# Writing Global Optimization Software

Leo Liberti

DEI, Politecnico di Milano, P.zza L. da Vinci 32, 20133 Milano, Italy
liberti@elet.polimi.it

**Summary.** Global Optimization software packages for solving Mixed-Integer Nonlinear Optimization Problems are usually complex pieces of codes. There are three main difficulties in coding a good GO software: embedding third-party local optimization codes within the main global optimization algorithm; providing efficient memory representations of the optimization problem; making sure that every part of the code is fully re-entrant. Finding good software engineering solutions for these difficulties is not enough to make sure that the outcome will be a GO software that works well, of course. However, starting from a sound software design makes it easy to concentrate on improving the efficiency of the global optimization algorithm implementation. In this paper we discuss the main issues that arise when writing a global optimization software package, namely software architecture and design, symbolic manipulation of mathematical expressions, choice of local solvers and implementation of global solvers.

**Key words:** MINLP, symbolic computation, multistart, variable neighbourhood search, branch-and-bound, implementation, software design.

## 1 Introduction

The object of Global Optimization (GO) is to find a solution of a given nonconvex mathematical programming problem. By "solution" we mean here a *global* solution, as opposed to a *local* solution; i.e., a point where the objective function attains the optimal value with respect to the whole search domain. By contrast, a solution is local if it is optimal with respect to a given neighbourhood. We require the objective function and/or the feasible region to be nonconvex because in convex mathematical programming problems every local optimum is also a global one. Consequently, any method solving a convex problem locally also solves it globally.

In this paper we address Mixed-Integer Nonlinear Programming (MINLP) problems in their most general setting:

$$
\left.\begin{array}{rrcl}
\min\limits_{x\in\mathbb{R}^n} & & c^T x + f(x) & \\
\text{s.t.} & l \leq & Ax + g(x) & \leq u \\
& x^L \leq & x & \leq x^U \\
& & x_i \in \mathbb{Z} & \forall i \in Z
\end{array}\right\} \tag{1}
$$

In the above formulation, $x$ are the problem variables; some of them (those indexed by the set $Z \subseteq \{1,\ldots,n\}$) are constrained to take discrete values. The objective function and constraints consist of a linear and nonlinear part: $f : \mathbb{R}^n \to \mathbb{R}$ is a possibly nonlinear function, $g : \mathbb{R}^n \to \mathbb{R}^m$ is a vector of $m$ possibly nonlinear functions, $c \in \mathbb{R}^n$, $A$ is an $m \times n$ matrix, $l, u \in \mathbb{R}^m$ are the constraint bounds (which may be set to $\pm\infty$ if a particular constraint is never active), and $x^L, x^U \in \mathbb{R}^n$ are the variable bounds (again, some of these bounds may be set to $\pm\infty$). We limit the discussion to the case where $f, g$ are continuous functions of their arguments. Formulation (1) encompasses most kinds of mathematical programming problems. For example, if $f = 0$, $g$ is a constant (say $-b$), $l = -\infty$, $u = 0$, $x^L = 0$, $x^U = \infty$, and $Z = \emptyset$ we have the canonical formulation of a Linear Programming problem (LP).

At the time of writing this paper, there is no GO software established as standard. In fact, the GO software market is still rather poor and definitely belonging to the academic world; GO is not being used extensively in the corporate environment yet. Part of the reason for this is that linear modelling is often a sufficient approximation to real-life processes, so GO software is not required. Even when a nonlinear model arises, a lot of effort is put into linearizing it by standard or novel modelling techniques. Finally, most GO algorithms rely on calling a local optimization procedure as a black-box function, and the fastest local optimization algorithms for Nonlinear Programming (NLP) problems are often inherently fragile: they may fail to converge for many different reasons even when the problem itself is reasonably smooth and well-behaved. This makes general-purpose robust GO software codes virtually non-existent (by contrast, GO software targeted at solving one particular problem can be made rather robust). Seeing as there are no standardized guidelines for designing GO software, this paper attempts to fill the gap by discussing a set of methods that should make general-purpose GO software robust and hopefully efficient. This work is based on two different "software frameworks" for GO designed by the author during his Ph.D. One of these, $oo\mathcal{OPS}$(object-oriented OPtimization System) [48], can be tested via the on-line interface at `http://or.dhs.org/liberti/ooOPS`. The other, MORON (MINLP Optimization and Reformulation Object-oriented Navigator), is still very much work in progress.

Most published papers on GO proposing a novel algorithm or a variant of an existing algorithm also include computational results which have been derived from some kind of software implementation of the method being proposed. This suggests that there should be quite a lot of working GO software available. Unfortunately, this is not the case. Most of these implementations are no more than prototypes designed to produce the computational results.

Methodologically speaking, at least in the academic world, there is nothing wrong about writing software with a view to publishing a paper and nothing else. This is not the approach to software writing that we are proposing to illustrate here, however. We are interested in reviewing software design methods and GO algorithms for general-purpose global optimization software. A clarifying analogy could be that we mean to describe methods to write a GO software akin to what CPLEX [30] is to mixed-integer linear programming. CPLEX is a software that can potentially solve any MILP problem which is thrown at it, regardless of problem structure. It would be desirable to have a corresponding software for solving arbitrarily structured MINLP problems. CPLEX owes much of its success to a period of intense research in the field of solution methods for MILPs in their most general form (cutting plane, branch-and-bound, branch-and-price), as well as algorithmic improvements which made the proposed algorithm practically viable (new families of cuts, polyhedral theory, automatic reformulation methods, hierarchies of convex relaxations leading to the convex hull). CPLEX is by no means the only MILP-solving software on the market: but its efficiency is widely recognized, and we believe that it may be called a *de facto* standard software for solving MILPs. In the last two decades, many precise and heuristic GO methods have been proposed for solving MINLPs. The work on algorithmic improvements for these methods, however, is lagging behind if compared to the MILP scene. For instance, symbolic reformulation techniques for MINLPs are still largely an unexplored world, at least as far as computer implementations go: the most common way to proceed seems to be that the reformulation is carried out by hand, and then the reformulated problem is hard-coded in a "single-purpose" solver wrapper. By contrast, automatic symbolic reformulation algorithms are a crucial part of the CPLEX pre-solver. One possible explanation for such a different state of affairs is that software design for MILP solvers is inherently simpler than that required by GO algorithms, and even a relatively careless design can result in a robust, efficient MILP solver. The situation is very different in GO algorithms where re-entrancy, good memory management, efficient data passing, and the ability to treat complex pieces of software like an NLP local solver as a black box are all of paramount importance. Yet, all of these issues have hardly been addressed in the existing GO literature. This paper attempts to move a few steps in this direction.

We propose a software design based on a framework that deals with the basic tasks required by any optimization solver code: reading the problem into memory, performing symbolic manipulation, providing and modifying problem structure data, wrapping the solvers into independent callable modules, configuring and running the solvers. Each solver can be called by any other solver on any given problem, effectively allowing the use of any solver (be it local or global) as a black-box. Solvers usually provide a numerical solution as output, but a solver in this framework may even be a specialized symbolic manipulation routine whose job is to change the structure of the problem.

The rest of this paper is organized as follows. Section 2 is a review of three existing algorithms that can be applied to optimization problems in form (1), where $Z$ (the set of integer variables) may or may not be non-empty; namely, MultiStart (MS), Variable Neighbourhood Search (VNS) and spatial Branch-and-Bound (sBB). Section 3 is a review of existing general-purpose GO software packages. Section 4 lays the foundations for the software framework where the GO solver codes are executed. Section 5 is an in-depth treatment of the techniques used in the symbolic manipulation of the mathematical expressions in the objective function and constraints of the problem. Section 6 is a review of some of the existing local LP and NLP solvers. Section 7 contains descriptions of the global optimization solver implementations of MS, VNS and sBB within the described framework.

## 2 Global Optimization algorithms

This section presents three of the the existing algorithms targeted at the solution of problem (1): MultiStart (MS) (in fact a variant thereof called Multi Level Single Linkage[1] (MLSL)), Variable Neighbourhood Search[2] (VNS) and spatial Branch-and-Bound (sBB). The first two are classified as stochastic algorithms, the latter as deterministic.

Most GO algorithms are two-phase. The solution space $S$ is explored exhaustively in the global phase, which iteratively identifies a promising starting point $\tilde{x}$. In the local phase, a local optimum $x^*$ is found starting from each $\tilde{x}$. The local phase usually consists of a deterministic local descent algorithm which the global phase calls as a black-box function. The global phase can be stochastic or deterministic. Algorithms with a stochastic global phase are usually heuristic algorithms, whereas deterministic global phases often provide a certificate of optimality, making the algorithm precise.

