A PYTHON/C LIBRARY FOR BOUND-CONSTRAINED GLOBAL OPTIMIZATION WITH CONTINUOUS GRASP

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

Abstract. This paper describes `libcgrpp`, a GNU-style dynamic shared Python/C library of the continuous greedy randomized adaptive search procedure (C-GRASP) for bound constrained global optimization. C-GRASP is an extension of the GRASP metaheuristic (Feo and Resende, 1989). After a brief introduction to C-GRASP, 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*. Without loss of generality, we take the domain S to be the 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 $\ell, x, u \in \mathbb{R}^n$.

C-GRASP is a stochastic local search method for finding cost-effective solutions to continuous global optimization problems subject to box constraints (Hirsch, 2006; Hirsch et al., 2007; 2010). Several improvements to C-GRASP were proposed in Hirsch et al. (2010) with the objective of speeding up implementations of the original algorithm (Hirsch et al., 2007) and increasing robustness, while at the same time keeping the overall algorithm simple to implement.

In this paper, we describe the C-GRASP library `libcgrpp`, a GNU-style dynamic shared Python/C library of the heuristic proposed in Hirsch et al. (2010) with some extensions. The library was developed using the `autoconf`, `automake`, and `libtool` packages (Calcote, 2010).

`libcgrpp` was 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 (van Rossum and Drake Jr., 2010c), any non-standard module or library, such as `SymPy` (SymPy, 2011),

Date: July 12, 2011.

Key words and phrases. GRASP, continuous GRASP, Global optimization, multimodal functions, continuous optimization, heuristic, stochastic algorithm, stochastic local search, nonlinear programming.

Cite as: R.M.A. Silva, M.G.C. Resende, P.M. Pardalos, and M.J. Hirsch, "A Python/C library for bound-constrained global optimization with continuous GRASP," AT&T Labs Research Technical Report, Florham Park, NJ 07932 USA, 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 C-GRASP heuristic is reviewed in Section 2. Section 3 shows how to download, install, configure, and use `libcgrpp`. An illustrative example is given in Section 4. Concluding remarks are made in Section 5.

2. Continuous GRASP

Continuous GRASP, or simply C-GRASP, is a method for finding good quality solutions to bound-constrained global optimization problems. A pseudo-code for C-GRASP is shown in Figure 1. The procedure takes as input the problem dimension n, lower and upper bound vectors ℓ and u, the objective function $f(\cdot)$, as well as the parameters h_s, h_e, and ρ_{lo}. As described in Hirsch et al. (2010), parameters h_s and h_e define, respectively, the initial and final grid discretization densities, while parameter ρ_{lo} specifies the portion of the neighborhood of the current solution that is searched during the local improvement phase.

Line 1 of the pseudo-code initializes the objective function value f^* of the best solution found to infinity. Since C-GRASP is a multi-start procedure, it is continued indefinitely, until one or more stopping criteria are satisfied in line 2. These stopping criteria could be based, for example, on the total number of function evaluations, the number of major iterations, or a target solution quality. The current implementation of our library can use one of two stopping criteria: (1) stop when the number of outer iterations (loop from line 2 to line 18 in the pseudocode) reaches

```
procedure C-GRASP(n, ℓ, u, f(·), h_s, h_e, ρ_lo)
1     f* ← ∞;
2     while stopping criteria not met do
3           x ← UnifRand(ℓ, u);
4           h ← h_s;
5           while h ≥ h_e do
6                 Impr_C ← false;
7                 Impr_L ← false;
8                 [x, Impr_C] ← ConstructGreedyRandomized(x, f(·), n, h, ℓ, u, Impr_C);
9                 [x, Impr_L] ← LocalImprovement(x, f(·), n, h, ℓ, u, ρ_lo, Impr_L);
10                if f(x) < f* then
11                      x* ← x;
12                      f* ← f(x);
13                end if
14                if Impr_C = false and Impr_L = false then
15                      h ← h/2;     /* make grid more dense */
16                end if
17          end while
18    end while
19    return(x*);
end C-GRASP;
```

Figure 1. Pseudo-code for C-GRASP.

a specified value; (2) stop when the optimality gap

$$GAP = |f(x) - f(x^*)| \leq \begin{cases} \epsilon & \text{if } f(x^*) = 0 \\ \epsilon \cdot |f(x^*)| & \text{if } f(x^*) \neq 0, \end{cases} \tag{1}$$

where x is the current best solution found by the heuristic and x^* is a known global minimum solution.

