To appear in *Optimization Software Class Libraries*, Stefan Voss and David L. Woodruff (eds.), Kluwer Academic Publishers, Boston.

Chapter #

The OptQuest Callable Library

MANUEL LAGUNA[1] and RAFAEL MARTÍ
University of Colorado at Boulder and Universitat de Valencia

Key words: metaheuristic optimization, evolutionary methods, scatter search.

Abstract: In this chapter we discuss the development and application of a library of functions that is the optimization engine for the OptQuest system. OptQuest is commercial software designed for optimizing complex systems, such as those formulated as simulation models. OptQuest has been integrated with several simulation packages with the goal of adding optimization capabilities. The optimization technology within OptQuest is based on the metaheuristic framework known as scatter search. In addition to describing the functionally of the OptQuest Callable Library (OCL) with an illustrative example, we apply it to a set of unconstrained nonlinear optimization problems.

1. INTRODUCTION

The OptQuest Callable Library (OCL) is the optimization engine of the OptQuest system[2]. The goal of OptQuest is to optimize complex systems, which are those that cannot be easily formulated as mathematical models and solved with classical optimization tools. Many real world optimization problems in business, engineering and science are too complex to be given tractable mathematical formulations. Multiple nonlinearities, combinatorial relationships and uncertainties often render challenging practical problems inaccessible to modeling except by resorting to more comprehensive tools (like computer simulation). Classical optimization methods encounter grave

[1] Partially supported by the visiting professor fellowship program of the University of Valencia (Grant Ref. No. 42743).
[2] OptQuest is a registered trademark of OptTek Systems, Inc. (www.opttek.com).

difficulties when dealing with the optimization problems that arise in the context of complex systems. In some instances, recourse has been made to itemizing a series of scenarios in the hope that at least one will give an acceptable solution. Due to the limitations of this approach, a long-standing research goal has been to create a way to guide a series of complex evaluations to produce high quality solutions, in the absence of tractable mathematical structures. (Note that in the context of optimizing simulations, a "complex evaluation" refers to the execution of a simulation model.)

Theoretically, the issue of identifying best values for a set of decision variables falls within the realm of optimization. Until quite recently, however, the methods available for finding optimal decisions have been unable to cope with the complexities and uncertainties posed by many real world problems of the form treated by simulation. The area of stochastic optimization has attempted to deal with some of these practical problems, but the modeling framework limits the range of problems that can be tackled with such technology.

The complexities and uncertainties in complex systems are the primary reason that simulation is often chosen as a basis for handling the decision problems associated with those systems. Consequently, decision makers must deal with the dilemma that many important types of real world optimization problems can only be treated by the use of simulation models, but once these problems are submitted to simulation there are no optimization methods that can adequately cope with them.

Recent developments are changing this picture. Advances in the field of metaheuristics—the domain of optimization that augments traditional mathematics with artificial intelligence and methods based on analogs to physical, biological or evolutionary processes—have led to the creation of optimization engines that successfully guide a series of complex evaluations with the goal of finding optimal values for the decision variables. One of those engines is the search algorithm embedded in OCL.

OCL is designed to search for optimal solutions to the following class of optimization problems:

$$\begin{array}{lll}
\text{Max or Min} & F(x) & \\
 & & \\
\text{Subject to} & Ax \leq b & \text{(Constraints)} \\
 & g_l \leq G(x) \leq g_u & \text{(Requirements)} \\
 & l \leq x \leq u & \text{(Bounds)}
\end{array}$$

where x can be continuous or discrete with an arbitrary step size.

The objective F(x) may be any mapping from a set of values x to a real value. The set of constraints must be linear and the coefficient matrix "A" and the right-hand-side values "b" must be known. The requirements are simple upper and/or lower bounds imposed on a function that can be linear or non-linear. The values of the bounds "g_l" and "g_u" must be known constants. All the variables must be bounded and some may be restricted to be discrete with an arbitrary step size.

In a general-purpose optimizer such as OCL, it is preferable to separate the solution procedure from the complex system to be optimized. A potential disadvantage of this "black box" approach is that the optimization procedure is generic and does not know anything about what goes on inside of the box and therefore does not use any problem-specific information (Figure 1). The clear advantage, on the other hand, is that the same optimizer can be used for many complex systems.

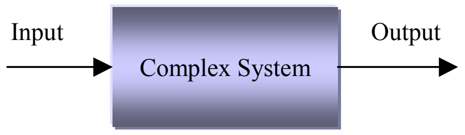

Figure 1. Complex system as a black box

OCL is a generic optimizer that overcomes the deficiency of black box systems of the type illustrated in Figure 1, and successfully embodies the principle of separating the method from the model. In such a context, the optimization problem is defined outside the complex system. Therefore, the evaluator can change and evolve to incorporate additional elements of the complex system, while the optimization routines remain the same. Hence, there is a complete separation between the model that represents the system and the procedure that is used to solve optimization problems defined within this model.

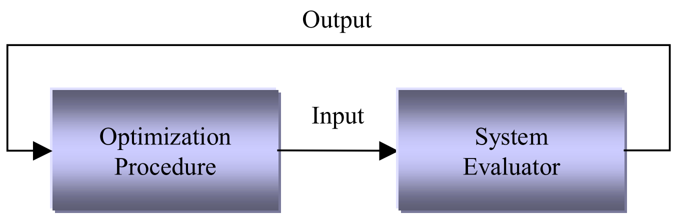

Figure 2. Coordination between optimization and system evaluation

The optimization procedure uses the outputs from the system evaluator, which measures the merit of the inputs that were fed into the model. On the basis of both current and past evaluations, the optimization procedure decides upon a new set of input values (see Figure 2). The optimization procedure is designed to carry out a special "non-monotonic search," where the successively generated inputs produce varying evaluations, not all of them improving, but which over time provide a highly efficient trajectory to the best solutions. The process continues until an appropriate termination criterion is satisfied (usually based on the user's preference for the amount of time to be devoted to the search).

OCL allows the user to build applications to solve problems using the "black-box" approach for evaluating an objective function and a set of requirements. Figure 3 shows a conceptualization of how OCL can be used to search for optimal solutions to complex optimization problems.

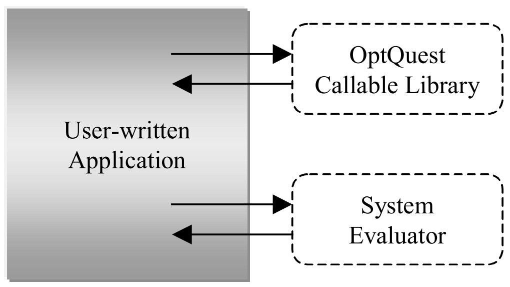

Figure 3. OCL linked to a user-written application

Figure 3 assumes that the user has a system evaluator that given a set of input values, it returns a set of output values that can used to guide a search. For example, the evaluator may have the form of a computer simulation that, given the values of a set of decision variables, it returns the value of one or more performance measures (that define the objective function and possibly a set of requirements). The user-written application uses OCL functions to define an optimization problem and launch a search for the optimal values of the decision variables.

