A PYTHON/C++ LIBRARY FOR BOUND-CONSTRAINED GLOBAL OPTIMIZATION USING BIASED RANDOM-KEY GENETIC ALGORITHM

R. M. A. SILVA, M. G. C. RESENDE, AND P. M. PARDALOS

Abstract. This paper describes `libbrkga`, a GNU-style dynamic shared Python/C++ library of the biased random-key genetic algorithm (BRKGA) for bound constrained global optimization. BRKGA (Gonçalves and Resende, 2011) is a general search metaheuristic for finding optimal or near-optimal solutions to hard optimization problems. It is derived from the random-key genetic algorithm of Bean (1994), differing in the way solutions are combined to produce offspring. After a brief introduction to BRKGA, we show how to download, install, configure, and use the library through an illustrative example.

1. Introduction

The objective of global optimization is to find a minimum or maximum of a multimodal function over a discrete or continuous domain. In its minimization form, global optimization is stated mathematically as finding a solution $x^* \in S \subseteq \mathbb{R}^n$ such that $f(x^*) \leq f(x)$, $\forall\ x \in S$, where S is some region of $\mathbb{R}^n$ and the multimodal objective function f is defined by $f : S \to \mathbb{R}$. Such a solution x^* is called a *global minimum*. In this paper, we limit ourselves to box constraints, i.e. the domain S is a hyper-rectangle $S = \{x = (x_1, \ldots, x_n) \in \mathbb{R}^n : \ell \leq x \leq u\}$, where $\ell, u \in \mathbb{R}^n$ such that $\ell \leq u$. Therefore, the minimization problem considered in this paper consists in finding $x^* = \text{argmin}\{f(x) \mid\ \ell \leq x \leq u\}$, where $f : \mathbb{R}^n \to \mathbb{R}$, and $\ell, x, u \in \mathbb{R}^n$.

Biased random-key genetic algorithm (BRKGA) is a general search metaheuristic for finding optimal or near-optimal solutions of hard optimization problems (Gonçalves and Resende, 2011). It is derived from the random-key genetic algorithm of Bean (1994), differing in the way solutions are combined to produce offspring. BRKGAs have three key features that specialize genetic algorithms:

- A fixed *chromosome* enconding using a vector of n random keys or *alleles* over the internal $[0, 1]$, where the value of n depends on the instance of the optimization problem;
- A well-defined evolutionary process adopting parameterized uniform crossover (Spears and DeJong, 1991) to generate *offspring* and thus evolve the population;
- The introduction of new chromosomes called *mutants* in place of the mutation operator usually found in genetic algorithms.

Key words and phrases. Biased random-key genetic algorithm, Global optimization, multimodal functions, continuous optimization, heuristic, stochastic algorithm, stochastic local search, nonlinear programming.

In this paper, we describe the BRKGA library `libbrkga`, a GNU-style dynamic shared Python/C++ library of the biased random-key genetic algorithm. The library was developed using the `autoconf`, `automake`, and `libtool` packages (Calcote, 2010), as well as part of C++ application programming interface for BRKGA developed by Toso and Resende (2012).

The library `libbrkga` is implemented as an embedded *Python-in-C* code (van Rossum and Drake Jr., 2010a;b) to take advantage of the simplicity offered by the Python programming language in implementing complex multimodal functions. Besides having access to the extensive standard library of Python, any non-standard module or library, such as `SymPy` (SymPy, 2011), can be used to implement a function. An important feature of our library is that the functions implemented in Python are loaded automatically without the need to recompile any code.

The paper is organized as follows. The BRKGA heuristic is reviewed in Section 2 and 3. Section 4 shows how to download, install, configure, and use `libbrkga`. An illustrative example is given in Section 5. Concluding remarks are made in Section 6.

2. Biased random-key genetic algorithms

Genetic algorithms with random keys, or *random-key genetic algorithms* (RKGA), were first introduced by Bean (1994) for solving combinatorial optimization problems involving sequencing. In a RKGA, chromosomes are represented as vectors of randomly generated real numbers in the interval $[0, 1]$. A deterministic algorithm, called a *decoder*, takes as input a solution vector and associates with it a solution of the combinatorial optimization problem for which an objective value or fitness can be computed.

A RKGA evolves a population of random-key vectors over a number of iterations, called *generations*. The initial population is made up of p vectors of random-keys. Each component of the solution vector is generated independently at random in the real interval $[0, 1]$. After the fitness of each individual is computed by the decoder in generation k, the population is partitioned into two groups of individuals: a small group of p_e *elite* individuals, i.e. those with the best fitness values, and the remaining set of $p - p_e > p_e$ *non-elite* individuals. To evolve the population, a new generation of individuals must be produced. All elite individuals of the population of generation k are copied without modification to the population of generation $k + 1$. RKGAs implement mutation by introducing *mutants* into the population. A mutant is simply a vector of random keys generated in the same way that an element of the initial population is generated. At each generation, a small number (p_m) of mutants is introduced into the population. With the p_e elite individuals and the p_m mutants accounted for in population $k + 1$, $p - p_e - p_m$ additional individuals need to be produced to complete the p individuals that make up the new population. This is done by producing $p-p_e-p_m$ offspring through the process of mating or crossover.