Each time the stopping criteria of line 2 are not satisfied, another iteration takes place (lines 3–17). During each iteration, the initial solution x is set, in line 3, to a random point distributed uniformly over the n-dimensional box defined by ℓ and u. Parameter h, which controls the discretization density of the search space, is re-initialized to h_s in line 4. The construction and local improvement phases are then called sequentially in lines 8 and 9, respectively. The solution returned from the local improvement procedure is compared against the current best solution in line 10. If the returned solution has a better objective value than the current best solution, then in lines 11–12 the current best solution is updated with the returned solution. In line 14, if variables $\texttt{Impr}_\texttt{C}$ and $\texttt{Impr}_\texttt{L}$ are `false`, then the grid density is increased by halving h, in line 15. The variable $\texttt{Impr}_\texttt{C}$ (resp. $\texttt{Impr}_\texttt{L}$) is `false` upon return from the construction (resp. local improvement) procedure if and only if no improvement is made in the construction (resp. local improvement) procedure. The grid density is increased at this stage because repeating the construction procedure with the same grid density will not improve the solution. This allows C-GRASP to start with a coarse discretization and adaptively increase the density as needed, thereby intensifying the search with a more dense discretization when a good solution has been found. The best solution found, at the time the stopping criteria are satisfied, is returned.

The *construction procedure* is shown in Figure 2. It takes as input a solution vector x. Initially, the procedure allows all coordinates of x to change (i.e. they are called *unfixed*). In turn, in line 10 of the pseudo-code, if `ReUse` is `false`, a line search is performed in each unfixed coordinate direction i of x with the other $n-1$ coordinates of x held at their current values. In lines 10 and 11 of the pseudo-code, the value z_i, for the i-th coordinate, that minimizes the objective function, together with the objective function value g_i, are saved. In line 11, $\breve{x}^i$ denotes x with the i-th coordinate set to z_i.

After looping through all unfixed coordinates (lines 7 to 16), in lines 17 to 23 a restricted candidate list (RCL) is formed containing the unfixed coordinates i whose g_i values are less than or equal to $\underline{g} + \alpha \cdot (\bar{g} - \underline{g})$, where $\bar{g}$ and $\underline{g}$ are, respectively, the maximum and minimum g_i values over all unfixed coordinates of x, and $\alpha \in [0, 1]$ is chosen uniformly at random in line 2. In line 24, a coordinate is chosen at random from the `RCL`, say coordinate $j \in$ `RCL`. Line 25 checks whether x_j and z_j are equal. If so, line 26 sets `ReUse` to the value `true`. Otherwise, in lines 28 to 30, `ReUse` is set to `false`, $\texttt{Impr}_\texttt{C}$ is set to `true`, and x_j is set to equal z_j. Finally, in line 32, the coordinate j of x is fixed, by removing j from the set `UnFixed`. The above procedure is continued until all of the n coordinates of x have been fixed. At that stage, x and $\texttt{Impr}_\texttt{C}$ are returned from the construction procedure.

From a given input point $x \in \mathbb{R}^n$, the *local improvement procedure* 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

```
procedure ConstructGreedyRandomized(x, f(·), n, h, ℓ, u, Imprc)
1     UnFixed ← {1, 2, ..., n};
2     α ← UnifRand(0, 1);
3     ReUse ← false;
4     while UnFixed ≠ ∅ do
5           g ← +∞;
6           ḡ ← −∞;
7           for i = 1, ..., n do
8                 if i ∈ UnFixed then
9                       if ReUse = false then
10                            z_i ← LineSearch(x, h, i, n, f(·), ℓ, u);
11                            g_i ← f(x̄^i);
12                      end if
13                      if g > g_i then g ← g_i;
14                      if ḡ < g_i then ḡ ← g_i;
15                end if
16          end for
17          RCL ← ∅;
18          Threshold ← g + α · (ḡ − g);
19          for i = 1, ..., n do
20                if i ∈ UnFixed and g_i ≤ Threshold then
21                      RCL ← RCL ∪ {i};
22                end if
23          end for
24          j ← RandomlySelectElement(RCL);
25          if x_j = z_j then
26                ReUse ← true;
27          else
28                x_j ← z_j;
29                ReUse ← false;
30                Imprc ← true;
31          end if
32          UnFixed ← UnFixed \ {j};     /* Fix coordinate j. */
33    end while
34    return(x,Imprc);
end ConstructGreedyRandomized;
```