2. SCATTER SEARCH

The optimization technology embedded in OCL is the metaheuristic known as *scatter search*. Scatter search has some interesting commonalties with *genetic algorithms* (GA), although it also has a number of quite distinct features. Several of these features have come to be incorporated into GA approaches after an intervening period of approximately a decade, while others remain largely unexplored in the GA context.

Scatter search is designed to operate on a set of points, called *reference points,* which constitute good solutions obtained from previous solution efforts. Notably, the basis for defining “good” includes special criteria such as diversity that purposefully go beyond the objective function value. The approach systematically generates combinations of the reference points to create new points, each of which is mapped into an associated feasible point. The combinations are generalized forms of linear combinations, accompanied by processes to adaptively enforce constraint-feasibility and encourage requirement-feasibility.

The scatter search process is organized to (1) capture information not contained separately in the original points, (2) take advantage of auxiliary heuristic solution methods (to evaluate the combinations produced and to actively generate new points), and (3) make dedicated use of strategy instead of randomization to carry out component steps.

```
1.  Apply a diversification generation method to build a starting set
    of solutions.  Designate a subset of the best points (judged by
    quality and diversity) to be reference points.
while (stopping criteria are not satisfied) {
    2.  Form combinations of subsets of the current reference
        points to create new points.  The combinations are (a)
        chosen to produce points both inside and outside the
        convex region spanned by the reference points, and (b)
        modified by generalized mapping processes to yield
        feasible points according to the constraints in the problem
        (both linear and integrality constraints).
    3.  Update the reference set by selecting points that can
        improve the quality and/or diversity of the set.
    if (no new combinations can be explored in the current set) {
        4.  Extract a collection of the best points in the current
            reference set to be used as starting points for a new
            application of the diversification generation method.
    }
}
```

Figure 4. Scatter search outline

Figure 4 sketches the scatter search approach in its original form. Extensions can be created to take advantage of memory-based designs typical of tabu search. Two particular features of the scatter search proposal deserve mention. The use of clustering strategies has been suggested for selecting subsets of points in step 2, which allows different blends of intensification and diversification by generating new points "within clusters" and "across clusters." Also, the solutions generated by the combination method in step 2 are often subjected to an improvement method, which typically consists of a local search procedure. The improvement method is capable of starting from a feasible or an infeasible solution created by the combination method.

It is interesting to observe similarities and contrasts between scatter search and the original GA proposals. Both are instances of what are sometimes called "population based" approaches. Both incorporate the idea that a key aspect of producing new elements is to generate some form of combination of existing elements. On the other hand, GA approaches are predicated on the idea of choosing parents randomly to produce offspring, and further on introducing randomization to determine which components of the parents should be combined. By contrast, the scatter search approach does not emphasize randomization, particularly in the sense of being indifferent to choices among alternatives. Instead, the approach is designed to incorporate strategic responses, both deterministic and probabilistic, that take account of evaluations and history. Scatter search focuses on generating relevant outcomes without losing the ability to produce diverse solutions, due to the way the generation process is implemented. For example, the approach includes the generation of new points that are not convex combinations of the original points. The new points constitute forms of *extrapolations*, endowing them with the ability to contain information that is not contained in the original reference points.

Scatter search is an *information-driven* approach, exploiting knowledge derived from the search space, high-quality solutions found within the space, and trajectories through the space over time. The incorporation of such designs is responsible for enabling OCL to efficiently search the solution space of optimization problems in complex systems.

3. THE OCL OPTIMIZER

OCL seeks to find an optimal solution to a problem defined on a vector x of bounded variables. The scatter search method implemented in OCL begins by generating a starting set of diverse points. This is accomplished by dividing the range of each variable into 4 sub-ranges of equal size. Then, a solution is constructed in two steps. First, a sub-range is randomly selected. The probability of selecting a sub-range is inversely proportional to its frequency count (which keeps track of the number of times the subrange has been selected). Second, a value is randomly chosen from the selected sub-range. The starting set of points also includes the following solutions:

- All variables are set to the lower bound
- All variables are set to the upper bound
- All variables are set to the midpoint $x = l + (u - l)/2$
- Other solutions suggested by the user

A subset of diverse points is chosen as members of the reference set. A set of points is considered diverse if its elements are “significantly” different from one another. OCL uses a Euclidean distance measure to determine how “close” a potential new point is from the points already in the reference set, in order to decide whether the point is included or discarded.

When the optimization model includes discrete variables, a simple rounding procedure is used to map fractional values to discrete values. When the model includes linear constraints newly created points are subjected to a feasibility test before they are sent to the evaluator (i.e., before the objective function value F(x) and the requirements G(x) are evaluated). Note that the evaluation of the objective function may entail the execution of a simulation, and therefore it is important to be sure to evaluate only those solutions that are feasible with respect to the set of constraints. For ease of notation, we represent the set of constraints as $Ax \leq b$, although equality constraints are also allowed. The feasibility test consists of checking (one by one) whether the linear constraints are satisfied. If the solution is infeasible with respect to one or more constraints, OCL formulates and solves a linear programming (LP) problem. The LP (or mixed-integer program, when x contains discrete variables) has the goal of finding a feasible solution x^* that minimizes the absolute deviation between x and x^*. Mathematically, the problem can be formulated as:

$$\begin{array}{ll} \text{Minimize} & d^- + d^+ \\ & \\ \text{subject to} & Ax^* \leq b \\ & x - x^* - d^- + d^+ = 0 \\ & l \leq x^* \leq u \\ & d^-,\ d^+ \geq 0 \end{array}$$

where d^- and d^+ are, respectively, negative and positive deviations of x^* from the infeasible point x. When constraints are not specified, infeasible points are made feasible by simply adjusting variable values to their closest bound and rounding when appropriately. That is, if $x > u$ then $x^* = u$ and if $x < l$ then $x^* = l$.

Once the reference set has been created, a combination method is applied to initiate the search for optimal solutions. The method consists of finding linear combinations of reference solutions. The combinations are based on the following three types, which assume that the reference solutions are x' and x'':

$$x = x' - d$$
$$x = x' + d$$
$$x = x'' - d$$

where $d = r\frac{x'' - x'}{2}$ and r is a random number in the range (0, 1). The number of solutions created from the linear combination of two reference solutions depends on the quality of the solutions being combined. Specifically, when the best two reference solutions are combined, they generate up to 5 new solutions, while when the worst two solutions are combined they generate only one.

