steepest descent**	**Fletcher-Reeves**	**Polak-Ribiere**	**Broyden-Fletcher-Goldfarb-Shanno**	**Newton**	**Levenberg-Marquardt**
Number of function evaluations for minimizing a quadratic function exactly:	unlimited	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
Time requirement for minimizing a quadratic function exactly:	unlimited	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^3)$	$O(n^3)$
Space requirement:	$O(n)$	$O(n)$	$O(n)$	$O(n^2)$	$O(n^2)$	$O(n^2)$
Does it always converge to a stationary point when applied to a twice continuously differentiable function, which is bounded from below?	yes	yes	no	?	no	yes
Type of the algorithm:	line searcher	line searcher, conjugate gradient	line searcher, conjugate gradient	line searcher, Quasi-Newton	both line search and no-search variations	trust region

In addition,

- General quadratic optimization with linear constraints on the variables is NP-hard [Vav91].

- If ε -approximate solutions are allowed, complexity of unconstrained quadratic optimization is not known. Research has focused to algorithms aiming for exact solutions. Yet often ε -approximate solutions are easier to find, as in linear programming for example.

Appendix A: References

[BeP66] Bell, M., Pike, M.: *Remark on Algorithm 178 [E4] Direct Search*. Communications of the ACM, Vol. 9, No. 9 (September 1966), 684-685.

[Céa71] Céa, J.: *Optimisation: théorie et algorithmes*. Dunod, 1971.

[CoP01] Coope, L., Price, C.: *On the Convergence of Grid-Based Methods for Unconstrained Optimization.* SIAM Journal on Optimization, Vol. 11, No. 4 (2001), p. 859-869.

[Him72] Himmelblau, D.: *Applied Nonlinear Programming*. McGraw-Hill, 1972.

[HoJ61] Hooke, R., Jeeves, T.A.: *Direct Search Solution of Numerical and Statistical Problems*. Journal of the ACM, Vol. 8, No. 2 (April 1961), 212-229.

[Kuh93] Kuha, J.:*Nonlinear optimization based on comparison of function values* (in finnish). Master's thesis, University of Helsinki, Department of Computer Science, 1993.

[ALR03] Appel, M., Labarre, R., Radulovic, D.: *On Accelerated Random Search*. SIAM Journal on Optimization, Vol. 14, Number 3 (March 2003), p. 708-731.

[MGH81] Moré, J., Garbow, B., Hillstrom, K.: *Testing Unconstrained Optimization Software*. ACM Transactions on Mathematical Software, Vol. 7, No. 1 (March 1981), 17-41.

[Pow70] Powell, M.: *A Survey of Numerical Methods for Unconstrained Optimization*. SIAM Review, Vol. 12, No. 1 (January 1970), 79-97.

[Tor97] Torczon, V.: *On the Convergence of Pattern Search Algorithms*. SIAM Journal on Optimization, Vol. 7, No. 1 (February 1997), p. 1-25.

[Vav91] Vavasis, S.: *Nonlinear Optimization - Complexity Issues*. Oxford University Press, 1991.

[WiB67] Wilde, B., Beightler, C.: *Foundations of Optimization*. 6th p., Prentice-Hall, 1967.

[Yu79] Yu, W.: *Positive basis and a class of direct search techniques* (in chinese). Scientia Sinica, Special Issue on Mathematics, 1 (1979), p. 53-67.

Appendix B: Installation Guide

In order to install Direct Optimizer:

1. Copy the files directoptimizer.xla and directoptimizer.hlp to a directory of your choice.
2. Add the Add-In to Excel's list of Add-Ins. You may need to refer to Excel's Help-system. For Excel 97, the procedure is:
   - Choose Tools, Add-Ins to display the Add-Ins dialog box
   - Click Browse and navigate to your copy of directoptimizer.xla
   - Click Open
   Note: In some versions of Windows, you will be asked if you want to copy the Add-In to the "Library". Do not do so, as the help file will not be copied with the Add-In.
3. A license-window opens. Read the license carefully. If you feel you can accept the license, click the button "I accept this license". Otherwise, click the button "I don't accept this license" and stop the installation.
4. The Direct Optimizer Add-In should now appear in the list of Add-Ins. Make sure the check box next to its entry is selected. A new "Direct Optimizer" menu entry to the Tools-menu has now been created. Choose it in order to activate Direct Optimizer.

If you are using Excel XP and your Macro security is set to "High", then the Add-In won't load and you won't get a warning. Go to Tools menu, Macro submenu, and click "Security" to change it to "Medium".

Appendix C: Quick Start

After installing Direct Optimizer as described in Appendix B, open the file testfunctions.xls distributed with Direct Optimizer. Select the "Rosenbrock" sheet. Click the button "Setup problem". Enter for example 100 as the problem size, and "1" as the starting point (-1.2, 1,-1.2, 1, ...).