Stochastic global phases identify the starting points $\tilde{x}$ either by some kind of sampling in $S$ (sampling approach), or by trying to escape from the basin of attraction of the local minima $x^*$ found previously (escaping approach), or by implementing a blend of these two approaches. Stochastic global phases do not offer certificates of optimality of the global optima they find, and they usually only converge to the global optimum with probability 1 in infinite time. In practice, though, these algorithms are very efficient, and are, at the time of this writing, the only viable choice for solving reasonably large-scale MINLPs. The efficiency of stochastic GO algorithms usually depends on the proper fine-tuning of the algorithmic parameters controlling intensification of sampling, extent of escaping and verification of termination conditions.

Deterministic global phases usually work by partitioning $S$ into smaller sets $S_1, \ldots, S_p$. The problem is then solved globally in each of the subsets

[1] Also see Chapter 5.
[2] Also see Chapters 6, 11 (Section 1.1).

$S_j$. The global solution of each restriction of the problem to $S_j$ is reached by recursively applying the global phase to each $S_j$ until a certificate of optimality can be obtained for each $S_j$. The certificate of optimality is obtained by computing upper and lower bounds $u, l$ to the objective function value. A local optimum $x^*$ in $S_j$ is deemed global when $|u - l| < \varepsilon$, where $\varepsilon > 0$ is a (small) constant. The convergence proofs for these algorithms rely on the analysis of the sequence of upper and lower bounds: it is shown that these sequences contain $\varepsilon$-convergent subsequences. The certificate of optimality for the global optimum of the problem with respect to the whole solution space $S$ is therefore really a certificate of $\varepsilon$-global optimality. Such global phases are called Branch-and-Select (the partitioning of the sets $S_j$ is called branching; the algorithm relies on selection of the most promising $S_j$ for the computation of bounds) [84]. Deterministic algorithms tend to be fairly inefficient on large-scale MINLPs, but they perform well on small and medium-scale problems. The efficiency of Branch-and-Select algorithms seems to depend strongly on the particular instance of the problem at hand, and on the algebraic formulation of the problem.

We note here, in passing, that not all solution methods for GO problems follow such iterative approaches of finding candidate solution points $\tilde{x}$ and applying local descents to identify the closest local minimum $x^*$. Where $f, g$ are very expensive to evaluate, the local phase is usually skipped (as it requires many function evaluations) and $x^*$ is set to $\tilde{x}$ (thus, these GO algorithms only consist of the global phase). There exist algebraic methods (based on the computation of Gröbner bases) that solve polynomially constrained polynomial problems, which do not actually present either a global or a local phase [26].

In the rest of this section, we shall give a short presentation of the following stochastic algorithms: Multistart (MS), Variable Neighbourhood Search (VNS); and of the deterministic algorithm called "spatial Branch-and-Bound" (sBB). In fact, there are many other stochastic GO algorithms. To name but a few which are not discussed in this paper: tabu search [38], genetic algorithms [72], simulated annealing [44], differential evolution [74], adaptive Lagrange multiplier methods [85], ant colony simulation [51], ruin and recreate [77], dynamic tunnelling methods [65].

### 2.1 Multistart

Multistart (MS) algorithms are conceptually the most elementary GO algorithms. Many local descents are performed from different starting points. These are sampled with a rule that is guaranteed to explore the solution space exhaustively (in infinite time), and the local minimum with the best objective function value is deemed the "global optimum". MS algorithms are stochastic GO algorithms with a sampling approach. Their many variants usually differ in sampling and local descent strategies.

One of the main problems that MS algorithms face is that the same local optimum is identified many times when the sampling rule picks starting points in the basin of attraction of the same local optimum. Since the local descent is the most computationally expensive step of MS, it is important to control the extent to which this situation occurs. Obviously, identifying a local optimum many times is useless to the end of finding the global optimum. The most common method used to inhibit multiple local descents to start in the same basin of attraction is called *clustering*. Sampled starting points are grouped together in clusters of nearby points, and only one local descent is performed in each cluster, starting from the most promising (in terms of objective function value) cluster point. One particularly interesting idea for clustering is the Multi Level Single Linkage (MLSL) method [60, 61]: a point $x$ is clustered together with a point $y$ if $x$ is not too far from $y$ and the objective function value at $y$ is better than that at $x$. The clusters are then represented by a directed tree, the root of which is the designated starting point from where to start the local optimization procedure (see Fig. 1).

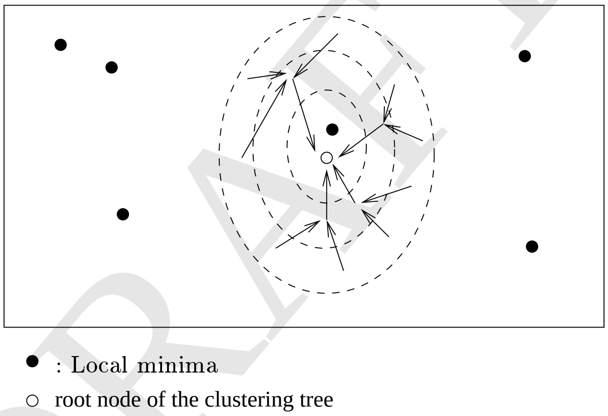


**Fig. 1.** Linkage clustering in the stochastic global phase. The points in the cluster are those incident to the arcs; each arc $(x, y)$ expresses the relation "$x$ is clustered to $y$". The arcs point in the direction of objective function descent. The root of the tree is the "best starting point" in the cluster.

One of the main problems with clustering is that as the number of problem variables increases, the sampled points are further apart (unless one is willing to spend an exponentially increasing amount of time on sampling, but this is very rarely acceptable) and cannot be clustered together so easily.

Despite their conceptual simplicity, MS algorithms for GO usually perform rather well on medium to large scale problems. The work of Locatelli and Schoen on MS with random and quasi-random sampling shows that MS is, to date, the most promising approach to solving the Lennard-Jones potential energy problem, arising in the configuration of atoms in a complex molecules [46, 67, 68, 47, 69].

A MS algorithm for GO problems in form (1) (called SobolOpt[3]) was developed by Kucherenko and Sytsko [37]. The innovation of the SobolOpt algorithm is that the sampling rule is not random but deterministic. More precisely, it employs Low-Discrepancy Sequences (LDSs) of starting points called *Sobol' sequences* whose distributions in Euclidean space have very desirable uniformity properties. Uniform random distributions where each point is generated in a time interval (as is the case in practice when generating a sampling on a computer) are guaranteed to be uniformly distributed in space in infinite time with probability 1. In fact, these conditions are very far from the normal operating conditions. LDSs, and in particular Sobol' sequences, are guaranteed to be distributed in space as uniformly as possible even in finite time. In other words, for any integer $N > 0$, the first $N$ terms of a Sobol' sequence do a very good job of filling the space evenly. One further very desirable property of Sobol' sequences is that any projection on any coordinate hyperplane of the Euclidean space $\mathbb{R}^n$ containing $N$ $n$-dimensional points from a Sobol' sequence will still contain $N$ projected $(n-1)$-dimensional Sobol' points. This clearly does not hold with the uniform grid distribution where each point is located at a coordinate lattice point (in this case the number of projected points on any coordinate hyperplanes is $O(N^{\frac{n-1}{n}})$, as shown in Fig. 2). The comparison between grid and Sobol' points in $\mathbb{R}^2$ is shown in Fig. 3.

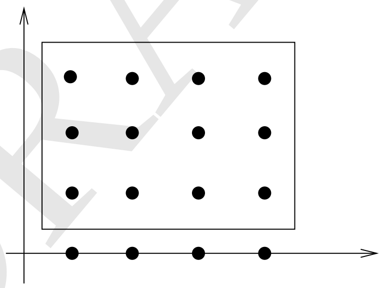

**Fig. 2.** Projecting a grid distribution in $\mathbb{R}^2$ on the coordinate axes reduces the number of projected points. In this picture, $N = 12$ but the projected points are just 4.

The SobolOpt algorithm has been used to successfully solve the Kissing Number Problem (KNP — determining the maximum number of nonoverlapping spheres of radius 1 that can be arranged adjacent to a central sphere of radius 1) up to 4 dimensions, using a GO formulation proposed in

[3] Also see Chapter 5, which is an in-depth analysis of the SobolOpt algorithm.

