Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computer Physics Communications 174 (2006) 152–159
Computer Physics Communications
www.elsevier.com/locate/cpc

Genetically controlled random search: a global optimization method for continuous multidimensional functions ☆

Ioannis G. Tsoulos, Isaac E. Lagaris *

Department of Computer Science, University of Ioannina, P.O. Box 1186, Ioannina 45110, Greece

Received 24 July 2005

Available online 27 October 2005

Abstract

A new stochastic method for locating the global minimum of a multidimensional function inside a rectangular hyperbox is presented. A sampling technique is employed that makes use of the procedure known as grammatical evolution. The method can be considered as a "genetic" modification of the Controlled Random Search procedure due to Price. The user may code the objective function either in C++ or in Fortran 77. We offer a comparison of the new method with others of similar structure, by presenting results of computational experiments on a set of test functions.

Program summary

Title of program: GenPrice
Catalogue identifier: ADWP
Program summary URL: http://cpc.cs.qub.ac.uk/summaries/ADWP
Program available from: CPC Program Library, Queen's University of Belfast, N. Ireland
Computer for which the program is designed and others on which it has been tested: the tool is designed to be portable in all systems running the GNU C++ compiler
Installation: University of Ioannina, Greece
Programming language used: GNU-C++, GNU-C, GNU Fortran-77
Memory required to execute with typical data: 200 KB
No. of bits in a word: 32
No. of processors used: 1
Has the code been vectorized or parallelized?: no
No. of lines in distributed program, including test data, etc.: 13 135
No. of bytes in distributed program, including test data, etc.: 78 512
Distribution format: tar. gz
Nature of physical problem: A multitude of problems in science and engineering are often reduced to minimizing a function of many variables. There are instances that a local optimum does not correspond to the desired physical solution and hence the search for a better solution is required. Local optimization techniques are frequently trapped in local minima. Global optimization is hence the appropriate tool. For example, solving a nonlinear system of equations via optimization, employing a "least squares" type of objective, one may encounter many local minima that do not correspond to solutions, i.e. minima with values far from zero.
Method of solution: Grammatical Evolution is used to accelerate the process of finding the global minimum of a multidimensional, multimodal function, in the framework of the original "Controlled Random Search" algorithm.
Typical running time: Depending on the objective function.

☆ This paper and its associated computer program are available via the Computer Physics Communications homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Corresponding author.
E-mail address: lagaris@cs.uoi.gr (I.E. Lagaris).

0010-4655/$ – see front matter
doi:10.1016/j.cpc.2005.09.007

PACS: 02.60.-x; 02.60.Pn; 07.05.Kf; 02.70.Lq; 07.05.Mh

Keywords: Global optimization; Stochastic methods; Genetic programming; Grammatical evolution

1. Introduction

A recurring problem in many applications is that of finding the global minimum of a function. This problem may be stated as: Determine

$$x^* = \arg\min_{x \in S} f(x).$$

The nonempty set $S \subset R^n$ considered here, is a hyper box defined as:

$$S = [a_1, b_1] \otimes [a_2, b_2] \cdots \otimes [a_n, b_n]$$

Recently several methods have been proposed for the solution of the global optimization problem. These methods can be divided in two main categories, deterministic and stochastic. Random search methods are widely used in the field of global optimization, because they are easy to implement and also since they do not depend critically on a priori information about the objective function. Various random search methods have been proposed, such as the Random Line Search [1], Adaptive Random Search [2], Competitive Evolution [3], Controlled Random Search [4], Simulated Annealing [5–8], Genetic Algorithms [9,10], Differential Evolution [11,12], methods based on Tabu Search [23], etc. This article introduces a new sampling technique for use with conjunction with Controlled Random Search. The method is based on the genetic programming procedure known as Grammatical Evolution. Performance comparison to other methods is quite favorable as might be verified by inspecting the reported results of our computational experiments in Table 1, Section 3.2. The suggested approach uses a population of randomly created moves, that guide the underlying stochastic search towards the global minimum. These random moves are produced by applying the method of grammatical evolution. Grammatical evolution is an evolutionary process that can produce code in an arbitrary language. The production is performed using a mapping process governed by a grammar expressed in Backus Naur Form. Grammatical evolution has been applied successfully to problems such as symbolic regression [14], discovery of trigonometric identities [15], robot control [16], caching algorithms [17], financial prediction [18], etc. The rest of this article is organized as follows: in Section 2 we give a brief presentation of the grammatical evolution and of the suggested algorithms. In Section 3 we list some experimental results from the application of the proposed method and a comparison is made against traditional global optimization methods and in Section 4 we present the installation and the execution procedures of the GenPrice.