In the course of searching for a global optimum, the combination method may not be able to generate solutions of enough quality to become members of the reference set. If the reference set does not change and all the combinations of solutions have been explored, a diversification step is triggered (see step 4 in Figure 4). This step consists of rebuilding the reference set to create a balance between solution quality and diversity. To preserve quality, a small set of the best (*elite*) solutions in the current reference set is used to seed the new reference set. The remaining solutions are eliminated from the reference set. Then, the diversification generation method is used to repopulate the reference set with solutions that are diverse with respect to the elite set. This reference set is used as the starting point for a new round of combinations.

3.1 Constraints Vs. Requirements

So far, we have assumed that the complex system to be optimized can be treated by OCL as a "black box" that takes x as an input to produce F(x) as an output. We have also assumed that for x to be feasible, the point must be within a given set of bounds and, when applicable, also satisfy a set of linear constraints. We assume that both the bounds and the coefficient matrix are known. However, there are situations where the feasibility of x is not known prior to performing the process that evaluates F(x), i.e., prior to executing the "black box" system evaluator. In other words, the feasibility test for x cannot be performed in the input side of the black box but instead has to be performed within the black box and communicated as one of the outputs. This situation is depicted in Figure 5.

This figure shows that when constraints are included in the optimization model, the evaluation process starts the mapping $x \to x^*$. If the only constraints in the model are integrality restrictions, the mapping is achieved with a simple rounding mechanism that transforms fractional values into integer values for the discrete variables. If the constraints are linear, then the mapping consists of formulating and solving the abovementioned linear programming problem. Finally, if the constraints are linear and the model also includes discrete variables, then the linear programming formulation becomes a mixed-integer programming problem that is solved accordingly. Obviously, if the optimization model has neither constraints nor discrete variables then $x^* = x$.

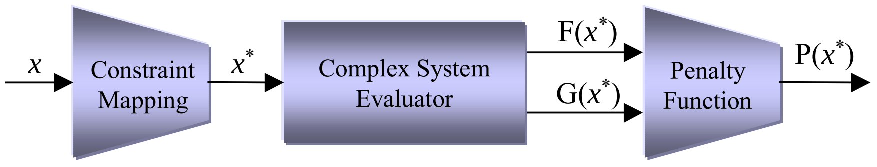

Figure 5. Solution evaluation

The mapped solution is processed through the complex system evaluator to obtain a set of performance measures. One of these measures is used as the *objective function* value F(x) and provides the means for the search to distinguish good from bad solutions. Other measures G(x) associated with the performance of the system can be used to define a set of *requirements*. A requirement is expressed as a bound on the value of a performance measure G(x). Thus, a requirement may be defined as an upper or a lower bound on an output of the complex system evaluator. Instead of discarding requirement-infeasible solutions, OCL handles them with a composite function P(x) that penalizes the requirement violations. The penalty is proportional to the degree of the violation and is not static throughout the search. OCL assumes that the user is interested in finding a requirement-feasible solution if one exists. Therefore, requirement-infeasible solutions are penalized more heavily when no requirement-feasible solution has been found during the search than when one is already available.

To illustrate the evaluation process in the context of a simulated system, consider an investment problem for which x represents the allocation of funds to a set of investment instruments. The objective is to maximize the expected return. Assume that a Monte Carlo simulation is performed to estimate the expected return F(x) for a given fund allocation. Hence, in this case, the complex system evaluator consists of a Monte Carlo simulator.

Restrictions on the fund allocations, which establish relationships among the x variables, are handled within the linear programming formulation that maps infeasible solutions into feasible ones. Thus, a restriction of the type “the combined investment in instruments 2 and 3 should not exceed the total investment in instrument 7,” results in the linear constraint $x_2 + x_3 \le x_7$. On the other hand, a restriction that limits the variability of the returns (as measured by the standard deviation) to be no more than a critical value c cannot be enforced in the input side of the Monte Carlo simulator. Clearly, the simulation must be executed first in order to estimate the variability of the returns. Suppose that the standard deviation of the returns is represented by G(x), then the requirement in this illustrative situation is expressed as G(x) $\le c$.

Note that the constraint-mapping mechanism within OCL does not handle nonlinear constraints. However, nonlinear constraints can be modeled as requirements and incorporated within the penalty function P(x). For example, suppose that an optimization model must include the following nonlinear constraint:

$$x_1x_2 - x_1^2 \le 120$$

Then, the system evaluator calculates, for a given solution x, the left-hand side of the nonlinear constraint and communicates the result as one of the outputs. OCL uses this output and compares it to the right-hand side value of 120 to determine the feasibility of the current solution. If the solution is not feasible a penalty term is added to the value of the objective function F(x).

4. OCL FUNCTIONALITY

The OptQuest Callable Library consists of a set of 23 functions that are classified into four categories:

General
Variables, Constraints and Requirements
Solutions
Parameter Setting

The functions are classified according to their purpose and the library is available for both C and Visual Basic applications. Table 1 shows the complete set of OCL functions. A problem can be formulated and optimized with as few as 5 functions, which are indicated with an asterisk in Table 1.

Table 1. OCL Functions

Category	Function	Brief Description
General	OCLGoodBye	Deletes a model and frees OCL's memory
	OCLInitPop*	Generates the starting set of reference solutions
	OCLSetup*	Allocates memory for an optimization model
Variables, Constraints and Requirements	OCLDefineVar*	Defines a decision variable
	OCLDefineReq	Defines a requirement
	OCLConsCoeff	Changes the value of a constraint coefficient
	OCLConsRhs	Changes the values of the right-hand-side of a constraint
	OCLConsType	Changes the type of constraint
Solutions	OCLGenerateAllSolutions	Generates all solutions to a pure discrete problem
	OCLGetBest	Retrieves the best solution currently in OCL's memory
	OCLGetPopSolution	Retrieves a solution from the current reference set
	OCLGetSolution*	Retrieves a trial solution for evaluation
	OCLPutPopSolution	Replaces one of the solutions that is currently in the reference set
	OCLPutSolution*	Places the evaluation of a trial solution in OCL's memory
	OCLSugPopSolution	Places a suggested solution in a temporary memory to be added to the reference set when the set is rebuilt
	OCLSugSolution	Places a suggested solution in OCL's database
Parameter Setting	OCLSetBoundFreq	Sets the frequency parameter for the boundary search strategy
	OCLSetCheckDup	Activates or deactivates the use of the database to check for duplicated solutions
	OCLSetObjPrecision	Sets the number of digits of precision for the objective function
	OCLSetVarPrecision	Sets the number of digits of precision for the decision variables
	OCLSetFileSolutions	Activates a log file of solutions
	OCLSetPopSize	Sets the size of the reference set
	OCLSetRandomSeed	Sets the seed for the random number generator
	OCLSetSolutions	Sets the targeted total number solutions generated during the search

* Required function.