You have now defined a problem with 100 variables to be solved. Start Direct Optimizer from the Tools-menu. Click "Target Cell:". Click the red cell (D1) with your mouse, and then click "OK" in the message box. Click "Guess" in the Direct Optimizer main window. Click "Solve". You will see a "Direct Optimizer - computing" window with a progress indicator. Wait for a while, until the result of computation window opens. Click OK.

Congratulations, you have solved your first problem with Direct Optimizer!

Appendix D: Using Direct Optimizer from VBA programs

Assuming that Direct Optimizer has been installed (as instructed in Appendix B), you can use Direct Optimizer from your own programs and macros. A function called DOSolve is offered for this purpose. It has been declared within the Direct Optimizer Add-In as:

```
Public Function DOSolve( _
    TargetCell As String, _
    MaxMinVal As Integer, _
    ValueOf As Double, _
    VariableCells As String, _
    Unconstrained As Boolean _
    ) As Boolean
'
' Parameters:
' TargetCell - cell to be optimized or solved
' MaxMinVal - problem type: 1=Max, 2=Min, 3=Val
' ValueOf: if MaxMinVal=3, then this gives the target value
' VariableCells: if <> "", then the variables, otherwise precedents are used
' Unconstrained: handle the problem as unconstrained or not
'
' this function returns FALSE if computation resulted in error,
' TRUE otherwise
```

DOSolve assumes, that:

- It is OK to project variables violating the bounds to the feasible region.
- It is OK to ignore non-valid bounds.
- It is OK to activate the worksheet containing the target cell.

Other settings than those given as parameter may be adjusted by starting Direct Optimizer interactively on the sheet in question, then modifying the desired options (such as the initial step size of the algorithm), and then pressing either Exit or Retain on the main screen. This causes these sheet-specific settings to be saved, and DOSolve reads them when it starts. DOSolve does not modify these settings by itself. The parameters given to DOSolve override the corresponding sheet-specific settings.

There are two main ways to access the procedure DOSolve: via Application.Run and referencing it directly. Both ways are described below. You can find these examples also from the file testfunctions.xls - open the Visual Basic Editor (in Excel 2000: Tools-Macro-Visual Basic Editor), and then see the module DO_VBA. Run the examples as you would run any other macro in Excel.

Note: it is a common practice to learn Excel programming with its Macro Recorder. You can learn how to call the Solver Add-In from your programs with it, but the same does not apply to Direct Optimizer. You need to follow the instructions below instead.

Accessing DOSolve via Application.Run

The procedure below uses DOSolve to minimize the Extended Rosenbrock test function found in the file testfunctions.xls (distributed with Direct Optimizer):

```
Sub DirectOptimizerBatchExample_1()
```

```
    ' in this example we are going to
    ' optimize the extended Rosenbrock test
    ' function from our program

    ' first we activate the sheet of interest:
    Application.Workbooks("testfunctions.xls").Worksheets("Rosenbrock").Activate

    ' then we optimize
    Dim success As Boolean
    success = Application.Run("directoptimizer.xla!DOSolve", "D1", _
                              2, 0, "", True)

End Sub
```

Here the target cell has been given ("D1"), the problem is a minimization problem (2), the value of ValueOf parameter is insignificant, the variables have been specified with an empty string so that they will be guessed (to those precedents of the target cell which do not contain formulas), and finally the problem is considered as unconstrained.

The advantage of this approach is that you only need to install the Direct Optimizer Add-In in order to run your program - less work, if distributing your programs to other persons. The disadvantage of this approach is that the parameters have to be given without specifying the name of the parameter, thus requiring special attention that no mistakes are made.

Accessing DOSolve directly

The procedure below refers to DOSolve directly in order to minimize the Extended Rosenbrock test function. Before this program can be run, you need to open the Visual Basic Editor (in Excel 2000: Tools-Macro-Visual Basic Editor), then access the Tools-References menu and add a check-mark next to DirectOptimizer.

```
Sub DirectOptimizerBatchExample_2()

    ' in this example we are also going to
    ' optimize the extended Rosenbrock test
    ' function from our program
    '
    ' NOTE: THIS EXAMPLE REQUIRES THAT YOU HAVE
    ' USED THE TOOLS-REFERENCES COMMAND IN THE
    ' VISUAL BASIC EDITOR TO ENABLE REFERENCES
    ' TO THE DIRECT OPTIMIZER ADD-IN!

    ' first we activate the sheet of interest:
    Application.Workbooks("testfunctions.xls").Worksheets("Rosenbrock").Activate

    ' then we optimize
    Dim success As Boolean
    success = DOSolve( _
        TargetCell:="D1", _
        MaxMinVal:=3, _
        ValueOf:=10, _
        VariableCells:="", _
        Unconstrained:=False)

End Sub
```

Here the target cell has been given ("D1"), the problem requires this time solving (we want a value of 10 to the target cell), the variables have been specified with an empty string so that they will be guessed, and finally the bounds for the variables, if given in the Data Validation of the variable cells, will be taken into account.