FIGURE 2. Pseudo-code for C-GRASP construction phase.

current solution and h be the current grid discretization parameter. Define

$$S_h(\bar{x}) = \{x \in S \mid \ell \leq x \leq u,\ x = \bar{x} + \tau \cdot h,\ \tau \in \mathbb{Z}^n\}$$

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 $B_h(\bar{x})$.

A pseudo-code for the local improvement procedure is given in Figure 3. The procedure starts from a solution $x \in S \subseteq \mathbb{R}^n$ found in the construction procedure. The current best local improvement solution x^* is initialized to x in line 1 of the pseudo-code and the current best solution f^* of the local improvement phase is initialized in line 2 as $f(x^*)$. Based on the current value of the discretization parameter h and the number of points in $B_h(x^*)$, the number of grid points is

```
procedure LocalImprovement(x, f(·), n, h, ℓ, u, ρ_lo, Impr_L, MaxPointsToExamine)
1     x* ← x;
2     f* ← f(x);
3     NumGridPoints ← ∏_{i=1}^{n} ⌈(u_i − ℓ_i)/h⌉;
4     PointsToExamine ← ⌈ρ_lo · NumGridPoints⌉;
5     if PointsToExamine > MaxPointsToExamine then
6           PointsToExamine ← MaxPointsToExamine;
7     end if
8     NumPointsExamined ← 0;
9     while NumPointsExamined ≤ PointsToExamine do
10          NumPointsExamined ← NumPointsExamined + 1;
11          x ← RandomlySelectElement(B_h(x*));
12          if ℓ ≤ x ≤ u and f(x) < f* then
13                x* ← x;
14                f* ← f(x);
15                Impr_L ← true;
16                NumPointsExamined ← 0;
17          end if
18    end while
19    return(x*, Impr_L);
end LocalImprovement;
```

FIGURE 3. Pseudo-code for C-GRASP local improvement phase.

computed in line 3. In line 4, the number of points `PointsToExamine` that can be evaluated is computed by using parameter ρ_{lo}, the portion of the neighborhood to be examined. However, different from what is described in Hirsch et al. (2010), in lines 5 to 7 we impose the restriction that the maximum number of points that can be evaluated in any neighborhood be limited to the value of the parameter `MaxPointsToExamine`.

Starting at the point x^*, in the loop in lines 9–18 the algorithm randomly selects `PointsToExamine` points in $B_h(x^*)$, one at a time. In line 12, if the current point x selected from $B_h(x^*)$ is feasible and is better than x^*, then x^* is set to x, f^* is set to $f(x)$, $\texttt{Impr}_\texttt{L}$ is set to `true`, `NumPointsExamined` is reset to zero, and the process restarts with x^* as the starting solution. $\texttt{Impr}_\texttt{L}$ is used to determine whether the local improvement procedure improved the best solution. Local improvement is terminated if an *h-local minimum* solution x^* is found. At that point, x^* and $\texttt{Impr}_\texttt{L}$ are returned from the local improvement procedure.

3. THE LIBCGRPP LIBRARY

This section begins by showing how to download (Section 3.2), build (Section 3.3), and install (Section 3.4) the `libcgrpp` library, as well as its package dependencies (Section 3.1). Then, the format of components required to use the library are presented as follows: function module (Section 3.5), parameter input file (Section 3.6), and calling C program (Section 3.7). Finally, the format of the output file is described in Section 3.8.

3.1. Dependencies.

The `libcgrpp` 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/`.