2. Description of the algorithm

2.1. Grammatical evolution

Grammatical evolution is an evolutionary algorithm that can produce code in any programming language. The algorithm requires the grammar of the target language in BNF syntax and the proper fitness function. Chromosomes in grammatical evolution, in contrast to classical genetic programming [20], are not expressed as parse trees, but as vectors of integers. Each integer denotes a production rule from the BNF grammar. The algorithm starts from the start symbol of the grammar and gradually creates the program string, by replacing non-terminal symbols with the right hand of the selected production rule. The selection is performed in two steps:

- Read an element from the chromosome (with value V).
- Select the rule according to the scheme

$$\text{Rule} = V \bmod \mathcal{R}, \tag{1}$$

where $\mathcal{R}$ is the number of rules for the specific non-terminal symbol.

The process of replacing non-terminal symbols with the right hand of production rules is continued until either a full program has been generated or the end of chromosome has been reached. In the latter case we can reject the entire chromosome or we can start over (wrapping event) from the first element of the chromosome. In our approach we allow at most two wrapping events to occur. If the limit of two wrapping events is reached the chromosome is rejected. The rejection of a chromosome means that a large fitness value is assigned to the chromosome and as a consequence it will not be used in the crossover procedure. Further details about the grammatical evolution procedure can be found in [13,14,19].

2.2. Used grammar

The grammar of the package is a small portion of the grammar of the C programming language. The grammar can be expressed as follows in BNF notation:

```
<START>::=<expr>
<expr>:=(<expr><binary_op><expr>)
       |<func_op>(<expr>)
       |<terminal>
<binary_op>::=+|-|{*}|/
<func_op>::=sin | cos | exp | log
<terminal>::=<digitlist>.<digitlist>
             |x
<digitlist>::=<digit>
             |<digit><digit>
             |<digit><digit><digit>
<digit>::=0|1|2|3|4|5|6|7|8|9
```

The symbol named START denotes the starting symbol of the grammar. As we can see the employed programming language supports four functions and at most three digit numbers. Note that it is straightforward to extend the function repertoire and upgrade the support to multiple digit numbers.

2.3. Description of the Genetic Random Search (GRS) algorithm

The algorithm named GRS creates random trails that can be embedded in any stochastic procedure to guide the search towards the global minimum. This algorithm is essential for the method Genetically Controlled Random Search method introduced in this article and its main steps are the following:

INPUT data:

- A point $x = (x_1, x_2, \ldots, x_n),\ x \in S \subset R^n$.
- ϵ, a small positive number. Typical values for this parameter are 10^{-4} to 10^{-5}.
- k, a small positive integer, usually between 10 and 20.
- **Set** selection rate. The value of this parameter denotes the fraction of chromosomes that will pass unchanged to the next generation, and therefore the fraction of new chromosomes which will be created through the process of crossover. Typical values for this parameter is 0.1, 0.2, etc.
- **Set** mutation rate. The value of this parameter controls the average number of changed in a chromosome. Typical values for this parameter is 0.02, 0.05, etc.

INITIALIZATION step:

- The initialization of each element of the genetic population is performed by selecting a random integer in the range $[0, 255]$.