A simple multistart example

Below is an example how to write a program which calls Direct Optimizer from randomized starting points. This example optimizes the Extended Rosenbrock test function one thousand times from different starting points and gives an error message, if the optimization run does not give a satisfactory result or results in an error. You can find also this example from the included file testfunctions.xls, module DO_VBA.

```
Sub DirectOptimizerStressTest()

    ' first we activate the sheet of interest:
    Application.Workbooks("testfunctions.xls").Worksheets("Rosenbrock").Activate

    MsgBox "This macro performs a stress test on Extended " & vbCrLf & _
        "Rosenbrock test function with 20 variables." & vbCrLf & _
        "Set it up so first!"

    ' initialize random number generator
    Randomize

    Const upcount As Long = 1000
    Dim counter As Long
    counter = 0

    While (counter < upcount)
        counter = counter + 1
        Dim c As Range
        Dim a As Range

        Set a = Application.Range("G1:Z1")

        For Each c In a
            'randomize contents
            c.Value = 40 * Rnd - 20
        Next c

        ' then we optimize
        Dim success As Boolean

        success = Application.Run("directoptimizer.xla!DOSolve", "D1", _
                                  2, 0, "", False)
        ' target is D1, we minimize , valueof is dummy,
        ' guess variables, obey possible constraints

        If (success = False) Then
           MsgBox "Optimization run failed!"
           Exit Sub
        ElseIf (Application.Range("D1").Value > 0.00000001) Then
           MsgBox "Problem in stress test with the returned value!"
           Exit Sub
        End If
```

```
    Wend

    MsgBox "Stress test OK!"

End Sub
```

Known problems

Problem:

If you change the constant "upcount" in the DirectOptimizerStressTest from 1000 to 10000 and run it on Win98/Excel 2000 SP-3, the following symptoms occur:

- When you open in Windows Start-Programs-Accessories-System Tools-Resource Meter and follow the "GDI resources" during the stress test, the given percentage gradually decreases (this may take hours to become evident).
- The command buttons "Setup problem" and "Evaluate solution" will finally not show.
- Before the test is complete, Excel and/or Windows becomes non-stable and may crash.

Work-around:

Run DOSolve in a loop only on a sheet not containing Command Button controls (such as the "Setup problem"-button on the Extended Rosenbrock test function sheet). If you want to use such buttons, locate them on a different sheet than the model to be optimized and link them to your model. Just remember to keep the entire model (both the variables and the target cell) on a single sheet.

This way you can call DOSolve in a loop and the GDI resources remain the same.

Discussion:

This is most likely caused by a memory leak in the Command Button control. It is possible that other controls contain similar problems. This problem is less evident in Windows 2000/XP than in Windows 95/98/Me, as they handle the GDI resources differently, but problems could occur also there.

At the time of writing this, it is not known if this problem has been corrected in later versions of Excel.

This problem is relevant only when calling DOSolve hundreds of times. With "casual usage" it does not matter.

Appendix E: Handling constraints

Direct Optimizer has built-in support for simple box-like constraints only - that is, for each variable x_i you can define constants a_i and b_i so that

$$a_i \le x_i \le b_i$$

Often this is not enough. You would like to have something more complicated, for example

$$x_1 + 2x_2 + x_3^2 - 10 \ge 0$$

for variables x_1, x_2 and x_3.

One straightforward method for problems like these is to apply either penalty- or barrier-techniques. Any standard textbook on nonlinear optimization should describe these and their usage.

As a concrete example, if your Excel-sheet contains in the cell C1 your complicated constraint so that it should remain non-negative, you can define to cell P1 a "penalty term" like this:

=MIN(C1, 0)^2

You do need to square the penalty term. An attempt to use absolute value may cause the target function to be non-differentiable, and typically results in premature convergence to a non-optimal point.[4]

Then you can define to cell M1 a "penalty multiplier" for example like this:

=100

Let us say the cell T1 is your target cell to be minimized. Then the penalty should increase the value of your target cell when violating the bound. The new target cell T2 would look like this:

=T1+M1*P1

The value of the cell M1 has great effect on computation. The larger it is, the more accurate the solution is, but the more difficult the optimization is. You should fine-tune its value yourself via a couple of trials.

Penalty functions are only one of the many possible techniques. In the ideal case the user should not need to consider details like these. If in doubt, Solver has sophisticated constraint support and it should be utilized when applicable. Possibly in the future Direct Optimizer will offer more advanced built-in constraint support.

[4] Note also that we assume that Windows has been configured to use comma as a list separator symbol. For example finnish regional settings introduce semicolon as such by default, and this example will result in a parsing error on Excel in such computers.