3.2. **Downloads.** Full distribution of the `libcgrpp` library is available at `http://www2.research.att.com/~mgcr/src/cgrasp`. The package is distributed as the tar file `cgraspp-0.0.1.tar.gz` containing the following directory structure:

```
.:
AUTHORS     configure     INSTALL      NEWS    src
ChangeLog   COPYING       Makefile     README  THANKS
./src:
cgrasparser.py  cgrasp.h     mt19937ar.c  simclist.c
cgrasp.c        Makefile     mt19937ar.h  simclist.h
```

Each file of this directory is described in Table 1.

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

files	**description**
`cgrasp.c`	Embedded Python-in-C code of `C-GRASP` (Fig. 1)
`cgrasp.h`	C header file of `cgrasp.c`
`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`
`simclist.c`	SimCList (SimCList, 2011) library for handling lists
`simclist.h`	C header file of `simclist.c`
`cgrasparser.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

3.3. **Building.** The `libcgrpp` 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 `cgraspp-0.0.1.tar.gz` source tarball:
```
$ tar -xvf cgraspp-0.0.1.tar.gz
```
(2) Run the configure script to create the Makefiles:
```
$ cd cgraspp-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.

3.4. **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:

  `libcgrpp.a  libcgrpp.la  libcgrpp.so`
  `libcgrpp.so.0  libcgrpp.so.0.0.0`
- `/usr/local/include`, the header files:

  `cgrasp.h  mt19937ar.h  simclist.h`
- `/usr/local/lib/python2.7/site-packages/cgraspp`, the Python script:

  `cgrasparser.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.

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

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

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 function, the 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`).

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

- `-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;
- `-ep <d>`: sets parameter ϵ of Equation (1) of Section 2 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>`;
- `-he <d>`: sets the ending grid discretization density h_e to the positive real number `<d>`;

- `-ro <d>`: sets the local improvement parameter ρ_{lo} to the positive real number `<d>` such that $0 <$ `<d>` ≤ 1;
- `-ls <n>`: turns local improvement procedure on (off) if `<n>` is equal to 1 (0);
- `-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:

```
-hs 0.5 -he 0.0001 -ro 0.01 -ls 1 -mp 100 -of output.file
-sd 270001 -md ackley -ft f -ds 5 -ov 0 -ep 0.001
-dm -10 10 1 -5 3 4:5 -13 7
```

This input file specifies that C-GRASP 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 2) will be such that $GAP = |A_5(x') - 0| \leq \epsilon = 0.001$, using the following parameters: $h_s = 0.5$, $h_e = 0.0001$, $\rho_{lo} = 0.01$, $seed = 270001$, and `MaxPointsToExamine` $= 100$.

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

3.7. **Using the library in C.** To use the function `double cgrasp(int, **char)` of the `libcgrpp` library in a C program (which we shall call here `userprog.c`):

(1) Put `#include <cgrasp.h>` in the source code of the C program `userprog.c`:

```
#include <cgrasp.h>
...
double x;
...
void main(int argc, char **argv){
  ...
  x = cgrasp(argc,argv);
  ...
}
```

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

```
$ gcc -I<pathname> userprog.c -o userprog -lcgrpp
```

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/cgraspp
```

sets `PYTHONPATH` to the current directory `$PWD`, to specify the directory of the function module and `/usr/local/lib/python2.7/site-packages/cgraspp` to specify the directory of the C-GRASP input parser `cgrasparser`.

Finally, to run the program, type:

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

as for example:

```
$ ./userprog input
```

3.8. **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) Summary of the execution, including information about the instance itself as well as the execution parameters;
  (2) For each objective function improvement, a line is printed with the following format:

  ```
  <responsible>:
  <keyword> <value>
  ```

  The procedure responsible for the improvement can be one of the following:
  - `random`: Procedure `UnifRand` in line 3 of Algorithm 1;
  - `construction`: Algorithm 2;
  - `local search`: Algorithm 3.