LOOP step:

- **For** $i = 1, \ldots, k$ **Do**
  - **Set** $x_{\text{old}} = x$.
  - **Create** a new generation of chromosomes in the population with the use of the genetic operations (crossover, mutation, reproduction). At first the chromosomes are sorted in a way such that the best of them is placed at the beginning of the population and the worst at the end. After that, $c = (1 - s) \times g$ new chromosomes are created through the process of crossover. The parameter s denotes the value of selection rate and the parameter g denotes the total number of chromosomes in the genetic population. The new chromosomes will replace the worst in the population at the end of the crossover procedure. For every couple of children two chromosomes are selected from the population with the method of tournament selection i.e.: First a group of $K \geqslant 2$ randomly selected individuals is created. The individual with the best fitness value is selected for mating while the rest are discarded. Having selected two chromosomes, two new are created by the process of one-point crossover. In that procedure the chromosomes are cut at a randomly chosen point and the right-hand-side subchromosomes are exchanged, as shown in Fig. 1. Crossover does not respect the boundaries between the different parts of the chromosomes. After the crossover, mutation is applied to every chromosome in the population; for every chromosome in the population and for every element in the chromosome a random number in the range $[0, 1]$ is chosen. If this number is less than or equal to the mutation rate, the corresponding element is changed randomly, otherwise it remains intact.
  - **For** every chromosome **Do**
    - **Split** the chromosome uniformly into n pieces, one for each dimension. Each piece corresponds to a random movement and is denoted by $p_i,\ i = 1, \ldots, n$. On every piece p_i the grammatical evolution transformation is applied, which is based on the proposed grammar. This determines a univariate function f_i.
    - **Denote** by d the vector $(d_1 = f_1(x_1), d_2 = f_2(x_2), \ldots, d_n = f_n(x_n))$.
    - **Set** $x_+ = x + d$.
    - **If** $x_+ \notin S$ or $f(x_+) > y$ **then**
      - **Set** $x_- = x - d$.
      - **If** $x_- \notin S$ or $f(x_-) > y$ **then**
        Set the fitness value to a very large number.
      - **Else**
        Set the fitness value to $f(x_-)$.
      - **Endif**
    - **Else**
      - **Set** the fitness value to $f(x_+)$.
    - **Endif**
  - **Endfor**
  - **Set** $x = x + d_{\text{best}}$, where d_{best} the movement that corresponds to the chromosome with the best fitness value.
  - **If** $|x - x_{\text{old}}| \leqslant \epsilon$, terminate and **return** x as the located minimizer.
- **Endfor**
- **Return** x as the located minimizer.

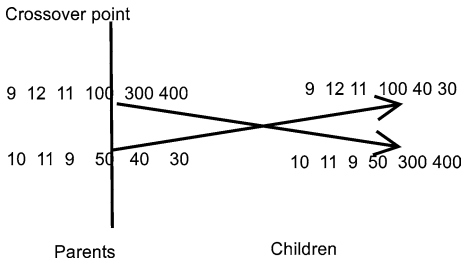

Fig. 1. One-point crossover.

2.4. Genetically Controlled Random Search (GCRS)

The Controlled Random Search is a population based optimization algorithm and it has been applied successfully to many problems [24] and is the base of our new procedure described below:

Initialization step:

- **Set** the value for the parameter N. A commonly used value for that is $N = 25n$.
- **Set** a small positive value for ϵ.
- **Create** the set $T = \{z_1, z_2, \ldots, z_N\}$, by randomly sampling N points from S.

Min_Max step:

- **Select** the points $z_{\min} \in T$ and $z_{\max} \in T$, that yield the minimum and maximum f-values correspondingly. Set

$$f_{\min} = f(z_{\min}) \quad \text{and} \quad f_{\max} = f(z_{\max}). \tag{2}$$

- **If** $|f_{\max} - f_{\min}| < \epsilon$, **then goto Local_Search** step.