Additional functions in the library are used to change parameter settings or perform advanced operations such as monitoring and changing the composition of the reference set. Regardless of the complexity of the application that uses OCL as its optimization engine, the following general structure must be followed:

- Allocate memory for the optimization model by indicating the number of variables, constraints and requirements in the problem, as well as defining the direction of the optimization as minimize or maximize (OCLSetup).
- Define decision variables (OCLDefineVar).
- Initialize the reference set (OCLInitPop) or generate all solutions in the case of small pure integer problems (OCLGenerateAllSolutions).
- Iterate by retrieving a solution from OCL's database (OCLGetSolution), evaluating the solution (user-provided system evaluator) and placing the evaluated solution back into OCL's database (OCLPutSolution).

Suppose that we would like to use the C version of OCL to search for the optimal solution to the following unconstrained nonlinear optimization problem:

$$\text{Minimize} \quad 100\left(x_2 - x_1^2\right)^2 + (1 - x_1)^2 + 90\left(x_4 - x_3^2\right)^2 + (1 - x_3)^2 + 10.1\left((x_2 - 1)^2 + (x_4 - 1)^2\right) + 19.8(x_2 - 1)(x_4 - 1)$$

$$\text{Subject to} \quad -10 \le x_i \le 10 \quad \text{for } i = 1, \ldots, 4$$

According to the general structure of OCL, we need to start by allocating memory and indicating the direction of the optimization. To do this, we use the OCLSetup function, which has the following prototype:

```
long OCLSetup(long nvar, long ncons, long req, char
      *direc, long lic);
```

`nvar`	An integer indicating the number of decision variables in the problem
`ncons`	An integer indicating the number of constraints in the problem
`req`	An integer indicating the number of requirements in the problem
`direc`	An array of characters with the word "`MAX`" to indicate maximization or "`MIN`" to indicate minimization
`lic`	A valid license number

Therefore, the OCLSetup function call for our example would look like this:

```
        nprob = OCLSetup(4, 0, 0, "MIN", ??????);
```

where `nprob` is a positive integer that indicates a unique problem number within OCL's memory. If OCLSetup returns a negative value, then the setup operation has failed. (Note that in an actual code "??????" must be replaced with a valid license number.) After setting up the problem, we need to define the decision variables using the OCLDefineVar function that has the following prototype:

```
long OCLDefineVar(long nprob, long var, double low,
      double sug, double high, char *type, double step);
```

`nprob`	A unique number that identifies an optimization problem within OCL's memory. This is the identifier returned by OCLSetup.
`var`	An integer indicating the variable number that corresponds to the current definition.
`low`	A double indicating the minimum value for the corresponding variable.
`sug`	A double indicating the suggested value for the corresponding variable. The suggested value is typically included in the initial reference set, unless the value results in an infeasible solution. The `OCLNULL` value can be used when no suggested value is available.
`high`	A double indicating the maximum value for the corresponding variable.
`type`	An array of characters with the word "`CON`" to define a continuous variable or "`DIS`" to define a discrete variable.
`step`	A double indicating the step size for a discrete variable. Step sizes may be integer or fractional and must be strictly greater than zero. Step sizes for continuous variables are ignored.

The function call to define the variables in our example can be programmed as follows:

```
for (i = 1; i <= 4; ++i)
   OCLDefineVar(nprob, i, -10, OCLNULL, 10, "CON", 1);
```

Note that although we use a "1" as the last argument of the function, this value is ignored because all the variables are defined as continuous.

We are now ready to build the starting reference set. This step is performed with a call to the OCLInitPop function. The "Pop" in the function name refers to "population", which is the terminology preferred in the GA community. The prototype of this function consists of a single argument containing the unique problem identifier returned by OCLSetup. The function call looks as follows, assuming that `nprob` is the identifier returned by OCLSetup:

```
                    OCLInitPop(nprob);
```

It is important to point out that all the functions in OCL return an integer value. If the return value is positive, the function call was successful. Otherwise, if the return value is negative, the function call failed and the return value is the error code.

After a successful initialization of the reference set, the search can begin. The search is performed with a series of calls to three functions: OCLGetSolution, a user-provided system evaluator and OCLPutSolution. The first function retrieves a solution from OCL's database, the second evaluates the solution and the third places the evaluated solution back into OCL's database. The prototype for OCLGetSolution is:

```
long OCLGetSolution(long nprob, double *sol);
```

`sol`	A pointer to an array of doubles where OCL places the solution. The array should have enough space to hold the variable values in positions 1 to `nvar`, as defined in OCLSetup.

As before, `nprob` is the unique problem identifier returned by OCLSetup. If the call to OCLGetSolution is successful, the function returns a unique solution identifier `nsol`. The prototype for OCLPutSolution is:

```
long OCLPutSolution(long nprob, long nsol, double
      *objval, double *sol);
```

`nsol`	A solution identifier returned by OCLGetSolution.
`objval`	An array of doubles with the values of the objective function and the requirements. The array should have a size of at least `req+1` positions, as defined in OCLSetup. The objective function value should be `objval[0]` and the *i*th requirement should be `objval[i]`. Note that if no requirements are defined, then `objval` can be dimensioned as a simple double variable. `OCLNULL` may be used to instruct OCL to discard the solution.
`sol`	An array of doubles with the values of the decision variables. If the solution values are the same as the ones retrieved with a call to OCLGetSolution, the value `OCLNULL` may be used.

The evaluation function is outside the scope of OCL and is the responsibility of the user. For our example, we can code the evaluator simply as:

```
double evaluate(double *x)
{
      return(100*pow(x[2]-pow(x[1],2),2)+pow(1-x[1],2)
             +90*pow(x[4]-pow(x[3],2),2)+pow(1-x[3],2)
             +10.1*((pow(x[2]-1,2)+pow(x[4]-1,2))
             +19.8*(x[2]-1)*(x[4]-1));
}
```

Assuming that we want to search for the optimal solution to this problem allowing OCL to perform a maximum of 10000 function evaluations, the code to perform such a search has the following form:

```
for (i = 1; i <= 10000; i++)
{
  nsol = OCLGetSolution(nprob,x);
  if (nsol < 0) {
    printf("OCLGetSolution error code %d\n", nsol);
    exit(1);
  }
  objval = evaluate(x);
  status = OCLPutSolution(nprob, nsol, &objval,
           OCLNULL);
  if (status < 0) {
    printf("OCLPutSolution error code %d\n", status);
    exit(1);
  }
}
```

This code retrieves, evaluates and returns the objective function value of 10000 solutions. It also checks for possible error codes from calls to OCLGetSolution and OCLPutSolution. Note that the partial code above does not keep track of the best solution found. This can be done by adding an "if" statement that compares the objective function value of the current solution with the best objective function value found during the search. Alternatively, the OCLGetBest function can be called at any time during the search to retrieve the values associated with the best solution, which OCL automatically monitors. The function is called with the following arguments:

```
              OCLGetBest(nprob, x, &objval);
```

where `nprob` is the unique problem identifier, `x` is the array where the variable values are stored and `objval` is the variable where the objective function value is returned. The entire C code for this example is shown in Appendix A at the end of this chapter.