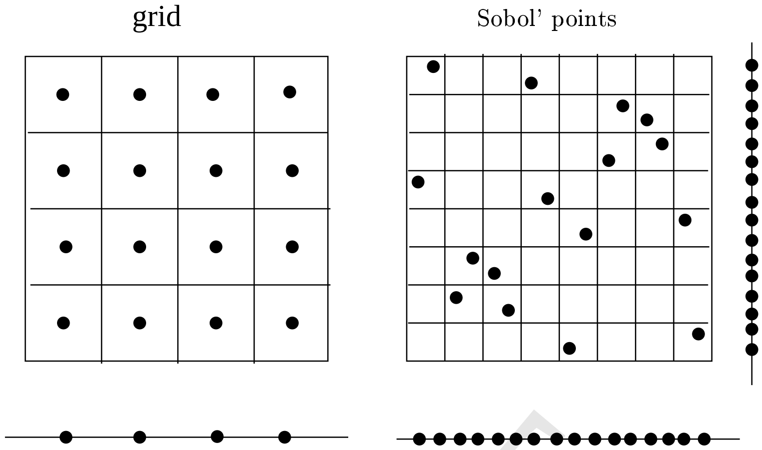


**Fig. 3.** Comparison between projected distribution of grid points and Sobol' points in $\mathbb{R}^2$.

[43]. The kissing number in 3 dimensions was first conjectured by Newton to be equal to 12. Newton was proven to be right only 250 years later by Leech. The 4D case was only very recently proved to be equal to 24 (a result by O. Musin, still unpublished). The 5D case is still open. Unfortunately the problem formulation for the 5D case is too large to be solved by the SobolOpt algorithm. Research in this field is ongoing.

A computational comparison of SobolOpt versus a spatial Branch-and-Bound algorithm has been carried out and discussed in [41], showing that SobolOpt performs well in box-constrained as well as equation and inequality-constrained NLP problems. Some positive results have been obtained even for modestly-sized MINLPs with a few integer variables[4].

### 2.2 Variable Neighbourhood Search

Variable Neighbourhood Search[5] (VNS) is a relatively recent metaheuristic which relies on iteratively exploring neighbourhoods of growing size to identify better local optima [29, 28, 27]. More precisely, VNS escapes from the current local minimum $x^*$ by initiating other local searches from starting points sampled from a neighbourhood of $x^*$ which increases its size iteratively until a local minimum better than the current one is found. These steps are repeated until a given termination condition is met. VNS is a combination of both the sampling and the escaping approaches, and has been applied to a wide variety of problems both from combinatorial and continuous optimization. Its

[4] Also see Chapter 5, Section 4 for computational experiments with the SobolOpt algorithm.
[5] Also see Chapters 6, 11 (Section 1.1).

early applications to continuous problems were based on a particular problem structure. In the continuous location-allocation problem the neighbourhoods were defined according to the meaning of problem variables (assignments of facilities to customers, positioning of yet unassigned facilities and so on) [12]. In the bilinearly constrained bilinear problem the neighbourhoods took advantage of a kind of successive linear programming approach, where the problem variables can be partitioned so that fixing the variables in either set yields a linear problem; the neighbourhoods of size $k$ were then defined as the vertices of the LP polyhedra that were $k$ pivots away from the current vertex. In short, none of the early applications of VNS was a general-purpose one.

The first VNS algorithm targeted at problems with fewer structural requirements, namely, box-constrained NLPs, was given in [52][6] (the paper focuses on a particular class of box-constrained NLPs, but the proposed approach is general). Since the problem is assumed to be box-constrained, the neighbourhoods arise naturally as hyperrectangles of growing size centered at the current local minimum $x^*$.

1. Set $k \leftarrow 1$, pick random point $\tilde{x}$, perform local descent to find a local minimum $x^*$.
2. Until $k > k_{\max}$ repeat the following steps:
   a) define a neighbourhood $N_k(x^*)$;
   b) sample a random point $\tilde{x}$ from $N_k(x^*)$;
   c) perform local descent from $\tilde{x}$ to find a local minimum $x'$;
   d) if $x'$ is better than $x^*$ set $x^* \leftarrow x'$ and $k \leftarrow 1$; go to step 2;
   e) set $k \leftarrow k + 1$

In the pseudocode algorithm above, the termination condition is taken to be $k > k_{\max}$. This is the most common behaviour, but not the only one (the termination condition can be based on CPU time or other algorithmic parameters). The definition of the neighourhoods may vary. If $N_k(x)$ is taken to be a hyperrectangle $H_k(x)$ of size. $k$ centered at $x$, sampling becomes easy; there is a danger, though, that sampled points will actually be inside a smaller hyperrectangular neighbourhood. A way to deal with this problem is to take $N_k(x) = H_k(x) \backslash H_{k-1}(x)$, although this makes it harder to sample a point inside the neighbourhood.

Some work is ongoing to implement a modification of the VNS for GO so that it works on problems in the more general form (1) (at least when $Z = \emptyset$). This was obtained by replacing the box-constrained local descent algorithm in step (2c) with an SQP algorithm capable of locally solving constrained NLPs.

[6] Also see Chapter 6, which is an in-depth analysis of the implementation of the VNS algorithm for box-constrained global optimization in [52], together with a presentation of computational results.

### 2.3 Spatial Branch-and-Bound

Spatial Branch-and-Bound (sBB) algorithms are the extension of traditional Branch-and-Bound (BB) algorithms to continuous solution spaces. They are termed "spatial" because they successively partition the Euclidean space where the problem is defined into smaller and smaller regions where the problem is solved recursively by generating converging sequences of upper and lower bounds to the objective function value. Traditional BB algorithms are used for finding the optimal solution of MILP problems. They work by generating subproblems where some of the integer variables are fixed and the others are relaxed, thus yielding an LP, which is easier to solve. Eventually, the solution space is explored exhaustively and the best local optimum found is shown to be the optimal solution.

Central to each sBB algorithm is the concept of a *convex relaxation* of the original nonconvex problem; this is a convex problem whose solution is guaranteed to provide an underestimation for the objective function optimal value in the original problem. At each iteration of the algorithm, restrictions of the original problem and its convex relaxations to particular sub-regions of space are solved, so that a lower and an upper bound to the optimal value of the objective function can be assigned to each sub-region; if the bounds are very close together, a global optimum relative to the subregion has been identified. The particular selection rule of the sub-regions to examine makes it possible to exhaustively explore the search space rather efficiently.

Most sBB algorithms for the global optimization of nonconvex NLPs conform to the following general framework:

1. (Initialization) Initialize a list of regions to a single region comprising the entire set of variable ranges. Set the convergence tolerance $\varepsilon > 0$, the best current objective function value as $U := \infty$ and the corresponding solution point as $x^* := (\infty, \ldots, \infty)$. Optionally, perform optimization-based bounds tightening (see Section 7.3) to try to reduce the variable ranges.
2. (Choice of Region) If the list of regions is empty, terminate the algorithm with solution $x^*$ and objective function value $U$. Otherwise, choose a region $R$ (the "current region") from the list according to some rule (a popular choice is: choose the region with lowest associated lower bound). Delete $R$ from the list. Optionally, perform feasibility-based bounds tightening on $R$ (see Section 7.3) to try to further reduce the variable ranges.
3. (Lower Bound) Generate a convex relaxation of the original problem in the selected region $R$ and solve it to obtain an underestimation $l$ of the objective function with corresponding solution $\bar{x}$. If $l > U$ or the relaxed problem is infeasible, go back to step 2.
4. (Upper Bound) Solve the original problem in the selected region with a local minimization algorithm to obtain a locally optimal solution $x'$ with objective function value $u$.

5. (Pruning) If $U > u$, set $x^* = x'$ and $U := u$. Delete all regions in the list that have lower bounds bigger than $U$ as they cannot possibly contain the global minimum.
6. (Check Region) If $u - l \leq \varepsilon$, accept $u$ as the global minimum for this region and return to step 2. Otherwise, we may not yet have located the region global minimum, so we proceed to the next step.
7. (Branching) Apply a branching rule to the current region to split it into sub-regions. Add these to the list of regions, assigning to them an (initial) lower bound of $l$. Go back to step 2.

The first paper concerning continuous global optimization with a BB algorithm dates from 1969 [22]. In the 1970s and 1980s work on continuous or mixed-integer deterministic global optimization was scarce. Most of the papers published in this period dealt either with applications of special-purpose techniques to very specific cases, or with theoretical results concerning convergence proofs of BB algorithms applied to problems with a particular structure. In the last decade three sBB algorithms for GO appeared, targeted at constrained NLPs in form (1).