New_Point step:

- **Select** randomly the reduced set $\tilde{T} = \{z_{T_1}, z_{T_2}, \ldots, z_{T_{n+1}}\}$ from T.
- **Compute** the centroid G:

$$G = \frac{1}{n} \sum_{i=1}^{n} z_{T_i}.$$

- **Compute** a trial point $\tilde{z} = 2G - z_{T_{n+1}}$.
- **If** $\tilde{z} \notin S$ **or** $f(\tilde{z}) \geqslant f_{\max}$ **then** goto New_Point step.
- **Perform** a call to GRS procedure using as starting point the point $\tilde{z}$. This is the step that distinguishes the new method from the controlled random search [4].

Update step:

- $T = T \cup \{\tilde{z}\} - \{z_{\max}\}$.
- **Goto Min_Max** step.

Local_Search step:

- $z^* = \text{localSearch}(z)$, where localSearch is a deterministic local search procedure such as BFGS, DFP, etc. The local search procedure used in the GenPrice tool is the BFGS variant due to Powell [21].
- Return the point z^* as the discovered global minimum.

The **Local_Search** step is applied only at the end of the algorithm to ensure that the algorithm will find a true local minimum and not just an approximation of it.

3. Experimental results

The Genetically Controlled Random Search (GCRS) was tested against

1. The original Controlled Random Search (CRS).
2. The modified Controlled Random Search (PCRS) as described in [26].

We list also results from the Simulated Annealing (SA) as modified by Goffe et al. [8] not for immediate comparison since the methods are quite different, but only as a reference point (their code `simann.f` is available from the URL: http://www.netlib.org).

The comparison is made using a suite of well-known test problems. Each method was run 30 times on every problem using different random seeds. We have measured the ability of the method to discover the global minimum and the number of function evaluations it required. In all cases the selection rate was set to 90% and the mutation rate to 5%. The length of each chromosome was set to $10 \times d$, where d is the dimensionality of the objective function and the maximum number of iterations allowed in the GRS method (parameter K) was set to 10. We used the suggested (Ref. [4]) value of $N = 25n$, for the initial population in the methods CRS, PCRS and GCRS. Similarly we employed the parameters suggested in the documentation of the Simulated Annealing software, available from the URL http://www.netlib.org, namely: $N_S = 20$, $N_T = 5$, $T = 5.0$, $a = 0.5$, TLAST $= 4$ for SA. All the experiments were conducted on an AMD ATHLON 2400+ equipped with 256 MB RAM. The hosting operating system was Debian Linux and the used programming language was the GNU C++. The trial steps produced by the grammatical evolution were evaluated using the FunctionParser programming library [22].

3.1. Test functions

Camel

$f(x) = 4x_1^2 - 2.1x_1^4 + \frac{1}{3}x_1^6 + x_1x_2 - 4x_2^2 + 4x_2^4$, $x \in [-5, 5]^2$ with 6 local minima and global minimum $f^* = -1.031628453$.

Rastrigin

$f(x) = x_1^2 + x_2^2 - \cos(18x_1) - \cos(18x_2)$, $x \in [-1, 1]^2$ with 49 local minima and global minimum $f^* = -2.0$.

Griewank2

$f(x) = 1 + \frac{1}{200}\sum_{i=1}^{2} x_i^2 - \prod_{i=1}^{2} \frac{\cos(x_i)}{\sqrt{(i)}}$, $x \in [-100, 100]^2$ with 529 local minima and global minimum $f^* = 0.0$.

Gkls

$f(x) = \text{Gkls}(x, n, w)$, is a function with w local minima, described in [25], $x \in [-1, 1]^n$, $n \in [2, 100]$. In our experiments we use $n = 2, 3$ and $w = 50$.

GoldStein & Price

$$\begin{aligned} f(x) = {} & \left[1 + (x_1 + x_2 + 1)^2 \right. \\ & \times \left. (19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2)\right] \\ & \times \left[30 + (2x_1 - 3x_2)^2 \right. \\ & \times \left. (18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)\right]. \end{aligned}$$

The function has 4 local minima in the range $[-2, 2]^2$ and global minimum $f^* = 3.0$.