Figure 1 illustrates the evolution dynamics. On the left of the figure is the current population. After all individuals are sorted by their fitness values, the best fit are placed in the elite partition labeled ELITE and the remaining individuals are placed in the partition labeled NON-ELITE. The elite random-key vectors are copied without change to the partition labeled TOP in the next population (on the right side of the figure). A number of mutant individuals are randomly generated

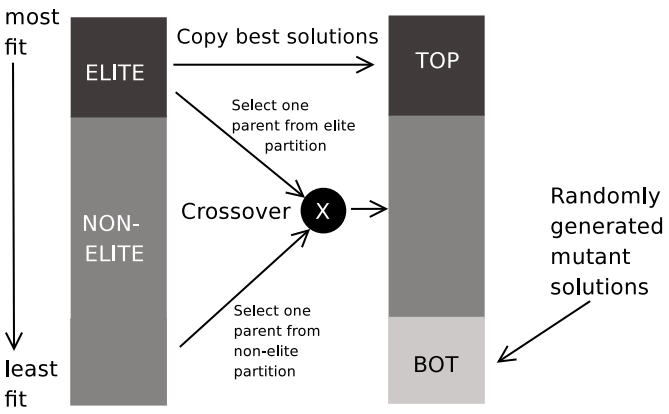

FIGURE 1. Transition from generation k to generation $k+1$ in a BRKGA.

and placed in the new population in the partition labeled BOT. The remainder of the population of the next generation is completed by crossover. In a RKGA, Bean (1994) selects two parents at random from the entire population. A *biased random-key genetic algorithm,* or BRKGA (Gonçalves and Almeida, 2002; Ericsson et al., 2002; Gonçalves and Resende, 2004), differs from a RKGA in the way parents are selected for mating. In a BRKGA, each element is generated combining one element selected at random from the partition labeled ELITE in the current population and one from the partition labeled NON-ELITE. In some implementations, the second parent has been selected from the entire population. Repetition in the selection of a mate is allowed and therefore an individual can produce more than one offspring. Since we require that $p_e < p - p_e$, the probability that an elite individual is selected for mating is greater than that of a non-elite individual and therefore the elite individual has a higher likelihood to pass on its characteristics to future generations. Another factor contributing to this end is *parameterized uniform crossover* (Spears and DeJong, 1991), the mechanism used to implement mating in BRKGAs. Let $\rho_e > 0.5$ be the probability that an offspring inherits the vector component of its elite parent. Let n denote the number of components in the solution vector of an individual. For $i = 1, \ldots, n$, the i-th component $c(i)$ of the offspring c takes on the value of the i-th component $e(i)$ of the elite parent e with probability ρ_e and the value of the i-th component $\bar{e}(i)$ of the non-elite parent $\bar{e}$ with probability $1 - \rho_e$.

Figure 2 illustrates the crossover process for two random-key vectors with four components each. Chromosome 1 refers to the elite individual and Chromosome 2 to the non-elite one. In this example the value of $\rho_e = 0.7$, i.e. the offspring inherits the component of the elite parent with probability 0.7 and of the other parent with probability 0.3. A randomly generated real in the interval $[0, 1]$ simulates the toss of a biased coin. If the outcome is less than or equal to 0.7, then the child inherits the component of the elite parent. Otherwise, it inherits the component of the other parent.

When the next population is complete, i.e. when it has p individuals, fitness values are computed for all of the newly created random-key vectors and the population is partitioned into elite and non-elite individuals to start a new generation.

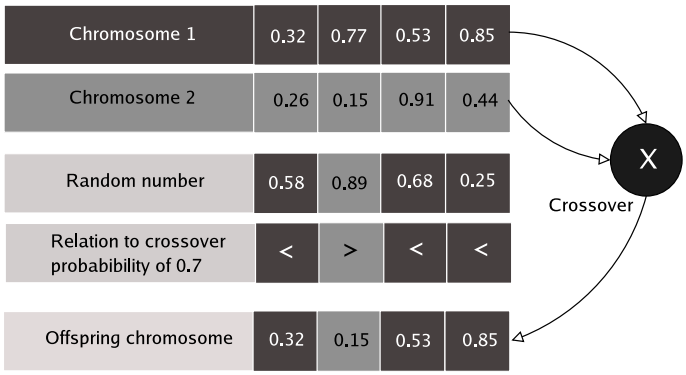

FIGURE 2. parameterized uniform crossover: mating in BRKGAs.

BRKGA heuristics are based on a general-purpose metaheuristic framework. In this framework, depicted in Figure 3, there is a clear divide between the *problem-independent* portion of the algorithm and the *problem-dependent* part. The problem-independent portion has no knowledge of the problem being solved. The only connection to the optimization problem being solved is the problem-dependent portion of the algorithm, where the decoder produces solutions from the vectors of random-keys and computes the fitness of these solutions. Therefore, to specify a BRKGA heuristic one need only define its chromosome representation and the decoder.

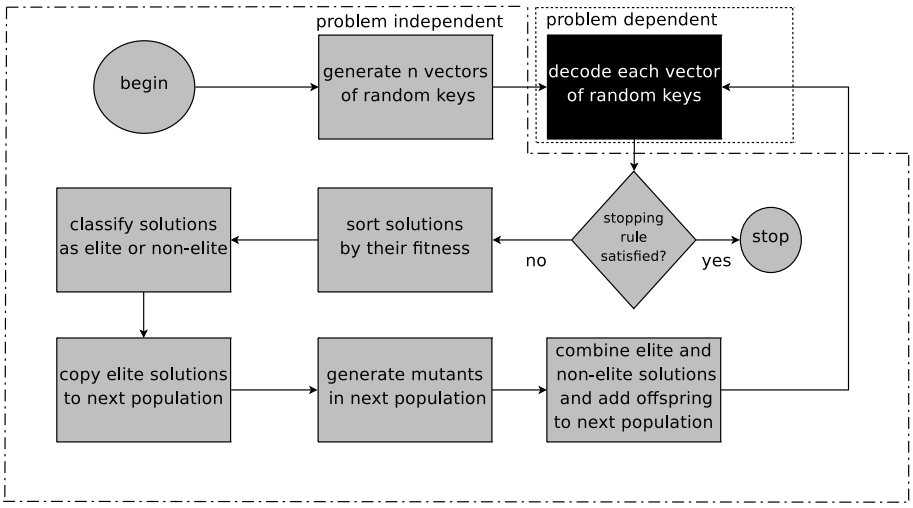

FIGURE 3. Flowchart of a BRKGA

A BRKGA for bound-constrained global optimization was proposed by Silva et al. (2012). To describe this BRKGA, one needs only to show how solutions are encoded as vectors of random keys and how these vectors are decoded to feasible solutions of the problem:

- *Encoding a solution to a vector of random keys.* A solution is encoded as a vector $\chi = (\chi_1, ..., \chi_n)$ of size n, where χ_i is a random number in the

interval $[0, 1]$, for $i = 1, \ldots, n$. The i-th component of χ corresponds to the i-th dimension of hyper-rectangle S.