- The Branch-and-Reduce algorithm, by Sahinidis and co-workers [62, 63], which was then developed into the BARON software (see below), one of the best sBB implementations around to date (see Section 3.1).
- The $\alpha$BB algorithm, by Floudas and co-workers [9, 7, 3, 1, 5, 2], that addressed problems of a slightly less general form than (1). This algorithm was also implemented in a software which was never widely distributed (see Section 3.2).
- The sBB algorithm with symbolic reformulation, by Smith and Pantelides [71, 75]. There are two implementations of this algorithm: an earlier one which was never widely distributed, called GLOP (see Section 3.3), and a recent one which is part of the $oo\mathcal{OPS}$ system, and for which development is still active.

All three algorithms derive lower bounds to the objective function by solving a convex relaxation of the problem. The main algorithmic difference among them (by no means the only one) is the way the convex relaxation is derived, although all three rely on the symbolic analysis of the problem expressions.

The list above does not exhaust all of the sBB variants that appeared in the literature fairly recently. As far as we know, however, these were the main contributions targeted at general-purpose GO, and whose corresponding implementations were undertaken with a software-design driven attitude aimed at producing a working software package. Other existing approaches for which we have no information regarding the implementation are Pistikopoulos' Reduced Space Branch-and-Bound approach [17] (which only applies to continuous NLPs), Grossmann's Branch-and-Contract algorithm [86] (which also only applies to continuous NLPs) and Barton's Branch-and-Cut framework [32]. We do not include interval-based sBB techniques here because their performance is rather weak compared to the algorithms cited above, where the lower bound is obtained with a convex relaxation of the problem. Interval-based sBB algorithms are mostly used with problems where the objective function and constraints are difficult to evaluate or inherently very badly scaled.

#### Convex relaxation

It is very difficult to devise an automatic method[7] for generating tight convex relaxations. Thus, many algorithms targeted at a particular class of problems employ a convex relaxation provided directly by the user — in other words, part of the research effort is to generate a tight convex relaxation for the problem at hand. The standard automatic way to generate a convex relaxation consists in linearizing all nonconvex terms in the objective function and constraints and then replacing each nonconvex definition constraint with the respective upper concave and lower convex envelopes. More precisely, each nonconvex term is replaced by a linearization variable $w$ (also called added variable) and a defining constraint $w =$ nonconvex term. The linearized NLP is said to be in *standard form*. The standard form is useful for all kinds of symbolic manipulation algorithms, as the nonconvex terms are all conveniently listed in a sequence of "small" constraints which do not require complex tree-like data structures to be stored [71, 39].

The defining constraint is then replaced by a constraint

$$\text{lower convex envelope} \leq w \leq \text{upper concave envelope.}$$

Since it is not always easy to find the envelopes of a given nonconvex term, slacker convex (or linear) relaxations are often employed. This approach to linearization was first formalized as an automatic algorithm for generating convex relaxations in [75], and implemented in the the GLOP software (see Section 3.3). The approach used by BARON is similar (see Section 3.1). The $\alpha$BB code avoids this step but is limited to solving problems in a given form (which, although very general, is not as general as (1)).

The downside to using linearization for generating the convex relaxation is that the standard form is a lifting reformulation, that is, the number of variables in the problem is increased (one for each nonconvex term). Furthermore, even if envelopes (i.e. tightest relaxations) are employed to over- and underestimate all the nonconvex terms, the resulting relaxation is not guaranteed to be the tightest possible relaxation of the problem. Quite on the contrary, relaxations obtained automatically in this way may in fact be very slack.

Common underestimators for bilinear [50, 8], trilinear, fractional, fractional trilinear, [5], convex univariate, concave univariate ([75], Appendix A.4) and piecewise convex and concave univariate terms [45] are all found in the literature.

[7] In fact, the work presented in Chapter 7 can also be seen as a first step in this direction, providing automatic symbolic techniques to verify the convexity properties of a given optimization problem and to generate new convex problems.

## 3 Global Optimization software

This section is a literature review on the existing software for GO. The review does not include software targeted at particular classes of problems, focusing on general-purpose implementations instead.

### 3.1 BARON

The BARON software (BARON stands for "Branch And Reduce Optimization Navigator"), written by Sahinidis and co-workers, implements a sBB-type algorithm called Branch-and-Reduce (because it makes extensive use of range reduction techniques both as a preprocessing step and at each algorithmic iteration) first described in [62, 63]. BARON aims at solving factorable nonconvex MINLPs. At the outset (1991), BARON was first written in the GAMS modelling language [11]. It was then re-coded in Fortran in 1994 and successively in a combination of Fortran and C for a more efficient memory management in 1996. The code was enriched in the number of local solvers during the years, and put online until around 2002, when it was decided that it would be distributed commercially as a MINLP solver for the GAMS system. Nowadays it can be purchased from GAMS (`www.gams.com`); for evaluation purposes, it is possible to download the whole GAMS modelling language, together with all the solvers, and run it in demo mode without purchasing a license. Unfortunately, the demo mode for global optimization solvers is limited to solving problems of up to 10 variables.

BARON and the Branch-and-Reduce algorithm it implements are further described in [66, 82, 83]. The documentation of the GAMS software contains a document about the usage of the BARON solver within GAMS. BARON is currently regarded as the state of the art implementation for a sBB solver, and in this author's experience the praise is wholly deserved.

The main feature in this Branch-and-Bound implementation is the range reduction technique employed before and after solving the lower and the upper bounding problems. Because range reduction allows for tighter convex underestimators, the algorithm has the unusual feature that a subproblem can be solved many times in the same node. As long as the range reduction techniques manage to find reduced ranges for at least one variable, the convexification on the same node becomes tighter and the variable bounds in both lower and upper bounding problems change. The subproblems are then solved repeatedly until a) the range reduction techniques fail to change the variable bounds or b) the number of times a subproblem is allowed to be solved in the same node reaches a pre-set limit. Condition (b) is a fail-safe device to prevent slow convergence cases from stopping the algorithm altogether. Let $P$ is the original problem and $R$ its convex relaxation. Let $L$ be a lower bound for the objective function of $R$ and $U$ an upper bound for the objective function of $P$. If the constraint $x_j - x_j^U \leq 0$ is active at the solution (i.e. the solution has

$x_j = x_j^U$ with Lagrange multiplier $\lambda_j^* > 0$) and if $U - L < \lambda_j^*(x_j^U - x_j^L)$, then increase the variable lower bound:

$$x_j^L := x_j^U - \frac{U-L}{\lambda_j^*}.$$

Similarly, if the constraint $x_j^L - x_j \le 0$ is active at the solution with Lagrange multiplier $\mu_j^* > 0$ and if $U - L < \mu_j^*(x_j^U - x_j^L)$, then decrease the variable upper bound:

$$x_j^U := x_j^L + \frac{U-L}{\mu_j^*}.$$

The geometrical interpretation of these range reduction tests is illustrated in Figure 4. It can be seen that the changes of the bounds effected by these rules do not exclude feasible solutions of $P$ with objective function values which are lower than $U$. Even if the variable bounds are not active at the

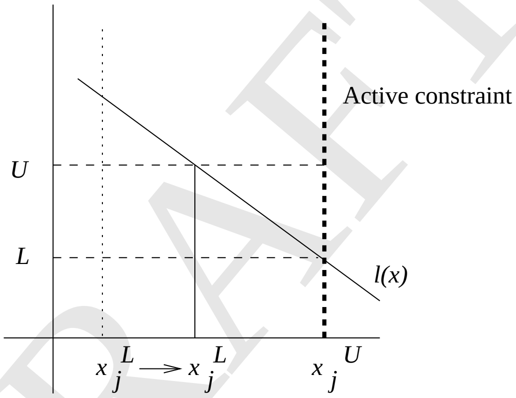


**Fig. 4.** Range reduction test. $l(x)$ is the straight line $\lambda_j^* x_j + L$.

solution, it is possible to "probe" the solution by fixing the variable value at one of the bounds, solving the partially restricted relaxed problem and checking whether the corresponding Lagrange multiplier is strictly positive. If it is, the same rules as above apply and the variable bounds can be tightened.

Another range reduction test is as follows: suppose that the constraint $\bar{g}_i(x) \le 0$ (where $\bar{g}$ is the relaxed convex underestimator for the original problem constraint $g$) is active at the solution with a Lagrange multiplier $\lambda_i^* > 0$. Let $U$ be an upper bound for the original problem $\mathbf{P}$. Then the constraint

$$\bar{g}_i(x) \ge -\frac{U-L}{\lambda_i^*}$$

does not exclude any solutions with objective function values better than $U$ and can be added as a "cut" to the current formulation of $\mathbf{R}$ to tighten it further.