Test2N

$$f(x) = \frac{1}{2}\sum_{i=1}^{n} x_i^4 - 16x_i^2 + 5x_i$$

with $x \in [-5, 5]^n$. The function has 2^n local minima in the specified range. In our experiments we used the cases of $n = 4, 5, 6, 7$.

Test30N

$$f(x) = \frac{1}{10}\sin^2(3\pi x_1) \times \sum_{i=2}^{n-1}\left((x_i-1)^2\left(1+\sin^2(3\pi x_{i+1})\right)\right) + (x_n-1)^2\left(1+\sin^2(2\pi x_n)\right)$$

with $x \in [-10, 10]^n$. The function has 30^n local minima in the specified range. In our experiments we used the cases of $n = 3, 4$.

Potential

The molecular conformation corresponding to the global minimum of the energy of N atoms interacting via the Lennard–Jones potential is determined for two cases: with $N = 3$ atoms and with $N = 5$ atoms. We refer to the first case as **Potential(3)** (a problem with 9 variables) and to the second as **Potential(5)** (a problem with 15 variables). The global minimum for the first cases is $f^* = 3$ and $f^* = -9.103852416$.

Neural

A neural network (sigmoidal perceptron) with 10 hidden nodes (30 variables) was used for the approximation of the function $g(x) = x\sin(x^2)$, $x \in [-2, 2]$. The global minimum of the training error is $f^* = 0.0$.

3.2. Results

In Table 1 we list the results for the Simulated Annealing in the column labeled SA, the Controlled Random Search in the column labeled CRS, the modified Controlled Random Search in the column denoted by PCRS and the results from the proposed Genetically Controlled Random Search in the column denoted by GCRS. The numbers in the cells represent the average number of function evaluations required by each method. The figures in parentheses denote the fraction of runs that located the global minimum and were not trapped in one of the local minima. Absence of this number denotes that the global minimum has been recovered in every single run. The proposed GCRS has shown superior performance among its peers. This can be deduced from the significantly lower number of the required function evaluations, and the fraction of runs that succeeded in finding the global minimum. The new ingredient in the algorithm is the GRS procedure, which is based on Grammatical Evolution. Hence, the observed performance enhancement is due to this new component. It remains to be seen in practice if the performance advantage observed for the present benchmark suite, will be maintained in other real world problems as well.

Table 1
Experimental results obtained by the methods of SA, CRS, PCRS and GCRS applied on several global optimization benchmarks

Function	SA	CRS	PCRS	GCRS
Camel	4820	1852	1409	1504
Rastrigin	4843	1903	1982	428
Griewank2	4832(0.27)	2105	2004	977
Gkls(2, 50)	4820	1627	1495	1220
Gkls(3, 50)	7228	3349	3059	2056
Goldstein	4842	1923	1456	961
Test2N(4)	9631	6835(0.97)	4831	4280(0.97)
Test2N(5)	12034(0.87)	25270(0.97)	12342	7958
Test2N(6)	14438(0.66)	32801(0.70)	8840(0.87)	9914
Test2N(7)	16840(0.37)	38057(0.40)	11751(0.63)	9740
Test30N(3)	7930(0.23)	3703	2124	1519
Test30N(4)	9858(0.23)	5135	4058	1416
Potential(3)	21404	198046	34985	9265
Potential(5)	36212	188646	39305	9096
Neural	76667(0.93)	122617	94016	14559

4. Software documentation

4.1. Distribution

The package is distributed in a tar.gz file named `GenPrice.tar.gz` and under UNIX systems the user must issue the following commands to extract the associated files:

1. gunzip `GenPrice.tar.gz`.
2. tar xfv `GenPrice.tar`.