- *Decoding a solution from a vector of random keys.* First, the decoder takes as input the vector of random keys χ and generates a solution $x \in S$ with $x_i = l_i + \chi_i \cdot (u_i - l_i)$, for $i = 1, \ldots, n$. After, the decoder proceed by trying to improve x using the local search described in the next section. The new solutions $x = (x_1, \ldots, x_n)$ produced by the local search usually disagree with the genes initially supplied in the vector of random keys to the decoder. In these cases, in order to reflect the changes made by the local search phase of the decoder, the components of the chromosomes are updated with the following values $\chi_i = (x_i - l_i)/(u_i - l_i)$, for $i = 1, \ldots, n$. During all decoder process, the solutions fitness are calculated by the objective function $f : S \to \mathbb{R}$ of the global optimization problem.

3. Local improvement procedure

Inspired by the local search introduced in Hirsch et al. (2007; 2010), our local improvement phase (with pseudo-code shown in Figure 4) can be seen as *approximating* the role of the gradient of the objective function $f(\cdot)$. From a given input point $x \in \mathbb{R}^n$, the local improvement algorithm generates a neighborhood, and determines at which points in the neighborhood, if any, the objective function improves. If an improving point is found, it is made the current point and the local search continues from the new solution.

Let $\bar{x} \in \mathbb{R}^n$ be the current solution and h be the current grid discretization parameter. Define $S_h(\bar{x}) = \{x \in S \mid \ell \leq x \leq u,\ x = \bar{x} + \tau \cdot h,\ \tau \in Z^n\}$ to be the set of points in S that are integer steps (of size h) away from $\bar{x}$. Let $B_h(\bar{x}) = \{x \in S \mid x = \bar{x} + h \cdot (x' - \bar{x})/\|x' - \bar{x}\|,\ x' \in S_h(\bar{x}) \setminus \{\bar{x}\}\}$ be the projection of the points in $S_h(\bar{x}) \setminus \{\bar{x}\}$ onto the hyper-sphere centered at $\bar{x}$ of radius h. The *h-neighborhood* of the point $\bar{x}$ is defined as the set of points in $B_h(\bar{x})$.

The procedure takes as input a starting solution $x \in S \subseteq \mathbb{R}^n$, the objective function $f(\cdot)$, lower and upper bound vectors ℓ and u, as well as the parameters h_s and h_e, the starting and ending grid discretization densities, respectively. The maximum number of points `MaxPointsToExamine` $\leq \prod_{i=1}^{n} \lceil (u_i - \ell_i)/h \rceil$ in $B_h(x^*)$ that are to be examined is also taken as an input parameter. If all of these points are examined and no improving point is found, the current solution x^* is considered an *h-local minimum*.

The current best local improvement solution x^* is initialized to x in line 1. In line 2, the objective function value f^* of the best solution found is initialized to $f(x)$. Next, the parameter h, that controls the discretization density of the search space, is initialized to h_s in line 3, and in line 4 the variable `Impr` is set to **false**. Starting at the point x^*, in the loop in lines 7–16, the algorithm randomly selects points in $B_h(x^*)$ (line 8), one at a time. In line 9, if the current point x selected from $B_h(x^*)$ is feasible and is better than x^*, then x^* is set to x (line 10), f^* is set to $f(x)$ (line 11), `NumPointsExamined` is set to zero (line 12), `Impr` is set to **true** (line 13), and the loop in lines 7–16 restarts with x^* as the starting solution. In line 17, if the variable `Impr` is still set to false, then in line 20 the grid density is increased by halving h, and the loop in lines 7–16 is re-initialized if $h \geq h_e$. Local improvement is terminated if an *h-local minimum* solution x^* is found. At that point, x^* is returned from the local improvement procedure in line 18 or 23.

```
procedure LocalImprovement(x, f(·), h_s, h_e, ℓ, u, MaxPointsToExamine)
1     x* ← x;
2     f* ← f(x);
3     h ← h_s;
4     Impr ← false;
5     while h ≥ h_e do
6           NumPointsExamined ← 0;
7           while NumPointsExamined ≤ MaxPointsToExamine do
8                 x ← RandomlySelectElement(B_h(x*));
9                 if ℓ ≤ x ≤ u and f(x) < f* then
10                      x* ← x;
11                      f* ← f(x);
12                      NumPointsExamined ← 0;
13                      Impr ← true;
14                end if
15                NumPointsExamined ← NumPointsExamined + 1;
16          end while
17          if Impr = true then
18                return x*;
19          else
20                h ← h/2;
21          end if
22    end while
23    return x*;
end LocalImprovement;
```

FIGURE 4. Pseudo-code for local improvement phase.

4. The library

This section begins by showing how to download (Section 4.2), build (Section 4.3), and install (Section 4.4) the `libbrkga` library, as well as its package dependencies (Section 4.1). Then, the format of components required to use the library are presented as follows: function module (Section 4.5), parameter input file (Section 4.6), and calling C++ program (Section 4.7). Finally, the format of the output file is described in Section 4.8.

4.1. **Dependencies.** The `libbrkga` library requires that the following packages be installed:

- Python programming language package (version $\geq$ 2.7), available at `http://www.python.org/download`;
- GNU Libtool library, available at `http://www.gnu.org/software/libtool/`.