As explained in Section 2.3, the lower bound to the objective function in each region is obtained by solving a convex relaxation of the problem. The techniques used by BARON to form the nonlinear convex relaxation of factorable problems are based on a symbolic analysis of the form of the factorable function. Each nonconvex term is analysed iteratively and then the convexification procedure is called recursively on each nonconvex sub-term. This approach to an automatic construction of the convex relaxation was first proposed in [75] (also see the implementation notes for the sBB solver in Section 7.3), and makes use of a tree-like data structure for representing mathematical expressions (see Section 5). The novelty of the BARON approach, and one of the main reasons why BARON works so well, is that it employs very tight linear relaxations for most nonconvex terms.

In particular, convex and concave envelopes are suggested for various types of fractional terms, based on the theory of convex extensions [80, 81]. The proposed convex underestimator for the term $\frac{x}{y}$, where $x \in [x^L, x^U]$ and $y \in [y^L, y^U]$ are strictly positive, is as follows:

$$\left.\begin{array}{c} z \geq \frac{x^L}{y_a}(1-\lambda) + \frac{x^U}{y_b}\lambda \\ y^L \leq y_a \leq y^U \\ y^L \leq y_b \leq y^U \\ y = (1-\lambda)y_a + \lambda y_b \\ x = x^L + (x^U - x^L)\lambda \\ 0 \leq \lambda \leq 1 \end{array}\right\} \tag{2}$$

The underestimator is modified slightly when $0 \in [x^L, x^U]$:

$$\left.\begin{array}{c} z \geq \frac{x^L(y^L+y^U-y_a)}{y^L y^U}(1-\lambda) + \frac{x^U}{y_b}\lambda \\ y^L \leq y_a \leq y^U \\ y^L \leq y_b \leq y^U \\ y = (1-\lambda)y_a + \lambda y_b \\ x = x^L + (x^U - x^L)\lambda \\ 0 \leq \lambda \leq 1 \end{array}\right\} \tag{3}$$

It is shown that these underestimators are tighter than all previously proposed convex underestimators for fractional terms, in particular:

- the bilinear envelope:

$$\max\left\{\frac{xy^U - yx^L + x^L y^U}{(y^U)^2}, \frac{xy^L - yx^U + x^U y^L}{(y^L)^2}\right\}$$

- the nonlinear envelope:

$$\frac{1}{y}\left(\frac{x+\sqrt{x^L x^U}}{\sqrt{x^L}+\sqrt{x^U}}\right)^2 .$$

The above convex nonlinear underestimators are then linearized using an outer approximation argument.

Furthermore, there is a specific mention of piecewise convex and piecewise concave univariate terms (called *concavoconvex* by the authors) and the respective convex and concave envelopes [83]. The convexification of this type of nonconvex term — an example of which is the term $x^3$ when the range of $x$ includes 0 — presents various difficulties, and it is usually not catered for explicitly (see [45] for a detailed study). An alternative to this envelope is suggested which circumvents the issue: by branching on the concavoconvex variable at the point where the curvature changes (i.e. the point where the concavoconvex term changes from concave to convex or vice versa) at a successive Branch-and-Bound iteration, the term becomes completely concave and completely convex in each region.

Other notable features of the BARON software include: generating valid cuts during pre-processing and and during execution; improving the branching scheme by using implication lists which for each nonconvex term point out the variable which most contributes to its nonconvexity [66, 83, 82, 64]; and most importantly, targeting particular problem formulations with specialized solvers [83]. These are available for:

- mixed-integer linear programming;
- separable concave quadratic programming;
- indefinite quadratic programming;
- separable concave programming;
- linear multiplicative programming;
- general linear multiplicative programming;
- univariate polynomial programming;
- 0-1 hyperbolic programming;
- integer fractional programming;
- fixed charge programming;
- problems with power economies of scale;

besides the "default" solver for general nonconvex factorable problems. BARON runs as a solver of GAMS; therefore, it runs on all architectures where GAMS can run.

### 3.2 $\alpha$BB

The $\alpha$BB algorithm [9, 6, 3, 2] solves problems where the expressions in the objective function and constraints are in factorable form. The convex relaxation of general twice-differentiable nonconvex terms is carried out by using a quadratic underestimation (based on the $\alpha$ parameter, which gives the name to the algorithm). Quadratic underestimations work for any twice-differentiable nonconvex term, but are usually very slack. In order to relax the original problem to a tight convex underestimator, the "usual" convex underestimators are proposed for bilinear, trilinear, fractional, fractional trilinear and concave univariate terms [5].

A function $f(x)$ (where $x \in \mathbb{R}^n$) is underestimated over the entire domain $[x^L, x^U] \subseteq \mathbb{R}^n$ by the function $L(x)$ defined as follows:

$$L(x) = f(x) + \sum_{i=1}^{n} \alpha_i (x_i^L - x_i)(x_i^U - x_i)$$

where the $\alpha_i$ are positive scalars that are sufficiently large to render the underestimating function convex. A good feature of this kind of underestimator is that, unlike other underestimators, it does not introduce any new variable or constraint, so that the size of the relaxed problem is the same as the size of the original problem regardless of how many nonconvex terms it involves. Since the sum $\sum_{i=1}^{n} \alpha_i (x_i^L - x_i)(x_i^U - x_i)$ is always negative, $L(x)$ is an underestimator for $f(x)$. Furthermore, since the quadratic term is convex, all nonconvexities in $f(x)$ can be overpowered by using sufficiently large values of the $\alpha_i$ parameters. From basic convexity analysis, it follows that $L(x)$ is convex if and only if its Hessian matrix $H_L(x)$ is positive semi-definite. Notice that:

$$H_L(x) = H_f(x) + 2\Delta$$

where $\Delta \equiv \mathrm{Diag}_{i=1}^n(\alpha_i)$ is the matrix with $\alpha_i$ as diagonal entries and all zeroes elsewhere (diagonal shift matrix). Thus the main focus of the theoretical studies concerning all $\alpha$BB variants is on the determination of the $\alpha_i$ parameters. Some methods are based on the simplifying requirement that the $\alpha_i$ are chosen to be all equal (uniform diagonal shift matrix), others reject this simplification (non-uniform diagonal shift matrix). Under the first condition, the problem is reduced to finding the parameter $\alpha$ that makes $H_L(x)$ positive semi-definite. It has been shown that $H_L(x)$ is positive semi-definite if and only if:

$$\alpha \geq \max\{0, -\frac{1}{2} \min_{i, x^L \leq x \leq x^U} \lambda_i(x)\}$$

where $\lambda_i(x)$ are the eigenvalues of $H_f(x)$. Thus the problem is now of finding a lower bound on the minimum eigenvalue of $H_f(x)$. The most promising method to this end seems to be Interval Matrix Analysis. Various $o(n^2)$ and $o(n^3)$ methods have been proposed to solve both the uniform and the non-uniform diagonal shift matrix problem [20].

The $\alpha$BB code itself is unfortunately not publicly distributed; refer to `http://titan.princeton.edu` for further details. The $\alpha$BB code is provided with a front-end parser module which is accepts mathematical expressions and generates corresponding "code lists". The parser is capable of reading certain types of quantifications in enumeration, summations and products, but is not equivalent to a full-fledged modelling language like AMPL or GAMS. Following parsing, automatic differentiation is applied to the code lists to generate the first and second order derivatives. Code lists for the lower bounding problem are also automatically generated. The main sBB iteration loop can then be started. $\alpha$BB runs on Unix architectures.

### 3.3 GLOP

GLOP is the implementation of the spatial Branch-and-Bound with symbolic reformulation proposed in [71, 73, 75]. This was the first sBB algorithm which generated convex relaxations automatically for optimization problems in arbitrary form (1), using the linearization method explained in Section 2.3. Other distinctive features of this algorithm are optimization and feasibility-based range reduction procedures. The sBB solver in $oo\mathcal{OPS}$ is based around a sBB derived directly from this algorithm. More information can be found in Section 7.3.

The GLOP code is not distributed. Extensive information about this software can be found in [71]. A brief description of this software is given here for the following reasons:

- Some of the ideas for $oo\mathcal{OPS}$ were borrowed from GLOP. More precisely, the software design and architecture are different, but the sBB algorithm implemented in GLOP is the basis of the sBB algorithm in $oo\mathcal{OPS}$.
- GLOP makes an interesting case study for an advanced software design, with solvers calling each other, implemented with programming techniques of about a decade ago. The first implementation was carried out mostly in C, with some of the local solvers being coded in Fortran.
- GLOP was experimentally inserted in the integrated software environment gPROMS for process synthesis [16]; in particular, a parallel version (which can be called from gPROMS) was coded (in Modula-2). No other sBB algorithm targeted at problems in form (1) ever had a working parallel implementation, to the best of our knowledge.
- GLOP is one of the few general-purpose sBB codes which can generate nonlinear, as well as linear, convex relaxation. Usually, the trade-off between tightness of convex relaxation and speed of solution is won by the latter; thus, most sBB implementations only produce linear relaxation. There are cases, however, where having a very tight convex relaxation (which may be nonlinear) is advantageous.