These steps create a directory named `GenPrice` with the following contents:

1. **bin**: A directory which is initially empty. After compilation of the package, it will contain the executable **make_genprice**.
2. **examples**: A directory that contains the test functions used in this article, written in ANSI C++ and the Fortran 77 version of the Six Hump Camel function.
3. **include**: A directory which contains the header files for all the classes of the package.
4. **src**: A directory containing the source files of the package.
5. **Makefile**: The input file to the `make` utility in order to build the tool. Usually, the user does not need to change this file.
6. **Makefile.inc**: The file that contains some configuration parameters, such as the name of the C++ compiler, etc. The user must edit and change this file before installation.

4.2. Installation

The following steps are required in order to build the tool:

1. Uncompress the tool as described in the previous section.
2. `cd GenPrice`.
3. Edit the file `Makefile.inc` and change (if needed) the five configuration parameters.
4. Type `make`.

The five parameters in `Makefile.inc` are the following:

1. **CXX**: It is the most important parameter. It specifies the name of the C++ compiler. In most systems running the GNU C++ compiler this parameter must be set to g++.

2. **CC**: If the user written programs are in C, set this parameter to the name of the C compiler. Usually, for the GNU compiler suite, this parameter is set to gcc.
3. **F77**: If the user written programs are in Fortran 77, set this parameter to the name of the Fortran 77 compiler. For the GNU compiler suite a usual value for this parameter is g77.
4. **F77FLAGS**: The compiler GNU FORTRAN 77 (g77) appends an underscore to the name of all subroutines and functions after the compilation of a Fortran source file. In order to prevent this from happening we can pass some flags to the compiler. Normally, this parameter must be set to -fno-underscoring.
5. **ROOTDIR**: Is the location of the GenPrice directory. It is critical for the system that this parameter is set correctly. In most systems, it is the only parameter which must be changed.

4.3. User written subprograms

The user can write his objective function either in C, C++ or in Fortran 77 in a single file. Each file has a series of functions in an arbitrary order. However, the C++ files must have the lines

```
extern ''C'' {
```

before the functions and the line

```
}
```

after them. The meaning of the functions are the following:

1. **getdimension**(): It is an integer function which returns the dimension of the objective function.
2. **getleftmargin**(left): It is a subroutine (or a void function in C) which fills the double precision array left with the left margins of the objective function.
3. **getrightmargin**(right): It is a subroutine (or a void function in C) which fills the double precision array right with the right margins of the objective function.
4. **funmin**(x): It is a double precision function which returns the value of the objective function evaluated at point x.
5. **granal**(x, g): It is a subroutine (or a void function in C) which returns in a double precision array g the gradient of the objective function at point x.

4.4. The utility make_genprice

After the compilation of the package, the executable `make_genprice` will be placed in the subdirectory `bin` in the distribution directory. This program creates the final executable and it takes the following command line parameters:

1. `-h`: Prints a help screen and terminates.
2. `-p filename`: The `filename` parameter specifies the name of the file containing the objective function. The utility checks the suffix of the file and it uses the appropriate compiler. If this suffix is .cc or .c++ or .CC or .cpp, then it invokes the C++ compiler. If the suffix is .f or .F or .for then it invokes the Fortran 77 compiler. Finally, if the suffix is .c it invokes the C compiler.
3. `-o filename`: The `filename` parameter specifies the name of the final executable. The default value for this parameter is GenPrice.

4.5. The utility GenPrice

The final executable `GenPrice` has the following command line parameters:

1. `-h`: The program prints a help and it terminates.
2. `-c count`: The integer parameter `count` specifies the number of chromosomes for the Genetic Random Search procedure. The default value for this parameter is 20.
3. `-s srate`: The double parameter `srate` specifies the selection rate used in the Genetic Random Search procedure. The default value for this parameter is 0.10 (10%).
4. `-m mrate`: The double parameter `mrate` specifies the mutation rate used in the Genetic Random Search procedure. The default value for this parameter is 0.05 (5%).
5. `-r seed`: The integer parameter `seed` specifies the seed for the random number generator. It can assume any integer value.
6. `-o filename`: The parameter `filename` specifies the file where the output from the `GenPrice` will be placed. The default value for this parameter is the standard output.