4.1 Constraints and Requirements

The illustration in the previous section does not include the OCL functions for constraints and requirements. In this section, we briefly describe how constraints and requirements can be defined with OCL. Assume that we would like to add the following linear constrain to the optimization model for our 4-variable example problem:

$$x_1 + 8x_2 - 3x_4 \leq 5$$

Then after a call to OCLSetup and before a call to OCLInitPop, we add calls to the constraint-related functions: OCLConsCoeff (to change the coefficient of a constraint), OCLConsRhs (to change the right-hand-side of a constraint) and OCLConsType (to change the constraint type). The calls to these functions can be made in any order, as long as they are made before the reference set is initialized and certainly before the search begins through calls to OCLGetSolution and OCLPutSolution. The prototypes for the constraint-related functions are:

```
long OCLConsCoeff(long nprob, long cons, long var,
      double coeffval);
long OCLConsRhs(long nprob, long cons, double rhsval);
long OCLConsType(long nprob, long cons, long type);
```

In these prototypes, `nprob` is the unique problem identifier returned by OCLSetup, `cons` is the constraint number, `var` is the variable number, `coeffval` is the coefficient value, `rhsval`, is the right-hand-side value and `type` is the constraint type (“`OCLLE` = less-than-or-equal, `OCLGE` = greater-than-or-equal, and `OCLEQ` = equal). The following function calls can be used to define the constraint in our example:

```
      OCLConsCoeff(nprob, 1, 1,  1);
      OCLConsCoeff(nprob, 1, 2,  8);
      OCLConsCoeff(nprob, 1, 4, -3);
      OCLConsRhs(nprob, 1, 5);
      OCLConsType(nprob, 1, 1, OCLLE);
```

These function calls assume that the constraint is the first one in the model. Also, the variable numbers match the ones used when OCLDefineVar was called.

The definition of requirements is slightly different from the definition of constraints. As mentioned before, a requirement is basically a bound on an output value of the system evaluator. Suppose that we would like to define a requirement to impose the following nonlinear restriction to our illustrative example:

$$10.8x_1^2 - 2.4x_3x_4 \geq 15.9$$

We use a call to OCLDefineReq to define the requirement. This function has the following prototype:

```
long OCLDefineReq(long nprob, long req, double low,
      double high);
```

Where `nprob` is the unique problem identifier returned by OCLSetup, `req` is the requirement number, `low` is the lower bound for the requirement, and `high` is the upper bound for the requirement. The constant `OCLNULL` can be used to leave either `low` or `high` undefined. The function call for our example is:

```
      OCLDefineReq(nprob, 1, 15.9, OCLNULL);
```

In addition to this definition, we need to modify the system evaluator. The evaluator must return the value of the objective function in `objval[0]` and the value of the requirement in `objval[1]`. The new evaluator looks like this:

```
void evaluate(double *x, double *objval)
{
   objval[0] = 100*pow(x[2]-pow(x[1],2),2)+pow(1-x[1],2)
               +90*pow(x[4]-pow(x[3],2),2)+pow(1-x[3],2)
               +10.1*((pow(x[2]-1,2)+pow(x[4]-1,2))
               +19.8*(x[2]-1)*(x[4]-1));
   objval[1] = 10.8*pow(x[1],2)-2.4*x[3]*x[4];
}
```

Other small changes are necessary to make OCL work with the requirement that has been defined. Obviously, the declaration of `objval` must be changed from a single double to an array of doubles and the `evaluate` function must be changed from double to void. The changes are reflected in the following partial code:

```
void evaluate(double *, double *);
.
.
  double objval[3];
.
.
  for (i = 1; i <= TOT_ITER; i++) {
    nsol = OCLGetSolution(nprob, x);
    evaluate(x, objval);
    OCLPutSolution(nprob, nsol, objval, OCLNULL);
  }
```

When requirements are included in an optimization model, OCLGetBest indicates the feasibility of the best solution. The return values for OCLGetBest can be either 0 if the best solution is requirement-feasible, 1 if the best solution is requirement-infeasible or a negative value representing an error code. Note that while all the solutions generated by OCL are constraint-feasible (if the constraint space is not empty), some may be requirement-infeasible and this is why the OCLGetBest is designed to provide information regarding the requirement-feasibility of the best solution found during the search.

The discussion about the functionality of OCL in which we have engaged is not meant to be exhaustive. OCL has additional functions that can be used to modify the way the search is performed. For example, OCL includes functions to change the default parameter settings. A function such as OCLSetPopSize can be used to change the number of solutions that are carried out in the reference set throughout the search. In some applications, changing this parameter may improve the quality of the best solution found. Other functions, such as OCLSetBoundFreq, are meant for the advanced user, who has a thorough understanding of both the problem context to solve and the way the search method works.

4.2 Boundary Search Strategy

The OCLSetBoundFreq controls the application of the boundary search strategy. This strategy is a mechanism to generate non-convex combinations of two solutions. In particular, the set of points generated on the line defined by x' and x'', but beyond the segment $[x', x'']$ "outside" of x'' is given by:

$$x = x'' + (x'' - x')t \qquad \text{for } t > 0$$

Note that the "normal" combination method in OCL defines $t = r/2$, where r is a random number between 0 and 1. The boundary strategy, however, considers three strategies to generate non-convex combinations in the (x', x'') line:

Strategy 1: Computes the maximum value of t that yields a feasible solution x when considering both the bounds and the constraints in the model.

Strategy 2: Considers the fact that variables may hit bounds before leaving the feasible region relative to other constraints. The first departure variable from the feasible region may happen because some variable hits a bound, which is followed by the others, before any of the linear constraints is violated. In such a case, the departing variable is fixed at its bound when it hits it, and the exploration continues with this variable held constant. OCL does this with each variable that encounters a bound before other constraints are violated. The process finishes when the boundary defined by the other constraints is reached.

Strategy 3: Considers that the exploration hits a boundary that may be defined by either bounds or any of the linear constraints. When this happens, one or more constraints are binding and the corresponding t-value cannot be increased without causing the violation of at least one constraint. At this point, OCL chooses a variable to make a substitution that geometrically corresponds to a projection that makes the search continue on a line that has the same direction, relative to the constraint that was reached, as it did upon approaching the hyperplane defined by the constraint. The process continues until the last unfixed variable hits a constraint. At this point, the value of all the previously fixed variables is computed.