GLOP uses the binary tree data structure to manipulate mathematical expressions symbolically (see Section 5.1). Initially, GLOP reads a problem definition file in a pre-parsed proprietary format which is basically a text description of the binary trees representing the mathematical expressions in the problem, as well as the other numerical data defining the problem. Derivatives are computed symbolically, and some degree of symbolic simplification is enforced. The user can select whether a nonlinear or a linear convex relaxation is desired, as well as the local upper and lower bounding solvers to use. The selection includes CPLEX and MINOS for the lower bounding solver, and CONOPT, together with various experimental local NLP codes, for the upper bounding solver. E. Smith, the author of the software, also designed and implemented a local NLP solver based on successive linear programming.

GLOP was not written according to object-oriented software design principles. Some of the overall algorithmic control is carried out using global variables. Some care has been paid to the global variables not interfering with parallel execution. However, this is not fully re-entrant software design. As a stand-alone software, GLOP runs on Unix architectures. It was tested on Solaris and Linux. As part of gPROMS, it runs on all architectures where gPROMS runs.

### 3.4 $oo\mathcal{OPS}$

$oo\mathcal{OPS}$ stands for object-oriented OPtimization System. It is a software framework for doing optimization. As such, it contains a parser module, various reformulator modules, and various global and local solver modules; it has full symbolic manipulation capabilities (with mathematical expressions represented by binary trees) and is designed to tackle large scale global optimization problems. Its software design is such that the code is fully re-entrant. The architecture makes it possible for a number of modules to configure the solver parameters, even at different stages of the solution process. Its parser is interfaced with AMPL so that a full-fledged modelling language can be used to input problems. The parser and the rest of the systems are separated, so that $oo\mathcal{OPS}$ can actually be used as an external library (with a well documented API [48]).

$oo\mathcal{OPS}$ was designed as the environment where an advanced sBB solver code based on the spatial Branch-and-Bound with symbolic reformulation (see Section 3.3) should have been executed. As such, solvers are largely interchangeable, in the sense that they all bind to the same set of library calls. Technically, it is even possible to call sBB itself as the local solver of another sBB instance. With time, more global solvers were added to $oo\mathcal{OPS}$, so that now it can be considered as a general-purpose global optimization software of good quality, implementing sBB, MLSL and VNS algorithms targeted at solving MINLPs in form (1). $oo\mathcal{OPS}$ is fairly reliable, and was tested on a number of different problem classes (bilinear pooling and blending problems, Euclidean location-allocation problems, the Kissing Number problem, molecular distance geometry problems, and various other problem classes and instances) with considerable success. $oo\mathcal{OPS}$ consists of over 40000 lines of C++, in addition to the code of several local solvers (SNOPT [24], UCF from the NAG library [53], `lp_solve` [10]. The distribution package of $oo\mathcal{OPS}$ contains all the source code for compiling the system. It was decided, at this stage, to interface only to solvers whose source code is available (hence the prominent exclusion of CPLEX from the list of local solvers). *oo*$\mathcal{OPS}$ can be linked as a static executable including all the solvers, or as a dynamically-linked executable which loads the available solver modules as needed.

The public distribution of *oo*$\mathcal{OPS}$ is at the moment still being discussed between Imperial College, who holds the rights to the code, and the author. *oo*$\mathcal{OPS}$ runs on most Unix architectures. It has been tested on Linux and Solaris; earlier versions had been produced to run on Windows, both under the CYGWIN environment and under the MS Visual C++ compiler, but maintenance of the Windows-based versions has been discontinued. *oo*$\mathcal{OPS}$ was mostly written by the author of this paper, but C. Pantelides, B. Keeping, P. Tsiakis, S. Kucherenko of CPSE, Imperial College, London all contributed to the software.

Sections 3.4 and 3.4 give some details about the inner working of *oo*$\mathcal{OPS}$. Please note that not all of the *oo*$\mathcal{OPS}$ system conforms to the guidelines given in Section 4 for writing a good optimization software; many ideas given in Section 4 were developed by considering the inefficiencies of the existing *oo*$\mathcal{OPS}$ implementation.

**Object classes**

*oo*$\mathcal{OPS}$ consists of 4 major classes of objects, each with its own interface.

1. The `ops` object.
   An `ops` object is a software representation of a problem problem. The corresponding interface provides the following functionality.
   - It allows problem objects to be constructed and modified in a structured manner.
   - It allows access to all numerical and symbolic information pertaining to the problem in structured, flat (unstructured) and standard form.
2. The `opssystem` object.
   This is formed by the combination of an `ops` object with a solver code. The corresponding interface provides the following functionality.
   - It allows the behaviour of the solver to be configured via the specification of any algorithmic parameters that the solver may support.
   - It permits the solution of the problem.
3. The `opssolvermanager` object.
   This corresponds to a particular solver and allows the creation of an `opssystem` object combining this solver with a given `ops` object. The corresponding interface provides the following functionality.
   - It allows the creation of many different `opssystem` objects, all of which have the same solver parameter configuration.
4. The `convexifiermanager` object.
   This embeds the convexification code. The corresponding interface provides the following functionality.

- It allows the creation of the convex relaxation of the problem (in *oo*$\mathcal{OPS}$ we only employ linear relaxations).
- It allows the on-the-fly update of the convex relaxation when the variable ranges change.

#### Control flow

In this section, we describe how the user code (also called the *client*) calls the objects described above to formulate and solve an optimization problem. This involves a number of steps.

1. Construction of the problem `ops` object.
2. Creation of an `opssolvermanager` for the solver code to be used.
3. Creation of the necessary `opssystem` (by passing the `ops` object created at step 1 to the `opssolvermanager` created at step 2).
4. Solution of the problem (via the `opssystem`'s `Solve()` method).
5. Solution data query (via the `ops` object's interface).

Note that the `opssystem`'s `Solve()` method places the solution values back in the `ops` object containing the original problem, so they can be recovered later by the user code (also called the *client* code) by using a variable query method.

### 3.5 Other GO software packages

There are a few other GO codes targeted at fairly large classes of NLPs and MINLPs, which we do not discuss in detail either because of availability issues, lack of stability or simply because they are "solvers" rather than "software packages".

- The LGO (Lipschitz Global Optimizer) solver, coded and marketed by János Pintér [55], only requires function evaluation (no derivatives) as external user-defined routines. Bounds on the objective function value are obtained through Lipschitz analysis, and then employed in a spatial Branch-and-Bound framework. Currently, several versions of this solver exist: the core library object (for both Windows and Unix operating systems), an integrated development environment for Windows, and solver engines for Microsoft Excel, GAMS, Mathematica and Matlab. See the website `http://www.pinterconsulting.com/l_s_d.html` for more information.
- The GlobSol solver, by Robert Kearfott [33], is based on interval arithmetics to compute upper and lower bounds for a spatial Branch-and-Bound framework. There is no integrated environment for this solver.
- The Coconut Environment. This is a large global optimization project which is ongoing at Universität Wien, headed by A. Neumaier. One of

the products of this project is a software framework for global optimization, which offers an API [70] for external linking, and includes local and global solvers. Unfortunately, the code is still too unstable to be used productively. See the project website at `http://www.mat.univie.ac.at/coconut-environment/`.

- Lindo API [79]. This is an optimization framework for programmers. It offers an API and several solvers, among which a global one. A limited version can be downloaded for evaluation purposes from the website `http://www.lindo.com`.
- MORON stands for MINLP Optimization and Reformulation Object-oriented Navigator. This project was started by the author of this paper in order to produce a GO code that could be released to the public. MORON is still very much work in progress, but it will offer substantial improvements over $oo\mathcal{OPS}$, the main one being much more advanced symbolic manipulation capabilities. MORON uses an $n$-ary tree representation for mathematical expressions (see Section 5.3), which makes it possible to simplify algebraic expressions much more effectively. The software architecture is mostly borrowed from $oo\mathcal{OPS}$ but the code is completely re-written from scratch.