4.6. A working example

Consider the Six Hump Camel function given by

$$f(x) = 4x_1^2 - 2.1x_1^4 + \frac{1}{3}x_1^6 + x_1x_2 - 4x_2^2 + 4x_2^4,$$

$$x \in [-5, 5]^2$$

with 6 local minima. The implementation of this function in C++ and in Fortran 77 is shown in Figs. 2 and 3. Let the file with the C++ code be named `camel.cc` and that with the Fortran code `camel.f`. Let these files be located in the `examples` subdirectory. Change to the `examples` subdirectory and create the `GenPrice` executable with the `make_genprice` command:

```
../bin/make_genprice -p camel.cc
```

or for the Fortran 77 version

```
../bin/make_genprice -p  camel.f
```

The `make_genprice` responds:

```
RUN./GenPrice IN ORDER TO RUN THE PROBLEM
```

Run `GenPrice` by issuing the command:

```
./GenPrice  -c 10 -r 1
```

The resulting output appears as:

```
FUNCTION EVALUATIONS = 1310
GRADIENT EVALUATIONS = 20
MINIMUM =  0.089842  -0.712656  -1.031628
```

```
extern "C"{
 int getdimension()
 {
   return 2;
 }

 void getleftmargin(double *left)
 {
   left[0] = -5.0;
   left[1] = -5.0;
 }

 void getrightmargin(double *right)
 {
   right[0] = 5.0;
   right[1] = 5.0;
 }

 double funmin(double *x)
 {
   double x1=x[0],x2=x[1];
   return 4*x1*x1-2.1*x1*x1*x1*x1+
     x1*x1*x1*x1*x1*x1/3.0+x1*x2-4*x2*x2+4*x2*x2*x2*x2;
 }

 void granal(double *x, double *g)
 {
   double x1=x[0],x2=x[1];
   g[0]=8*x1-8.4*x1*x1*x1+2*x1*x1*x1*x1*x1+x2;
   g[1]=x1-8*x2+16*x2*x2*x2;
 }
}
```

Fig. 2. Implementation of Camel function in C++.

```
integer function getdimension()
getdimension = 2
end

subroutine getleftmargin(left)
double precision left(2)
left(1) = -5.0
left(2) = -5.0
end

subroutine getrightmargin(right)
double precision right(2)
right(1) = 5.0
right(2) = 5.0
end

double precision function funmin(x)
double precision x(2)
double precision x1,x2
x1=x(1)
x2=x(2)
funmin=4*x1**2-2.1*x1**4+x1**6/3.0+x1*x2-4*x2**2+4*x2**4
end

subroutine granal(x,g)
double precision x(2)
double precision g(2)
double precision x1,x2
x1=x(1)
x2=x(2)
g(1)=8.0*x1-8.4*x1**3+2*x1***5+x2;
g(2)=x1-8.0*x2+16.0*x2**3;
end
```

Fig. 3. Implementation of Camel function in Fortran 77.

References

[1] H.A. Bremermann, A method for unconstrained global optimization, Math. Biosci. 9 (4,8) (1970) 1–15.

[2] E.J. Beltrami, J.P. Indusi, An adaptive random search algorithm for constrained optimization, IEEE Trans. Automat. Control 17 (4) (1972) 1004–1007.

[3] R.A. Jarvis, Adaptive global search by the process of competitive evolution, IEEE Trans. Systems Man Cybergenetics 75 (4) (1975) 297–311.

[4] W.L. Price, Global optimization by controlled random search, Comput. J. 20 (1977) 367–370.

[5] S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing, Science 220 (4) (1983) 671–680.

[6] P.J.M. van Laarhoven, E.H.L. Aarts, Simulated Annealing: Theory and Applications, D. Riedel, Boston, 1987.