Each of these three boundary strategies generates a "boundary solution" x^b outside x''. OCL also generates the solution in the midpoint between x^b and x''. Interchanging the role of x' and x'' gives the extension outside the "other end" of the line segment. The mechanism results in a total of 4 non-convex solutions out of a single combination of a pair of reference points. The user establishes, by setting the appropriate value using the OCLBoundFreq function, the percentage of combinations in which the boundary strategy is applied. The default value calls for applying the boundary strategy 50% of the time. Although this default value is fairly robust across a variety of problem classes, for problems where the best values are suspected to lie near the boundary of the feasible region, improved solutions may be found when the frequency value is increased. Similarly, if the best solutions to an optimization problem are not near a boundary, improved outcomes may result from decreasing the frequency for applying the boundary strategy.

5. OCL APPLICATION

In this section, we apply OCL to a set of hard nonlinear and unconstrained optimization problems. We have built an application based on OCL that allows us to test the performance of the library as compared to a competing generic optimizer based on genetic algorithms.

Table 2. Test problems

Problem number	Name and parameter values	Optimal value
1	Colville	0
2	Perm(4,0.5)	0
3	Perm(9,10^8)	0
4	Perm0(4,10)	0
5	Perm0(10,100)	0
6	PowerSum(8,18,44,114)	0
7	PowerSumZ(z_1,...,z_8,1,15) $z_1=1$, $z_i=2+z_{i-1}$	0
8	PowerSumZ(z_1,...,z_{10},0,2) $z_i=1/i$	0
9	PowerSumZ(z_1,...,z_6,-6,6) $z_i=i-1$	0
10	Trid(6)	-50
11	Trid(10)	-210
12	Rosenbrock(8)	0
13	Rosenbrock(20)	0
14	Rosenbrock2(2)	0
15	Rosenbrock2(6)	0
16	SixHumpCamelBack	-1.0316285
17	Levy(30)	0
18	Beale	0
19	Booth	0
20	Matyas	0
21	Powell(24)	0
22	Griewank(20)	0
23	Rvan(25)	0
24	Rastrigin(10)	0
25	Rastrigin(20)	0
26	Schwefel1(10)	0
27	Schwefel1(20)	0
28	Ackley(50)	0
29	Ackley(100)	0
30	Sphere(100)	0

Table 2 shows the set of 30 test problems that we have gathered from the following web pages:

http://www.maths.adelaide.edu.au/Applied/llazausk/alife/realfopt.htm
http://solon.cma.univie.ac.at/~neum/glopt/my_problems.html
http://www-math.cudenver.edu/~rvan/phd/node32.html

The number between parentheses associated with some of the function names are the parameter values for the corresponding objective function. A typical parameter refers to the number of variables in the function, since many of these functions expand to an arbitrary number of variables. Although the objective functions are built in a way that the optimal solutions are known, the optimization problems cannot be trivially solved by search procedures that do not exploit the special structure embedded in each function. A detailed description of the objective functions is provided in Appendix B.

Our OCL-based code has a "main" function with two arguments: the problem number and the total number of function evaluations. The problem number is used to select the correct objective function from the catalog of functions inside the system evaluator. The code also uses this number to allocate the appropriate memory using OCLSetup and to define the bounds for each variable calling OCLDefineVar.

We compare the performance of OCL with the results obtained from running Genocop III (http://www.coe.uncc.edu/~zbyszek/gchome.html) on the same set of test problems. Genocop III is the third version of a genetic algorithm designed to search for optimal solutions to optimization problems with continuous variables and linear and nonlinear constraints. The description of the first version of Genocop appears in the book by Michalewicz (1994). While this GA optimizer does not handle discrete variables, it does provide a way for explicitly defining nonlinear constraints. The performance of the Genocop depends on a set of 12 parameters (without counting the frequency distribution for the application of each operator). We did not attempt to find the best parameter setting for the set of problems on hand and instead we used the default values that the system recommends. Similarly, we used the default values for the OCL parameters, such as the size of the reference set and the frequency of application of the boundary strategy. A summary of our comparison is shown in Table 3.

The results in Table 3 were obtained allowing each procedure to perform 10,000 function evaluations. The values in bold indicate which procedure yields the solution with the better objective function value for each problem. The following observations can be made from the results in this table.

- OCL finds solutions that on the average are better than those found by Genocop, within the scope of the search (i.e., 10000 evaluations).
- OCL finds better solutions than Genocop more frequently (20 times for OCL vs. 3 times for Genocop).
- Genocop is on average 4 times faster than OCL.
- Problems 2 and 6 pose grave difficulties to both OCL and Genocop.

Table 3. Comparison of OCL and Genocop III

			Objective function value		CPU seconds	
Prob.	Var.	Optimal	OCL	Genocop	OCL	Genocop
1	4	0	**0.000361**	0.729826	3.8	0.3
2	4	0	**0.130655**	1.794891	4.0	0.9
3	9	0	**4.5839E+15**	2.12E+16	4.8	3.7
4	4	0	**0.073611**	0.79538	4.3	0.8
5	10	0	**74.020773**	276.9857	5.4	3.7
6	4	0	**0.00883**	0.02553	151.6	0.6
7	8	0	**8.30E+08**	1.23E+14	3.5	2.9
8	10	0	0	0	5.9	3.4
9	6	0	**147.688802**	1452.561	6.8	1.8
10	6	-50	**-49.958165**	-46.481	33.6	0.4
11	10	-210	**-127.15042**	-31.5584	3.8	0.9
12	8	0	**0.483572**	5.601258	3.2	0.8
13	20	0	**5.599877**	7.684962	6.9	2.8
14	2	0	**1.990099**	36.3949	4.8	0.3
15	6	0	**5.949696**	273.3088	6.3	0.8
16	2	-1.03163	-1.03163	-1.03163	5.8	0.3
17	30	0	0.69632	**0.300885**	74.3	5.5
18	2	0	0	0.00075	4.1	0.2
19	2	0	0	0	2.8	0.2
20	2	0	0	0	4.8	0.2
21	24	0	**0.085342**	2.740024	8.7	3.2
22	20	0	**0**	1.076349	3.8	2.6
23	25	0	**0.679836**	37.37577	42.9	3.9
24	10	0	**0**	1.025595	4.5	0.9
25	20	0	**0**	10.5079	6.3	2.5
26	10	0	844.068704	**1.387158**	1.8	0.8
27	20	0	1506.06706	**134.4911**	2.4	2.1
28	50	0	0	0	16.8	13.4
29	100	0	0	0	103.6	46.6
30	100	0	**2.418512**	1114.451	60.3	43.7

In addition to comparing the performance of both systems when considering the final solutions and the total computational time to find them, it is important to assess how quickly each method reaches the best solutions. Reaching good solutions quickly becomes more critical when the complexity of the system evaluator increases. Consider, for example, an application for which a single evaluation of the objective function consists of the execution

of a computer simulation that requires 2 CPU minutes. Clearly, in this context, the time to generate each solution becomes negligible. At the same time, it is not feasible to search for the best solution employing 10,000 function evaluations, unless one is willing to wait for 2 weeks to obtain a relatively good answer. A more reasonable approach is to limit the search to 500 function evaluations, whose execution will require somewhat less than 17 hours to execute.
Figure 6 depicts the trajectory followed by OCL and Genocop when tracking the average objective function value in a set of 28 problems that excludes problems 2 and 6. Note that the graph is drawn on a logarithm scale to be able to accommodate the large average objective function value of 8.8E+12 yielded by Genocop after 100 evaluations.