## 4 Optimization software framework design

As has been remarked, a general-purpose GO software is a complex program requiring interactions among many software modules. One of the major difficulties, requiring a high degree of programming abstraction, is to be able to replace a given software module with another one of the same type but different mathematical properties. For example, as most GO algorithm call a local optimization procedure as a sub-algorithm, we may wish to replace the default local solver embedded in the GO solver with a sparse local solver when solving large-scale problems to global optimality. Thus the local solver should not be "hard-wired" in the GO solver, but rather be a pluggable software module. Suppose further that we know the problem to be a bilinear one: then we might wish to replace the standard pre-solver (*reformulator* module) with one which is more apt to the task. With this philosophy, nearly every step in the global solution of a problem is implemented as a pluggable software module. This calls for a *core* software module where every other module can be plugged in. The core module supplies the necessary Application Programming Interfaces (APIs) so that the different modules can be called using the same standard function calls.

This type of software design may, with a bit of tweaking, be implemented in almost every programming language. Its proper semantic domain, however, is within the object-oriented programming paradigm. We chose C++ [78] to write implementations of the proposed software design, as it is widely available on almost every hardware and Operating System platform.

### 4.1 Control Flow

In this software description we follow a top-down approach, so that the reader may find higher level requirements before reading about the consequent implementation choices at lower level. The first thing that is necessary to know about how a software works is the control flow.

At first, a file representation of problem (1) is read into memory, parsed, and transformed into a memory representation of the problem. This task is carried out by the *parser* module, and the memory representation of the problem is stored within the data fields of the core module, which exposes a common API for each solver module to retrieve the problem data. Subsequent to parsing, the first task is to apply a symbolic reformulator to the problem. The reformulator tries a number of standard symbolic manipulations to reduce the number of variables and constraints, as well as reformulating the problem to a simpler, or more convenient form, if required. Next, based on user request, the control is passed to the *solver* module. Each solver is embedded into a *wrapper* module which allows solver software written by different people with different ad-hoc APIs to be interfaced with the core module uniformly. Solver modules written natively for the core module may dispense with the wrapper and directly use the core module's API. Each solver may call other modules (reformulators or solvers) as required by the algorithm being implemented. Before the solver terminates its execution, it stores the optimal solution, the optimal objective function value and other information regarding algorithmic performance and solution reliability within the core module, so that this information can be accessed by other solvers. When the first-level solver (i.e. the solver that was called first) terminates, the solution is output to the user and program execution stops.

Fig. 5 shows the control flow of a first-level solver module requiring two different second-level solvers, one for the main problem and the other to be applied to an auxiliary problem derived from the main one by means of symbolic reformulation (for example, consider a sBB algorithm requiring local solutions to a nonconvex NLP problem and to its convex relaxation).

### 4.2 Data Flow

Since we are envisaging an object-oriented architecture and re-entrant coding, global variables may not be used. The data relating to the mathematical formulation of the problem (i.e. the equations in (1)) are stored in the core module, as has been remarked. In practice, these data are fields of a class called `Problem` (more or less equivalent to what the class `ops` is in the $oo\mathcal{OPS}$ software, see Section 3.4). Since the parser module builds the `Problem` object corresponding to the main problem, it must have read/write permissions on `Problem` objects. Consider also that some GO algorithms need to solve auxiliary problems at each step as part of the overall solution method (e.g. sBB solves a lower-bounding problem at each step); since the auxiliary problems

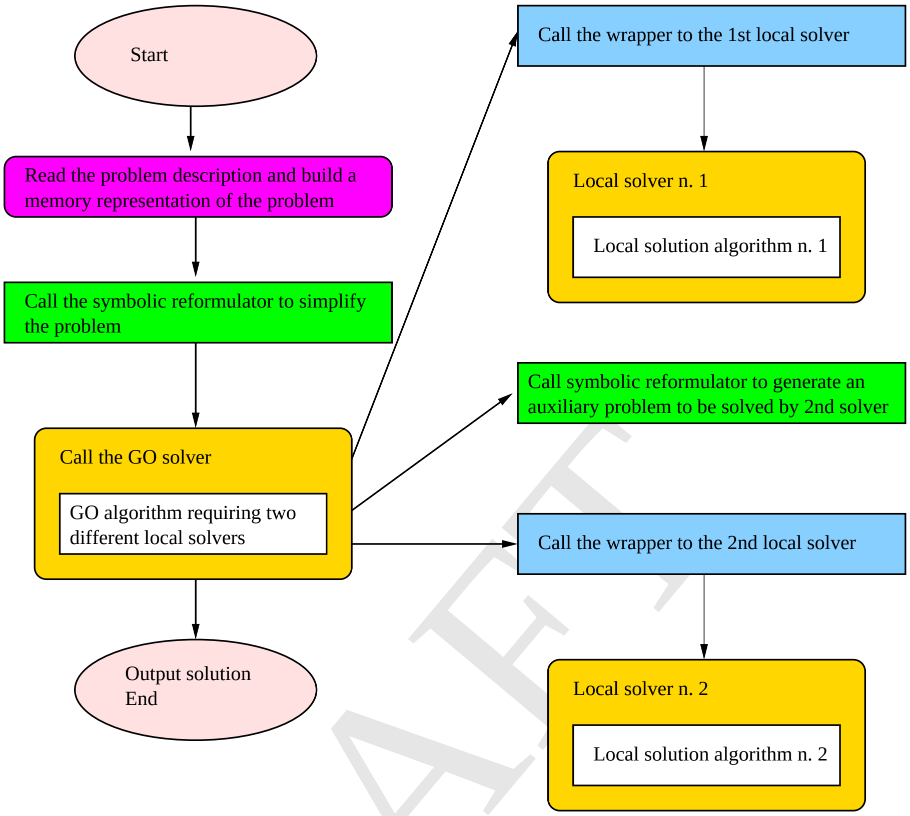


**Fig. 5.** Example of control flow.

may in principle be solved by any of the available solvers, it makes sense to embed them in an object of the `Problem` class. The reformulator module which generates the auxiliary problem must therefore have read/write permissions on `Problem` objects. Care must be taken to make the variable numbering consistent between the main problem and the derived auxiliary problems. Solvers need to read problem data rather than modify the problem, so they only need read access to `Problem` objects.

The `Problem` class, implementing the core module, is the fundamental class of the proposed GO framework. It stores information about variables, constraints and objective function. A variable has the following basic properties: integrality (whether it is an integer or a continuous variable), bounds and current value. A constraint has two basic properties: symbolic expression of the constraint and bounds. The objective function also has two basic properties, namely the symbolic expression it consists of, and the optimization direction (min or max). For simplicity, and without loss of generality, we shall thereafter assume that the optimization direction is that of minimization. The `Problem` class offers an API containing methods for performing the following actions:

- creation of problem entities
  1. create a new variable (including integrality, bounds and current value)
  2. create a new constraint (including constraint bounds)
  3. create the objective function (including optimization direction)
- output of problem data
  1. get problem sizes (number of variables, number of constraints, number of integer variables, number of linear constraints)
  2. get variable integrality, linearity, bounds and current values
  3. get constraint symbolic expression and bounds
  4. get the symbolic expression for the first (optionally second) order partial derivative of each constraint with respect to each variable
  5. get the symbolic expression and the optimization direction of the objective function
  6. get the symbolic expression of the first (optionally second) order partial derivative of the objective function with respect to each variable
- modification of problem data
  1. modify a variable (including integrality, bounds and current value)
  2. modify a constraint (including symbolic expression and bounds)
  3. modify the objective function
- output of dynamic information (i.e. depending on the current variable values)
  1. evaluate the constraint at the current variable values
  2. evaluate the constraint derivatives at the current variable values
  3. evaluate the objective function at the current variable values
  4. evaluate the objective function derivatives at the current variable values
  5. test whether current variable values are a feasible solution
  6. test whether a variable only occurs linearly in the problem or not
  7. test whether problem has a particular structure (linear, bilinear, convex)

The part of the API that tests wether the problem has a particular structure is called the symbolic analyser, and can also be considered as a separate module instead of being part of the `Problem` API. Testing whether a problem is linear or bilinear is easy; devising an algorithmic convexity test is far from trivial[8].