[7] A. Corana, M. Marchesi, C. Martini, S. Ridella, Minimizing multimodal functions of continuous variables with the "Simulated Annealing" algorithm, ACM Trans. Math. Software 13 (1987) 262–280.

[8] W.L. Goffe, G.D. Ferrier, J. Rogers, Global optimization of statistical functions with simulated annealing, J. Econometrics 60 (1994) 65–100.

[9] D. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Addison-Wesley Publishing Company, Reading, MA, 1989.

[10] Z. Michaelewizc, Genetic Algorithms + Data Structures = Evolution Programs, Springer-Verlag, 1996.

[11] R. Storn, K. Price, Differential evolution—a simple and efficient heuristic for global optimization over continuous spaces, J. Global Optimization 11 (1997) 341–359.

[12] M.M. Ali, A. Törn, Optimization of carbon and silicon cluster geometry for Tersoff potential using differential evolution, in: C.A. Floudas, P.M. Pardalos (Eds.), Optimization in Computational Chemistry and Molecular Biology, Kluwer Academic Publisher, 2000, pp. 287–300.

[13] M. O'Neill, Automatic programming in an arbitrary language: Evolving programs with grammatical evolution, PhD Thesis, University of Limerick, Ireland, August 2001.

[14] M. O'Neill, C. Ryan, Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language, Genetic Programming, vol. 4, Kluwer Academic Publishers, 2003.

[15] C. Ryan, M. O'Neill, J.J. Collins, Grammatical evolution: solving trigonometric identities, in: Proceedings of Mendel 1998: 4th International Mendel Conference on Genetic Algorithms, Optimization Problems, Fuzzy Logic, Neural Networks, Rough Sets, Brno, Czech Republic, 24–26 June 1998, pp. 111–119.

[16] M. O'Neill, J. Collins, C. Ryan, Automatic generation of robot behaviors using grammatical evolution, in: Proc. of AROB 2000, the 5th Internat. Symp. on Artificial Life and Robotics, pp. 351–354.

[17] M. O'Neill, C. Ryan, Automatic generation of caching algorithms, in: K. Miettinen, M.M. Mkel, P. Neittaanmki, J. Periaux (Eds.), Evolutionary Algorithms in Engineering and Computer Science, Jyvskyl, Finland, 30 May–3 June 1999, John Wiley & Sons, 1999, pp. 127–134.

[18] A. Brabazon, M. O'Neill, A grammar model for foreign-exchange trading, in: H.R. Arabnia, et al. (Eds.), in: Proceedings of the International Conference on Artificial Intelligence, 23–26 June 2003, vol. II, CSREA Press, 2003, pp. 492–498.

[19] M. O'Neill, C. Ryan, Grammatical evolution, IEEE Trans. Evolutionary Comput. 5 (2001) 349–358.

[20] J.R. Koza, Genetic Programming: On the Programming of Computer by Means of Natural Selection, MIT Press, Cambridge, MA, 1992.

[21] M.J.D. Powell, A tolerant algorithm for linearly constrained optimization calculations, Math. Programm. 45 (1989) 547.

[22] J. Nieminen, J. Yliluoma, Function Parser for C++, v2.7, available from http://www.students.tut.fi/~warp/FunctionParser/.

[23] D. Cvijoivic, J. Klinowski, Taboo search. An approach to the multiple minima problems, Science 667 (1995) 664–666.

[24] M.M. Ali, C. Storey, A. Törn, Application of some stochastic global optimization algorithms to practical problems, J. Optim. Theory Appl. 95 (3) (1997) 545–563.
[25] M. Gaviano, D.E. Ksasov, D. Lera, Y.D. Sergeyev, Software for generation of classes of test functions with known local and global minima for global optimization, ACM Trans. Math. Software 29 (2003) 469–480.
[26] F.V. Theos, I.E. Lagaris, D.G. Papageorgiou, PANMIN: sequential and parallel global optimization procedures with a variety of options for the local search strategy, Comput. Phys. Comm. 159 (2004) 63–69.