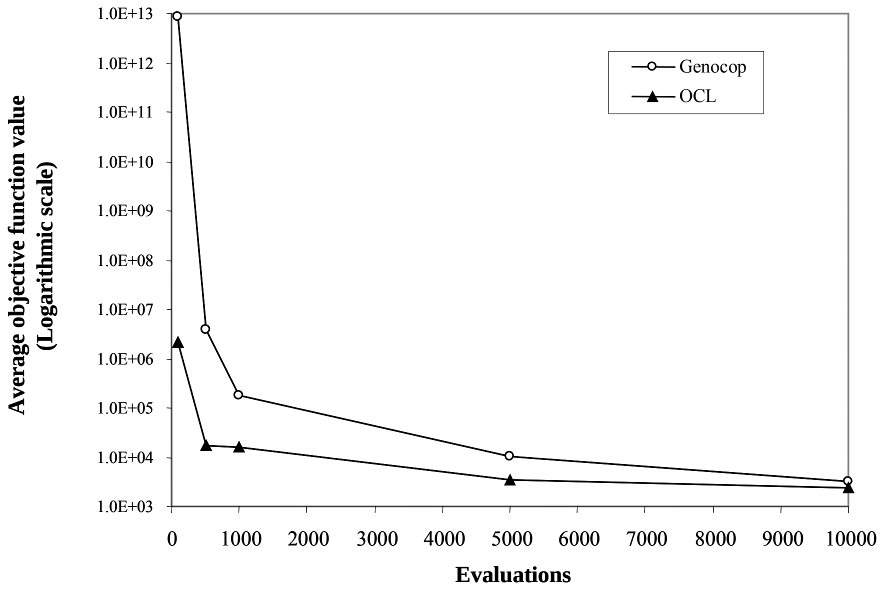

Figure 6. Performance graph for OCL and Genocop III

The purpose of constructing the performance graph depicted in Figure 6 is to assess the aggressiveness of each procedure as measure by the speed in which the search is capable of finding reasonably good solutions. Based on our testing, OCL seems to be more aggressive than Genocop. The aggressiveness in OCL, however, does not compromise the quality of the final solution. In other words, OCL aggressively attempts to improve upon the best solution during the early stages of the search and at the same time it makes use of diversifying mechanisms to be able to sustain an improving trajectory when allowed to extend the search beyond a limited number of objective function evaluations.

The aggressiveness of OCL as compared to Genocop can be measure by the gap between the average objective function values found by each method. After 500 evaluations, Genocop average is more than 200 times larger than OCL average. OCL average remains lower than Genocop values throughout the search with the final average for Genocop being 36% higher than OCL's, as shown in Table 4.

Table 4. Genocop solutions compared to OCL solutions

Evaluation number	Percentage deviation of the Genocop solutions from OCL solutions
100	417266887%
500	21639%
1000	983%
5000	202%
10000	36%

The performance graph in Figure 6 and the percentage deviation values in Table 4 seem to substantiate that OCL would be the preferred solution method for optimizing complex systems, for which the evaluation of the objective function determines the computational time of the search. These exhibits also show a general trend that seems to reveal that the quality of the solutions found by both methods becomes more alike as the number of evaluations increases.

6. CONCLUSIONS

In this chapter, we discussed the notion of optimizing a complex system when a "black-box" evaluator can be used to estimate performance based on a set of input values. In many business and engineering problems, the black-box evaluator has the form of a simulation model capable of mapping inputs into outputs, where the inputs are values for a set of decision variables and the outputs include the objective function value. We also addressed the development and application of a function library that uses scatter search in the context of optimizing complex systems.

The functionality of the library was illustrated with an illustrative example in nonlinear optimization. Our illustration resulted in a sample code that can be modified and expanded to apply OCL in other situations. Finally, we tested OCL by comparing its performance with Genocop III, a third-generation genetic algorithm. Our experiments with 30 nonlinear optimization problems show that OCL is a search method that is both aggressive and robust. It is aggressive because it finds high-quality solutions early in the search. It is robust because it continues to improve upon the best solution when allowed to search longer. These characteristics make OCL an ideal solution method for applications in which the evaluation of the objective function requires a non-trivial computational effort.

7. REFERENCES

Campos, V., F. Glover, M. Laguna and R. Martí (1999) "An Experimental Evaluation of a Scatter Search for the Linear Ordering Problem," University of Colorado at Boulder.

Campos, V., M. Laguna and R. Martí (1999) "Scatter Search for the Linear Ordering Problem," *New Methods in Optimization,* D. Corne, M. Dorigo and F. Glover (Eds.), pp. 331-339, McGraw-Hill.

Glover, F., M. Laguna and R. Martí (1999) "Scatter Search," to appear in *Theory and Applications of Evolutionary Computation: Recent Trends,* A. Ghosh and S. Tsutsui (Eds.), Springer-Verlag.

Glover, F. (1998) "A Template for Scatter Search and Path Relinking," in *Artificial Evolution, Lecture Notes in Computer Science* 1363, J.-K. Hao, E. Lutton, E. Ronald, M. Schoenauer and D. Snyers (Eds.), Springer-Verlag, pp. 13-54.

Laguna, M. "Scatter Search," to appear in *Handbook of Applied Optimization,* P. M. Pardalos and M. G. C. Resende (Eds.), Oxford Academic Press.

Michalewicz, Z. (1994) *Genetic Algorithms + Data Structures = Evolution Programs,* Second-Extended Edition, Springer-Verlag.

Optquest Callable Library User's Manual, Optimization Technologies, Inc., Boulder, CO, 2000 (www.opttek.com).