4.2. **Downloads.** Full distribution of the `libbrkga` library is available at `http://www.research.att.com/~mgcr/src/libbrkga`. The package is distributed as the tar file `brkga-0.0.1.tar.gz` containing the following directory structure:

```
.:
AUTHORS    configure   INSTALL    NEWS    src
ChangeLog  COPYING     Makefile   README  THANKS
```

```
./src:
brkgagopt.cpp       GoptDecoder.h   mt19937ar.c   Makefile
brkgagoptparser.py  BRKGA.h         mt19937ar.h
```

Each file of this directory is described in Table 1.

TABLE 1. Main source code files of the `libbrkga` library

files	description
`brkgagopt.cpp`	Embedded Python-in-C++ code of `BRKGA`
`BRKGA.h`	C++ header file of `brkgagopt.cpp`
`mt19937ar.c`	C source code of the Mersenne Twister random number generator of Matsumoto and Nishimura (1998)
`mt19937ar.h`	C header file of `mt19937ar.c`
`GoptDecoder.h`	C++ header file of BRKGA's decoder
`brkgagoptparser.py`	Parser for parameter input file
`AUTHORS`	Names and e-mail addresses of the authors
`ChangeLog`	Records the changes that are made to package
`configure`	Script that configures the package automatically
`COPYING`	GNU General Public License
`INSTALL`	Instructions for installing a GNU package
`Makefile`	File which `make` will read to build the library
`NEWS`	A record of user-visible changes to the package
`README`	Purpose of package and installation instructions
`THANKS`	Thanks to contributors

4.3. **Building.** The `libbrkga` library was designed to run on a Linux platform. Building the library from a distribution source tarball does not require `autoconf` and `automake` packages to be installed. To build the library, execute the following steps:

(1) unzip and untar the distribution `brkga-0.0.1.tar.gz` source tarball:

```
$ tar -xvf brkga-0.0.1.tar.gz
```

(2) Run the configure script to create the Makefiles:

```
$ cd brkga-0.0.1
$ ./configure
```

(3) Run the top-level Makefile:

```
$ make
```

The `configure` command invokes a shell script that is distributed with the package that automatically configures the library. It first probes the target system to determine parameters needed to generate a `Makefile` from a template stored in the file `Makefile.am`. When invoked, `make` executes the `Makefile` which compiles the source code of the package but does not install it.

4.4. **Installation.** To install the library, `make` is once again invoked, this time with the target `install`:

```
$ make install
```

Note that to install the library in some system directories, such as `/usr/local`, requires super-user privilege. During installation, the files are placed in specific directories, as follows:

- `/usr/local/lib` directory receives the libraries:
  `libbrkga.a  libbrkga.la  libbrkga.so`
  `libbrkga.so.0  libbrkga.so.0.0.0`
- `/usr/local/include`, the header files:
  `BRKGA.h  mt19937ar.h  GoptDecoder.h`
- `/usr/local/lib/python2.7/site-packages/brkga`, the Python script:
  `brkgagoptparser.py`

The `/usr/local` directory is called the *prefix*. The default prefix is always `/usr/local` but this can be set to any other directory when `configure` is invoked by adding a `--prefix` option. For example, suppose a user wants to install the package in directory `/home/username` instead of `/usr/local`:

```
$ ./configure --prefix=/home/username
$ make
$ make install
```

The `--prefix` argument tells `configure` where you want to install your package, and `configure` will take that into account and build the proper `Makefile` automatically.

4.5. **Function module implementation.** Objective functions are implemented using the Python language. Consider as an example the Ackley function (Ackley, 1987; Bäck, 1996),

$$A_n(x) = -20e^{-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}} - e^{\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)} + 20 + e, \tag{1}$$

which can be implemented in Python as follows:

```
from math import *
def f(x):
    sum1 = sum( x[i]**2 for i in range(len(x)) )
    sum2 = sum( cos(2*pi*x[i]) for i in range(len(x)) )
    r = 1.0/len(x)
    return -20.0*exp(-0.2*sqrt(r*sum1))-exp(r*sum2)+20.0+e
```

In Python, the keyword `def` introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented. At the end, a `return` statement returns a value. Therefore, in the above example, `f` is the name of the function, array `x` its parameter, and its body has four statements.

Python has a way to put definitions and statements in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module* and definitions from a module can be *imported* into other modules. For example, the first line `from math import *` above imports all the definitions from

the standard Python module `math` to the user's module `ackley`. The file name is the module name with suffix `.py` appended (e.g. `ackley.py`).

4.6. **Input file formats.** The input file must contain the following entries:

- `-n <n>`: sets the number of alleles per chromosome (parameter n) to the positive integer value `<n>`;
- `-p <n>`: sets the number of chromosomes in population (parameter p) to the positive integer value `<n>`;
- `-pe <n>`: sets the size of the elite set in population (parameter p_e) to the positive integer value `<n>`;
- `-pm <n>`: sets the number of mutants to be introduced in population at each generation (parameter p_m) to the positive integer value `<n>`;
- `-rho <n>`: sets the probability that an allele is inherited from the elite parent (parameter ρ_e) to the real number $0 \leq$ `<d>` ≤ 1;
- `-md <module-name>`: defines the name of the python module containing the multimodal function(s) to be minimized;
- `-ft <function-name>`: defines the name of the Python function that implements the multimodal function to be minimized;
- `-ds <n>`: sets the function dimension to the positive integer `<n>`;
- `-dm <l> <u> [list-of-exceptions]`: sets bounds of the hyper-rectangle $S = \{x = (x_1, \ldots, x_n) \in \mathbb{R}^n : \ell \leq x \leq u\}$, such that $\ell_i =$ `<l>` and $u_i =$ `<u>` for all $(i = 1, \ldots, n)$ dimensions. For example, `-dm -10 10` sets the lower and upper bounds for all dimensions to -10 and 10, respectively; Exceptions are used to specify bounds for dimensions for which bounds are different from `<l>` or `<u>`. They are expressed as follows:
  (1) `<i> <lo> <up>`, with $1 \leq$ `<i>` $\leq n$ and `<lo>` $\leq$ `<up>`: sets the lower ℓ_i and upper u_i bounds of i-th dimension to `<lo>` and `<up>`, respectively. For example, the exception `3 -12 20` sets the lower and upper bounds of the third dimension to -12 and 20, respectively.
  (2) `<i>:<j> <lo> <up>`, with $1 \leq$ `<i>` $\leq$ `<j>` $\leq n$ and `<lo>` $\leq$ `<up>`: sets the lower bounds ℓ_k to `<lo>`, and the upper bounds u_k to `<up>`, for all dimensions $k = i, \ldots, j$. For example, exception `7:10 -13 17` sets the lower and upper bounds of 7th to the 10th dimensions to -13 and 17, respectively;
  (3) combinations between formats (1) and (2) above described. For example, `2 1 15 4:6 -9 -3 7 -15 30 9:11 -5 5` sets $\ell_2 = 1$ and $u_2 = 15$; $\ell_4 = \ell_5 = \ell_6 = -9$ and $u_4 = u_5 = u_6 = -3$; $\ell_7 = -15$ and $u_7 = 30$; $\ell_9 = \ell_{10} = \ell_{11} = -5$, $u_9 = u_{10} = u_{11} = 5$;
- `-ov <d>` or `-it <n>` or `-fe <n>`: sets the target optimal objective function value to the real number `<d>` or sets the number of iterations to the positive integer value `<n>` or sets the number of function evaluations to the positive integer value `<n>`. Note that only one of the three entries can be used as the stopping criterion, i.e. they cannot be used in pairs or all together; Furthermore, option `-ov <d>` is only used when the optimal objective function value is known a priori. Otherwise, option `-it <n>` or option `-fe <n>` is used.
- `-ep <d>`: sets parameter ϵ to the positive real number `<d>`; Note that this entry can only be used in conjunction with `-ov <d>`.

- `-sd <n>`: sets the seed of the pseudo-random generator to the positive integer `<n>`;
- `-hs <d>`: sets the starting grid discretization density h_s to the positive real number `<d>` (this number could be, for example, $0.1 \cdot \min\{u_i - l_i : i = 1, \ldots, n\})$;
- `-he <d>`: sets the ending grid discretization density h_e to the positive real number `<d>` (this number must be smaller than the one set by `-hs <d>` and in case `-ep <d>` is used it should be no larger than the parameter set by `-ep <d>`);
- `-mp <n>`: sets the parameter `MaxPointsToExamine` in the local improvement procedure to the positive integer `<n>`;
- `-of <file-name>`: defines the name of the output file to which the solution is written.

An example input file is:

```
-n 5 -p 100 -pe 30 -pm 20 -rho 0.7 -sd 270001
-hs 0.5 -he 0.0001 -mp 100 -of output.file
-md ackley -ft f -ds 5 -ov 0 -ep 0.0001
-dm -10 10 1 -5 3 4:5 -13 7
```

This input file specifies that BRKGA will try to find a solution $x' \in S = \{x = (x_1, \ldots, x_5) \in \mathbb{R}^5 : (-5, -10, -10, -13, -13) \leq x \leq (3, 10, 10, 7, 7)\}$, such that function `f` of module `ackley.py` that implements A_5 (Equation 1) will be such that $GAP = |A_5(x') - 0| \leq \epsilon = 0.0001$, using the following parameters: $n = 5$, $p = 100$, $p_e = 30$, $p_m = 20$, $\rho_e = 0.7$, $h_s = 0.5$, $h_e = 0.0001$, $seed = 270001$, and `MaxPointsToExamine` $= 100$.

The program that parses this input file format was developed with Pyparsing (McGuire, 2007).

4.7. **Using the library in C++.** To use the function `double brkga(int, **char)` of the `libbrkga` library in a C++ program (which we shall call here `userprog.cpp`):

(1) Put `#include <brkga>` in the source code of the C++ program `userprog.cpp`:

```
#include <brkga>
...
using namespace std;

void main(int argc, char **argv){
  ...
  double x;
  x = brkga(argc,argv);
  ...
}
```

(2) Link `userprog.cpp` with the `libbrkga` library at compilation time, recalling to specify the `<pathname>` of the `Python.h` header file:

```
$ g++ -I<pathname> userprog.c -o userprog -lbrkga
```

To support embedding, the Python Application Programming Interface (API) defines a set of functions, macros, and variables that provide access to most aspects

of the Python run-time system. The Python API is incorporated in a C++ source file by including the header `Python.h`.

Before running the program, the environment variables `LD_LIBRARY_PATH` and `PYTHONPATH` must be set appropriately. `LD_LIBRARY_PATH` contains a colon-separated list of directories in which the dynamic linker should search for shared objects. Therefore, to inform the dynamic linker where the Python API is installed (more specifically where the `Python.h` header file is located), `LD_LIBRARY_PATH` must be set with the Python libraries directory pathname:

```
$ export LD_LIBRARY_PATH=<python-libs-dir>
```

For example:

```
$ export LD_LIBRARY_PATH=/usr/local/lib
```

`PYTHONPATH` also contains a colon-separated list of directories, similar to `PATH` in so far as it defines a search path. However, unlike `PATH` (which specifies to the operating system in which directories to look for executable files), `PYTHONPATH` is used by the Python interpreter to locate modules to import. Therefore, the location of the Python modules that implement the multimodal functions to be minimized must be specified in `PYTHONPATH`:

```
$ export PYTHONPATH=<python-modules-dirs>
```

For example, the command:

```
$ export PYTHONPATH=$PWD:/usr/local/lib/python2.7/site-packages/brkga
```

sets `PYTHONPATH` to the current directory `$PWD`, to specify the directory of the function module and `/usr/local/lib/python2.7/site-packages/brkga` to specify the directory of the BRKGA input parser `brkgagoptparser`.