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

  (3) Total CPU time (in seconds) in the following format:
  `time: <value>`
  For example:
  `time: 165.650009`
  (4) Total function evaluations in the following format:
  `evaluations: <value>`

For example:
`evaluations: 1566509`

(5) Value of the overall best solution found in the following format:
`optimum: <value>`
For example:
`optimum: 0.000000`

Consider as an example the output generated by Algorithm 1 to find a solution $x \in [-10, 10]^2$, such that the **Booth** function:

$$BO(x) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2 \leq \epsilon = 0.001 \tag{3}$$

using the following parameters: $h_s = 0.5$, $h_e = 0.0001$, $\rho_{lo} = 0.01$, $seed = 270001$, and `MaxPointsToExamine` $= 100$.

```
random:
time: 0.000000
evaluations: 1
best value: 127.067622
solution: 6.046783 -5.067851

construction:
time: 0.000000
evaluations: 80
best value: 77.292449
solution: 7.546783 -2.067851

local search:
time: 0.010000
evaluations: 1126
best value: 0.103908
solution: 1.042637 2.824020

construction:
time: 0.010000
evaluations: 2367
best value: 0.061733
solution: 1.042637 3.074020

local search:
time: 0.020000
evaluations: 3423
best value: 0.005461
solution: 0.954859 3.017175

construction:
time: 0.030000
evaluations: 6546
best value: 0.005367
solution: 1.017359 3.017175

local search:
time: 0.030000
evaluations: 7553
```

```
best value: 0.000675
solution: 1.005291 3.006945

construction:
time: 0.040000
evaluations: 14036
best value: 0.000201
solution: 0.989666 3.006945

local search:
time: 0.040000
evaluations: 15049
best value: 0.000095
solution: 1.000248 2.995449

time: 0.040000
dimension: 2
epsilon: 0.000100
seed: 270001
h_s: 0.500000
h_e: 0.000100
ro: 10.000000
LS option: 1
LS max points: 1000.000000
output file: output.file
```

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

4. 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 <cgrasp.h>
double main(int argc, char **argv){
  double res;
  res = cgrasp(argc,argv);
  return res;
}
```

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

(2) Implement the function to be minimized as a Python module. For example, the Booth function described in Equation (3) 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:

```
-hs 0.5 -he 0.0001 -ro 0.01 -ls 1 -mp 100 -of output.file -sd 270002
-md booth -ft g -ds 2 -dm -10 10 -ov 0 -ep 0.001
```

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.c`:

```
$ gcc -I/usr/local/include/python2.7 program.c -o program -lcgrpp
```

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/cgraspp
```

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

```
$ ./program input
```

which will generate the following output:

```
random:
time: 0.000000
evaluations: 1
best value: 346.236119
solution: 9.866860 2.305230

construction:
time: 0.000000
evaluations: 81
best value: 0.290981
solution: 1.366860 2.805230

local search:
time: 0.000000
evaluations: 184
best value: 0.143849
solution: 1.013306 3.158784

construction:
time: 0.000000
evaluations: 525
best value: 0.105516
solution: 0.763306 3.158784

local search:
time: 0.000000
evaluations: 644
best value: 0.005744
solution: 0.948869 3.026495

local search:
time: 0.010000
evaluations: 1396
```

```
best value: 0.004045
solution: 0.969845 3.046070

local search:
time: 0.010000
evaluations: 2570
best value: 0.000127
solution: 1.002841 2.992989

time: 0.010000
dimension: 2
epsilon: 0.001000
seed: 270002
h_s: 0.500000
h_e: 0.000100
ro: 0.010000
LS option: 1
LS max points: 100.000000
output file: output.file
```

Suppose that you decide to change the function to be optimized. For example, instead of Booth function, consider the Ackley function described in Equation (2). The necessary steps to incorporate this new function are as follows:

(1) Implement the Ackley function in Python as described in Section 3.5 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:

```
-hs 0.5 -he 0.0001 -ro 0.01 -ls 1 -mp 100 -of output
-sd 270001 -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:

```
h=0.500000, h_e=0.000100
iteration: 0
random:
time: 0.000000
evaluations: 1
best value: 20.933162
solution: 21.105262 -3.902664  3.711931   3.212137  20.324969
24.701692 21.233718 25.782630 27.744391 -14.956639  -6.553142
23.285696 -5.779033 18.354341  8.369533  15.338570  17.463161
19.988780  3.374752 -4.781582 12.275036  12.074889 -10.080807
22.278906 22.198215  4.374315  2.281934  25.520283  10.808203
 2.876172
```