Appendix A

The following is the complete C code to search for the optimal solution of the example problem in section 4 using OCL. For simplicity we have left out the error-code checking. In actual applications, however, it is recommended to check for the return values of the OCL function in order to detect errors during the search.

```c
#include "ocl.h"
#include <stdio.h>
#include <math.h>

#define NUM_VAR  4
#define TOT_ITER 10000

double evaluate(double *);

void main(void)
{
  double x[NUM_VAR+1], objval;
  long   nprob, nsol;
  int    i;

  /* Allocating memory */
  nprob = OCLSetup(NUM_VAR,0,0,"MIN", ??????);

  /* Defining variables */
  for (i = 1; i <= NUM_VAR; i++)
    status = OCLDefineVar(nprob, i, -10, OCLNULL, 10,"CON", 1);

  /* Initializing the reference set */
  OCLInitPop(nprob);

  /* Generate and evaluate TOT_ITER solutions */
  for (i = 1; i <= TOT_ITER; i++) {
    nsol = OCLGetSolution(nprob, x);
    objval = evaluate(x);
    OCLPutSolution(nprob, nsol, &objval, OCLNULL);
  }

  /* Display the best solution found */
  status = OCLGetBest(nprob, x, &objval);
  printf("Best solution value is %9.6f\n", i, objval);
  for(i = 1; i <= NUM_VARIABLES; i++)
    printf("x[%2d] = %9.6lf\n", i, x[i]);

  /* Free OCL memory */
  status = OCLGoodBye(Example);
}

/* Evaluation function */

double evaluate(double *x)
{
  return(100*pow(x[2]-pow(x[1],2),2)+pow(1-x[1],2)
    +90*pow(x[4]-pow(x[3],2),2)+pow(1-x[3],2)+
    10.1*((pow(x[2]-1,2)+pow(x[4]-1,2))+19.8*(x[2]-1)*(x[4]-1));
}
```

Appendix B

This appendix contains the description of the set of test functions in Table 2. The description consists of the objective function and the bounds for each variable.

Colville

Minimize
$$f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 + 90(x_4 - x_3^2)^2 + (1 - x_3)^2 + 10.1\left((x_2 - 1)^2 + (x_4 - 1)^2\right) + 19.8(x_2 - 1)(x_4 - 1)$$

Subject to $-10 \le x_i \le 10$ for $i=1,...,4$.

Perm(n, β)

Minimize
$$f(x) = \sum_{k=1}^{n}\left(\sum_{i=1}^{n}\left(i^k + \beta\right)\left(\left(\frac{x_i}{i}\right)^k - 1\right)\right)^2$$

Subject to $-n \le x_i \le n$ for $i=1,..., n$.

Perm0(n, β)

Minimize
$$f(x) = \sum_{k=1}^{n}\left(\sum_{i=1}^{n}\left(i + \beta\right)\left(x_i^k - \left(\frac{1}{i}\right)^k\right)\right)^2$$

Subject to $-n \le x_i \le n$ for $i=1,..., n$.

PowerSum($b_1,...,b_n$)

Minimize
$$f(x) = \sum_{k=1}^{n}\left(\left(\sum_{i=1}^{n} x_i^k\right) - b_k\right)^2$$

Subject to $0 \le x_i \le n$ for $i=1,..., n$.

PowerSumZ($z_1,\ldots,z_n, l, u$)

Minimize $f(x) = \sum_{k=1}^{n} \left(\left(\sum_{i=1}^{n} x_i^k \right) - b_k \right)^2$

Subject to $b_k = \sum_{i=1}^{n} z_i^k$ $k=1,\ldots, n$

$l \le x_i \le u$ for $i=1,\ldots, n$.

Trid(n)

Minimize $f(x) = \left(\sum_{i=1}^{n} (x_i - 1)^2 \right) - \sum_{i=2}^{n} x_i x_j$

Subject to $-n^2 \le x_i \le n^2$ for $i=1,\ldots, n$.

Rosenbrock(n)

Minimize $f(x) = \sum_{i=1}^{n/2} 100(x_{2i} - x_{2i-1}^2)^2 + (1 - x_{2i-1})^2$

Subject to $-10 \le x_i \le 10$ for $i=1,\ldots, n$.

Rosenbrock2(n)

Minimize $f(x) = \sum_{i=1}^{n} 100(x_i - x_{i-1}^2)^2 + (1 - x_{i-1})^2$

Subject to $-10 \le x_i \le 10$ for $i=1,\ldots, n$.

SixHumpCamelBack

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

Subject to $-10 \le x_1, x_2 \le 10$

Levy(*n*)

Minimize

$$f(x) = \sin^2(\pi y_1) + \sum_{i=1}^{k-1} (y_i - 1)^2 (1 + 10\sin^2(\pi y_i + 1)) + (y_k - 1)^2 (1 + \sin^2(2\pi x_k))$$

Subject to $y_i = 1 + \frac{x_i - 1}{4}$ for $i = 1, \ldots, n$

$-10 \le x_i \le 10$ for $i = 1, \ldots, n$

Beale

Minimize

$$f(x) = (1.5 - x_1 + x_1 x_2)^2 + (2.25 - x_1 + x_1 x_2^2)^2 + (2.625 - x_1 + x_1 x_2^3)^2$$

Subject to $-4.5 \le x_1, x_2 \le 4.5$

Booth

Minimize $f(x) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2$

Subject to $-10 \le x_1, x_2 \le 10$

Matyas

Minimize $f(x) = 0.26(x_1^2 + x_2^2) - 0.48 x_1 x_2$

Subject to $-10 \le x_1, x_2 \le 10$

Powell(*n*)

Minimize

$$f(x) = \sum_{j=1}^{n/4} (x_{4j-3} + 10x_{4j-2})^2 + 5(x_{4j-1} - x_{4j})^2 + (x_{4j-2} - 2x_{4j-1})^4 + 10(x_{4j-3} - x_{4j})^4$$

Subject to $-4 \le x_i \le 5$ for $i = 1, \ldots, n$

Griewank(n)

Minimize $f(x) = \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$

Subject to $-600 \le x_i \le 600$ for $i = 1, \ldots, n$

Rvan(n)

Minimize $f(x) = \sum_{i=1}^{n} i(2x_i^2 - x_{i-1})^2 + (x_1 - 1)^2$

Subject to $-10 \le x_i \le 10$ for $i = 1, \ldots, n$

Rastrigin(n)

Minimize $f(x) = 10n + \sum_{i=1}^{n} x_i^2 - 10\cos(2\pi x_i)$

Subject to $-5.12 \le x_i \le 5.12$ for $i = 1, \ldots, n$

Schwefel1(n)

Minimize $f(x) = 418.9829n + \sum_{i=1}^{n} - x_i \sin\sqrt{|x_i|}$

Subject to $-500 \le x_i \le 500$ for $i = 1, \ldots, n$

Ackley(n)

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

Subject to $-30 \le x_i \le 30$ for $i = 1, \ldots, n$

Sphere(n)

Minimize $f(x) = \sum_{i=1}^{n} x_i^2$

Subject to $0 \le x_i \le 100$ for $i = 1, \ldots, n$