Finally, to run the program, type:

```
$ <program_name> <input_file_name>
```

as for example:

```
$ ./userprog input
```

4.8. **Output.** The program produces three kinds of output:

- `STDERR` (terminal): occasional error messages;
- `STDOUT` (terminal, unless redirected to a file with `>`) and `FILE` (file name specified by the "`-of`" option in the input file):
  (1) For each objective function improvement, a line is printed with the following format:

      ```
      <keyword> <value>
      ```

      Keywords are self-descriptive:
      - `time`: CPU time (in seconds) of improvement;
      - `best value`: objective function value of improved solution;
      - `chromosome`: values of alleles from chromosome responsible for the improved solution;
      - `solution`: improved solution $x = (x_1, \ldots, x_n) \in S \subset \mathbb{R}^n$.

  (2) Total CPU time (in seconds) in the following format:
      `time: <value>`
      For example:
      `time: 165.650009`

(3) Value of the overall best solution found in the following format:
`optimum: <value>`
For example:
`optimum: 0.000000`
(4) Overall best solution found in the following format:
`solution: <value>`
For example:
`solution: 1.042637 3.074020`

Consider as an example the output generated by BRKGA to find a solution $x \in [-10, 10]^2$, such that the **Booth** function:

$$BO(x) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2 \leq \epsilon = 0.001 \tag{2}$$

is minimized using the following parameters: $n = 2$, $p = 100$, $p_e = 30$, $p_m = 20$, $\rho_e = 0.7$, $h_s = 0.5$, $h_e = 0.0001$, $seed = 270001$, and `MaxPointsToExamine` $= 100$.

```
time: 0
best value: 0.364822436897729
chromosome: 0.555903959151367 0.658309974830011
solution: 1.11807918302734 3.16619949660022
time: 0
best value: 0.346961865714319
chromosome: 0.555724931575396 0.658135478047686
solution: 1.11449863150791 3.16270956095372
time: 0.0100000016391277
best value: 0.338488940337037
chromosome: 0.555406193917939 0.658273567572868
solution: 1.10812387835878 3.16547135145736
...
time: 0.0199999995529652
best value: 0.0797904488735659
chromosome: 0.552521069310988 0.65411561060226
solution: 1.05042138621977 3.0823122120452
time: 0.0300000011920929
best value: 0.0725596441205848
chromosome: 0.552457758851641 0.653873759851306
solution: 1.04915517703282 3.07747519702611
...
time: 0.0399999991059303
best value: 0.00208143439742911
chromosome: 0.549781849591604 0.651186244074817
solution: 0.995636991832074 3.02372488149633
time: 0.0399999991059303
best value: 0.0011093298963903
chromosome: 0.549845019966439 0.650862914454092
solution: 0.996900399328776 3.01725828908185
time: 0.0399999991059303
best value: 0.000415380069146002
chromosome: 0.54974127275453 0.65063545782961
solution: 0.9948254550906 3.01270915659219
```

The global minimum of Booth function in domain $[-10, 10]^2$ is $x^* = (1, 3)$ with $BO(x^*) = 0$. `brkga` reached an ϵ-optimal solution in 0.04 seconds.

5. An example

In this section, we illustrate the use of the library with an example. We give step-by-step instructions on how to solve the example problem.

(1) Create the following program with your favorite editor:

```
#include <brkga>
using namespace std;
double main(int argc, char **argv){
  double res;
  res = brkga(argc,argv);
  return res;
}
```

and save the code in the file `program.cpp`.

(2) Implement the function to be minimized as a Python module. For example, the Booth function described in Equation (2) can be implemented as:

```
def g(x):
   return ( x[0] + 2*x[1] - 7 )**2 + ( 2*x[0] + x[1] - 5)**2
```

and save the module in the file `booth.py`.

(3) Create a file with the parameters to be used by program, as for example:

```
-n 2 -p 100 -pe 30 -pm 20 -rho 0.7 -sd 270002
-hs 0.5 -he 0.0001 -mp 100 -of output.file
-md booth -ft g -ds 2 -ov 0 -ep 0.001 -dm -10 10
```

and name it, for example, `input`. Do not forget to set the options `-md` and `-ft` to the file name and function name, respectively. In this example, `-md` and `-ft` assume the values `booth` and `g`, respectively.

(4) Compile the program `program.cpp`:

```
$ g++ -I/usr/local/include/python2.7 program.cpp -o program -lbrkga
```

in order to create an executable file `program`.

(5) Update the environment variables `LD_LIBRARY_PATH` and `PYTHONPATH`:

```
$ export LD_LIBRARY_PATH=/usr/local/lib
$ export PYTHONPATH=\$PWD:/usr/local/lib/python2.7/site-packages/brkga
```

(6) Type the following command to run the program:

```
$ ./program input
```

which will generate the following output:

```
time: 0
best value: 12.4012134738348
chromosome: 0.581168300768524 0.548574313721584
solution: 1.62336601537048 0.971486274431687
time: 0
best value: 12.3474407799779
chromosome: 0.581425563994368 0.548582444743471
solution: 1.62851127988736 0.971648894869421
```

```
time: 0
best value: 1.37095603268167
chromosome: 0.581271020237308 0.606722778504266
solution: 1.62542040474616 2.13445557008533
...
time: 0
best value: 0.00557839553419341
chromosome: 0.551554290499392 0.647371106326124
solution: 1.03108580998784 2.94742212652249
time: 0.0100000016391277
best value: 0.00469662572651842
chromosome: 0.551673490240309 0.647503577550343
solution: 1.03346980480617 2.95007155100685
time: 0.0100000016391277
best value: 0.00386471817351207
chromosome: 0.551771612778688 0.647686909115509
solution: 1.03543225557376 2.95373818231018
...
time: 0.0100000016391277
best value: 0.00161491569733645
chromosome: 0.551430859265283 0.648589965864915
solution: 1.02861718530566 2.97179931729831
time: 0.0100000016391277
best value: 0.00124031772545151
chromosome: 0.551235158765371 0.64874553358115
solution: 1.02470317530743 2.97491067162299
time: 0.0100000016391277
best value: 0.00093213902486317
chromosome: 0.551099949712698 0.648945371002258
solution: 1.02199899425395 2.97890742004516
```