```
h=0.500000, h_e=0.000100
iteration: 0
construction:
time: 0.080000
evaluations: 2672
best value: 1.695881
solution:  0.105262  0.097336  0.211931  0.212137 -0.175031
 0.201692  0.233718 -0.217370  0.244391  0.043361 -0.053142
-0.214304  0.220967 -0.145659 -0.130467 -0.161430 -0.036839
-0.011220 -0.125248  0.218418 -0.224964  0.074889 -0.080807
-0.221094  0.198215 -0.125685 -0.218066  0.020283 -0.191797
-0.123828

h=0.500000, h_e=0.000100
iteration: 0
local search:
time: 0.080000
evaluations: 2861
best value: 1.441325
solution:  0.426598  0.136822  0.151578  0.050235 -0.204186
 0.155583  0.085362 -0.089813  0.200006  0.165673 -0.068200
 0.025389  0.278641  0.043196  0.124905 -0.008284  0.035707
 0.189945 -0.004520  0.332007  0.073254  0.213014  0.124168
 0.002367 -0.055073  0.084247 -0.090377  0.030453  0.113455
 0.043277

...

h=0.000977, h_e=0.000100
iteration: 0
construction:
time: 154.340012
evaluations: 6992912
best value: 0.001055
solution: -0.000427  0.000313 -0.000397  0.000071 -0.000374
 0.000042  0.000050 -0.000464 -0.000361 -0.000465  0.000010
 0.000233 -0.000099 -0.000214 -0.000161  0.000323  0.000199
-0.000482  0.000022  0.000400 -0.000134  0.000119 -0.000005
-0.000110  0.000052 -0.000059  0.000316 -0.000127  0.000124
 0.000262


h=0.000977, h_e=0.000100
iteration: 0
local search:
time: 154.340012
evaluations: 6993060
best value: 0.000971
solution: -0.000443  0.000323  0.000021  0.000286 -0.000003
 0.000094  0.000271 -0.000211  0.000333 -0.000158  0.000357
 0.000406 -0.000194 -0.000120 -0.000148 -0.000023  0.000243
-0.000415 -0.000305  0.000223 -0.000179  0.000060  0.000207
```

```
-0.000031 -0.000045 -0.000202  0.000436  0.000166  0.000166
-0.000020

time: 154.340012
dimension: 30
epsilon: 0.001000
seed: 270001
h_s: 0.500000
h_e: 0.000100
ro: 0.010000
LS option: 1
LS max points: 100.000000
output file: output.file
```

5. Concluding remarks

In this paper, we describe how to download, install, configure, and use an implementation of the C-GRASP heuristic for bound constrained global optimization introduced by Hirsch et al. (2007; 2010). Since C-GRASP makes no use of derivative nor *a priori* information, it is a well-suited approach for solving general global optimization problems.

The C-GRASP library `libcgrpp` was implemented in C, and on the runs done for this paper it was compiled with the `gcc` 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`. To handle lists, we used the `SimCList` library, available at `http://mij.oltrelinux.com/devel/simclist`.

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 done while he was a post-doc scholar at AT&T Labs Research in Florham Park, New Jersey, and 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). The research of P.M. Pardalos was partially supported by grants from the United States Air Force and The Defense Threat Reduction Agency (DTRA) of the United States Department of Defense.

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

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.J. Hirsch. *GRASP-based heuristics for continuous g lobal optimization problems*. PhD thesis, University of Florida, Gainesville, FL, 2006.

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.

SimCList, 2011. URL `http://mij.oltrelinux.com/devel/simclist/`. Last visited on July 12, 2011.

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

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.

G. van Rossum and F.L. Drake Jr., editors. *The Python Library Reference, Release 2.7*. Python Software Foundation, Wolfeboro Falls, NH, 2010c.

(Ricardo M. A. Silva) Centro de Informática (CIn), Federal University of 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`

(Micahel J. Hirsch) Raytheon Company, Intelligence and Information Systems, 300 Sentinel Drive, Annapolis Junction, MD, 20701, USA.
E-mail address: `mjh8787@ufl.edu`