Suppose that you decide to change the function to be optimized. For example, instead of Booth function, consider the Ackley function described in Equation (1). The necessary steps to incorporate this new function are as follows:

(1) Implement the Ackley function in Python as described in Section 4.5 and save it in file `ackley.py`.
(2) Update at least the options related to the function in the file `input`: `-md`, `-ft`, `-ds`, `-ov`, and `-dm`. For example:

```
-n 2 -p 100 -pe 30 -pm 20 -rho 0.7 -sd 270001
-hs 0.5 -he 0.0001 -mp 100 -of output.file
-md ackley -ft f -ds 30 -ov 0 -ep 0.001 -dm -15 30
```

(3) Run the program again:

```
$ ./program input
```

which will generate the following output:

```
time: 0.00999999791383743
best value: 19.1725252961828
chromosome:       0.638643671604371  0.905998561724551  0.281621670470377 0.513249500668442
0.348389990408839  0.665679093633857  0.0107783206790348 0.190419937479696 0.641232143378347
0.677723271158611  0.0966381069453744 0.45433941498349   0.512932069438723 0.150989073578167
```

```
0.443569251061121  0.346650077875665  0.458253420734324  0.434912044616373 0.468066306535667
0.0531229862946047 0.145311589717316  0.59960529854232   0.352107937350035 0.42377728797228
0.423693094890408  0.572061629506032  0.352959659627264  0.506297421802206 0.776051438377806
0.450260529856232
solution:           13.7389652221967  25.7699352776048  -2.32702482883303  8.09622753007989
  0.677549568397747 14.9555592135236 -14.5149755694434  -6.43110281341369 13.8554464520256
 15.4975472021375  -10.6512851874582   5.44527367425705  8.08194312474253 -8.20549168898247
  4.96061629775047   0.599253504404928 5.62140393304457  4.57104200773676  6.06298379410501
-12.6094656167428   -8.46097846272078 11.9822384344044   0.844857180751566 4.0699779587526
  4.06618927006835  10.7427733277714   0.883184683226903 7.78338398109926 19.9223147270013
  5.26172384353044
time: 0.00999999791383743
best value: 19.0968958877304
chromosome:       0.638651165932213  0.222323366475622  0.281619427856971  0.513232053763447
0.429619020665968 0.436421305732719  0.0107883712725076 0.190419578739964  0.641213139974188
0.677713785139143 0.0440928567274927 0.454326880656463  0.867414208090463  0.15098525094168
0.156826504491548 0.308829861748414  0.458263763416431  0.0627351854969807 0.468057902616373
0.313481387175526 0.388080379541698  0.599588304472725  0.352115702202921  0.423787079802667
0.423689611390388 0.831719401838573  0.352964340886717  0.506304447612037  0.776053118377535
0.457594001455955
solution:          13.7393024669496    -4.99954350859702   -2.32712574643631 8.09544241935513
 4.33285592996854    4.63895875797235 -14.5145232927372   -6.43111895670163 13.8545912988384
 15.4971203312614 -13.0158214472628     5.44470962954085  24.0336393640708  -8.20566370762441
-7.94280729788035 -1.10265622132138     5.62186935373938 -12.1769166526359    6.0626056177368
-0.893337577101345 2.46361707937639    11.9814737012726    0.845206599131426 4.07041859112003
 4.06603251256747 22.4273730827358      0.883395339902266  7.78370014254167 19.9223903269891
 5.59173006551798
...
time: 18.1300010681152
best value: 0.00100373106409934
chromosome:       0.333334096034686 0.333336017947008 0.333331112934064 0.333338490667066
0.333336652873691 0.333332217329821 0.333357193290142 0.333337136934286 0.333334685523522
0.333333499304594 0.333336640021927 0.333334226179959 0.333327756504409 0.333337598339253
0.33333783622858 0.333336202547095 0.333340745454414 0.333337812065997 0.333335273082437
0.33333595488429 0.33333572862701 0.333333688741552 0.333338250239139 0.333337405859493
0.333335200543652 0.333333882032975 0.333337189067944 0.333332319347127 0.333340256542409
0.333333087405219
solution:             3.43215608644698e-05  0.000120807615378027 -9.99179671392625e-05
0.00023208001799091   0.000149379316100706 -5.02201580658834e-05  0.00107369805639301
0.000171162042875039  6.08485584692176e-05  7.46870671264332e-06  0.000148800986705666
4.01780981693634e-05 -0.000250957301608068  0.000191925266376458  0.000202630286095129
0.000129114619262793  0.000333545448608419  0.000201542969870516  8.72887096541319e-05
0.000117969793031136  0.000107788215473192  1.59933698178349e-05  0.00022126076124529
0.000183263677195455  8.40244643356414e-05  2.46914838886617e-05   0.00017350805746652
-4.56293792989726e-05 0.000311544408388542 -1.10667651469498e-05
time: 18.1400012969971
best value: 0.00097975917400861
chromosome:       0.333339508569033 0.333338747226589 0.333332520386603 0.333337881721986
0.333336029624341 0.333334655018578 0.333356036440079 0.333335457466844 0.333336448549794
0.333333586944863 0.333338466942912 0.333336634859702 0.333329395853438 0.333337800647458
0.333332217920314 0.333339070638648 0.333335360653192 0.333337762400589 0.333336214406694
0.333335596049069 0.333337203112934 0.333334447280189 0.33333928894749  0.333336221590173
0.333336382531488 0.333336271222185 0.333334268303523 0.33333176655625  0.333338190357919
0.33333591251443
solution:              0.000277885606484674 0.000243625196509001 -3.65826028509986e-05
 0.000204677489360705  0.00012133309535578   5.94758360161762e-05  0.0010216398035432
 9.55860079621118e-05  0.000140184740745397  1.14125188268588e-05  0.000231012431038735
 0.000148568686578088 -0.000177186595308854  0.000201029135604713 -5.01935858601144e-05
 0.000258178739159121  9.12293936554676e-05  0.000199308026521905   0.00012964830124850 8
 0.000101822208096181  0.000174140082046748  5.01276085262958e-05   0.00026800263703386
 0.000129971557797148  0.000137213916969969  0.000132204998333663   4.20736585393655e-05
-7.05049687699244e-05  0.000218566106358509  0.000116063149356194
```

6. Concluding remarks

In this paper, we describe how to download, install, configure, and use an implementation of the BRKGA heuristic for bound constrained global optimization. Since BRKGA makes no use of derivative nor *a priori* information, it is a well-suited approach for solving general global optimization problems.

The BRKGA library `libbrkga` was implemented in C++, and on the runs done for this paper it was compiled with the `g++` version 4.4.3 compiler with flags `-O6 -funroll-all-loops -fomit-frame-pointer`. The pseudo-random number generator adopted is the *Mersenne Twister* implemented by Matsumoto and Nishimura (1998) and available at `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html`.

All runs reported in this paper were done on a computer with a quad core 2.8 GHz 6 MB cache Intel i7 I7-720QM processor and 6 Gb of 1333 MHz DDR3 SD RAM memory, running Ubuntu 10.04 LTS (Lucid Lynx).

Acknowledgment

The research of R.M.A Silva was partially supported by the Brazilian National Council for Scientific and Technological Development (CNPq), the Foundation for Support of Research of the State of Minas Gerais, Brazil (FAPEMIG), Coordination for the Improvement of Higher Education Personnel, Brazil (CAPES), and Foundation for the Support of Development of the Federal University of Pernambuco, Brazil (FADE).

References

D.H. Ackley. *A connectionist machine for genetic hillclimbing*. Kluwer Academic Publishers, Boston, 1987.

T. Bäck. *Evolutionary algorithms in theory and practice*. Oxford University Press, New York, 1996.

J.C. Bean. Genetic Algorithms and Random Keys for Sequencing and Optimization. *ORSA J. on Computing*, 6:154–160, 1994.

J. Calcote. *Autotools: A practitioner's guide to GNU Autoconf, Automake, and Libtool*. No Starch Press, San Francisco, 2010.

M. Ericsson, M.G.C. Resende, and P.M. Pardalos. A genetic algorithm for the weight setting problem in OSPF routing. *J. of Combinatorial Optimization*, 6: 299–333, 2002.

J.F. Gonçalves and J. Almeida. A hybrid genetic algorithm for assembly line balancing. *J. of Heuristics*, 8:629–642, 2002.

J.F. Gonçalves and M.G.C. Resende. An evolutionary algorithm for manufacturing cell formation. *Computers and Industrial Engineering*, 47:247–273, 2004.

J.F. Gonçalves and M.G.C. Resende. Biased random-key genetic algorithms for combinatorial optimization. *J. of Heuristics*, 17:487–525, 2011.

M.J. Hirsch, C.N. Meneses, P.M. Pardalos, and M.G.C. Resende. Global optimization by continuous grasp. *Optimization Letters*, 1:201–212, 2007.

M.J. Hirsch, P.M. Pardalos, and M.G.C. Resende. Speeding up continuous GRASP. *Journal of Operational Research*, 205:507–521, 2010.

M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

P. McGuire. *Getting Started with Pyparsing*. O'Reilly Media, Sebastopol, CA, 2007.

R.M.A. Silva, M.G.C. Resende, P.M. Pardalos, and J. F. Gonçalves. Biased random-key genetic algorithm for bound-constrained global optimization. In D. Aloise, P. Hansen, and C. Rocha, editors, *Proceedings of the Global Optimization Workshop 2012*, pages 133–136, 2012.

W.M. Spears and K.A. DeJong. On the virtues of parameterized uniform crossover. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 230–236, 1991.

SymPy, 2011. URL `http://sympy.org/`. Last visited on July 11, 2011.

R.F. Toso and M.G.C. Resende. A c++ application programming interface for biased random-key genetic algorithms. Technical report, Algorithms and Optimization Research Department, AT&T Labs Research, 2012.

G. van Rossum and F.L. Drake Jr., editors. *Python/C API Reference Manual, Release 2.7*. Python Software Foundation, Wolfeboro Falls, NH, 2010a.

G. van Rossum and F.L. Drake Jr., editors. *Extending and embedding Python, Release 2.7*. Python Software Foundation, Wolfeboro Falls, NH, 2010b.

(Ricardo M. A. Silva) Centro de Informática (CIn), Universidade Federal de Pernambuco, Av. Prof. Luís Freire s/n, Cidade Universitária, Recife, PE, Brazil.
E-mail address: rmas@cin.ufpe.br

(Mauricio G. C. Resende) Algorithms and Optimization Research Department, AT&T Labs Research, 180 Park Avenue, Room C241, Florham Park, NJ 07932 USA.
E-mail address: mgcr@research.att.com

(Panos M. Pardalos) Department of Industrial and Systems Engineering, University of Florida, 303 Weil Hall, Gainesville, FL, 32611, USA.
E-mail address: pardalos@ufl.edu