Fig. 6 shows an example of data flow in the proposed framework. The arrows represent the relationship "modifies the data of" between modules. It appears clear that certain reformulators change the core modules (`Problem` object), whilst others read the data from the core module to generate another

[8] Some work is being carried out on such a task; see Chapter 7.

core module for an auxiliary problem. Each solver wrapper has to internally store the data of the problem it is solving in order to pass these data to the solver proper in its required format. Figure 6 describes a sBB solver: the

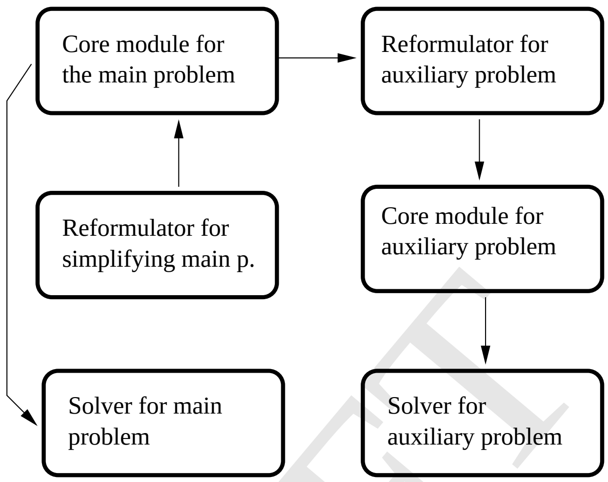


**Fig. 6.** Example of data flow. An arrow between modules $A$ and $B$ means that $A$ modifies the internal data of $B$.

core module for the main problem contains the data relative to the problem. This data is modified when the problem is simplified by the simplifying reformulator. The core module then loads its data in the main solver module (the global solver), which solves the problem. During the solution process, the solver module tells the core module to create a reformulator for generating a convex relaxation. The core module loads its data in the reformulator, which generates another core module for the auxiliary problem (the convex relaxation). The main solver then instructs the core module of the auxiliary problem to load its data in the auxiliary solver (the local solver), which solves the convex relaxation. The implementation of this sequence of data exchanges might be carried out so as to reduce the amount of transferred data.

### 4.3 Parser module

The ideal parser reads the mathematical formulation of an optimization problem, and transforms it in the appropriate data structures. Since optimization problems are often expressed in terms of quantifiers, indices and multidimensional sets of various kinds, to design such a parser is akin to crafting a modelling language, which is an extremely difficult and time-consuming task. Existing modelling languages (e.g. AMPL [19, 11]) usually provide an API for external solvers. If the API goes as far as providing symbolic information for the equations, we can design our software framework as a “super-solver” hooked to the modelling language software. The implementations described in this paper use AMPL as a modelling language. The documented AMPL API is very limited in its capabilities for passing structural problem information to solvers. However, AMPL also offers an undocumented API [21, 23] which makes it possible to read the mathematical expressions in the problem in a tree-like fashion. Both $oo\mathcal{OPS}$ and MORON work by using the undocumented AMPL API for building the internal data structures.

It is, of course, very easy to write a parser that reads optimization problems in flat form, i.e., where no quantifier appears in any of the problem expressions (for example, a constraint like $\forall i \in N (x_i + x_1 \leq 1)$ would be written in the flat form as a list of $|N|$ constraints where all the indices have been made explicit). Such parsers can be written by using standard programming tools like LEX and YACC [42] or from scratch: one good starting point can be found in the first chapters of [78]. Again, both $oo\mathcal{OPS}$ and MORON are equipped with such flat form parser modules. Interfacing with AMPL was actually carried out with a small external program which uses the undocumented AMPL API to produce a flat form representation of the problem that the optimization software can read.

### A standard MINLP description file format

The widespread adoption of a standard, flat-form problem definition file for MINLPs (akin to what the MPS and LP file formats are to linear programming) would be a very desirable event. AMPL is capable of producing `.nl` ASCII files which are complete descriptions of MINLP problems. However, they are so cryptic to the human eye that suggesting their adoption as standard would pose serious problems. Unfortunately, the same goes for the rather widespread SIF format [4], which also has more stringent limitations: for example, it is impossible to express arbitrary compositions of nonlinear functions.

$oo\mathcal{OPS}$ and MORON currently read flat-form definition files in the following format.

`#` a comment
`variables =`

$LowerBound_1 < VarName_1 < UpperBound_1 / VarType_1,$

$\vdots$

$LowerBound_n < VarName_n < UpperBound_n / VarType_n;$

`objfun =`

$[c^{\mathsf{T}} x + f(x)];$

`constraints =`

$[LowerBound_1 < a_1 x + g_1(x) < UpperBound_1],$

$\vdots$

$[LowerBound_m < a_m x + g_m(x) < UpperBound_m];$

`startingpoint =`

$x'_1, \ldots, x'_n;$

`options =`

$ParameterName_1 \ ParameterValue_1,$

$\vdots$

$ParameterName_k \ ParameterValue_k;$

The symbols are the same as in (1). Here, $a_1, \ldots, a_m$ are the rows of the matrix $A$. The symbol "$<$" has been employed instead of "$\leq$" because the text file is less cluttered with just one ASCII symbol (`<`) instead of two (`<=`), but the semantics of the (`<`) symbol is actually "less than or equal to". $VarType$ is a string (`Integer` or `Continuous`) describing the type of variable. Although objective function and constraints are separated in linear and nonlinear parts in the format description above, there is no reason why this task should not be performed directly by the software. Indeed, MORON separates linear and nonlinear parts automatically. $oo\mathcal{OPS}$, which relies on a less advanced symbolic manipulation library, requires the linear parts to be made explicit in the description file, with the special character "`|`" syntactically separating linear and nonlinear parts (with the semantics of "sum"). The nonlinear functions $f(x), g_1(x), \ldots, g_m(x)$ are strings describing the mathematical expressions. The only really non-trivial piece of code required for reading the above problem description file is therefore a small parser that, given a string containing a mathematical expression (containing just variable names, numbers, and operators, without indices or quantifiers), builds a binary (or $n$-ary) tree representing the expression. As has been remarked in Section 4.3, this task is actually fairly easy.

Below is an example of a problem expressed in the format described in this section.

```
# problem: Yuan 1988 (MINLP)
variables =
   0 < y1 < 1 / Integer,
```

```
   0 < y2 < 1 / Integer,
   0 < y3 < 1 / Integer,
   0 < x4 < 10 / Continuous,
   0 < x5 < 10 / Continuous;
objfun =
   [ 2*x4 + 3*x5 + 1.5*y1 + 2*y2 - 0.5*y3 ];
constraints =
   [1.25 < y1 + x4^2 < 1.25],
   [3 < 1.5*y2 + x5^1.5 < 3],
   [MinusInfinity < y1 + x4 < 1.6],
   [MinusInfinity < y2 + 1.3333*x5 < 3],
   [MinusInfinity <  -y1 - y2 + y3 < 0];
startingpoint =
   0, 0, 1, 1, 1;
options =
   MainSolver sBB,
   sBBLowerBoundSolver lp_solve,
   sBBUpperBoundSolver snopt;
```

Although a starting point is usually not required for most GO algorithms, it was decided to include the possibility of passing this information to the solver, since this standard might be used as input for a local NLP solver.

We suggest that a suitable extension of this file format (encompassing an arbitrary number of objective functions and other minor adjustments) should be used as a standard MINLP problem description file format.

### 4.4 Data structures

The most important data structures in the `Problem` class refer to the main problem entities: variables, objective function and constraints. The mathematical expressions defining objective function and constraints are stored symbolically in a tree-like fashion, as explained in Section 5 below. Many of the other properties of the problem entities, like names, bounds and so on, can be implemented as desired, as they do not pose any particular practical problem. $oo\mathcal{OPS}$ and MORON use the C++ STL `string` class [78] for names and the basic `double` type for real numbers.

The most puzzling issues in designing data structures for problem entities are to do with indexing of variables and constraints, and usually give rise to the worst bugs. Since many global optimization algorithms work by reformulating the problem symbolically, and since each reformulation is conveniently stored in a separate object, it is important to retain the identity of each variable and constraints, even when they undergo symbolic transformations. Variable IDs are the most critical pieces of information: a sBB algorithm has to act alternately on the original problem and on its convex relaxation. Since the convex relaxation may be the result of a complex symbolic manipulation (involving adding and removing problem variables as needed), a mapping between variables in the original and in the convexified problem is of the utmost importance. After much coding experimentation, the most practical solution seems to be the following:

- the original problem is first simplified as much as possible, and as many variables as possible are deleted from the problem;
- the variables in the simplified problem are flagged "unchangeable" to all subsequent reformulator modules;
- reformulators which perform liftings, i.e. which add variables to the problem, flag the added variables as "changeable", so that subsequent reformulators can remove them if required.

Each variable is assigned a unique ID when the problem is first read by the parser. These IDs are stored in lists. Deletion of a variable is equivalent to removing the corresponding ID from the list; no other variable is re-indexed (obviously, deleting a variable can only be done if the variable does not appear in any of the problem expressions). A variable can be added by finding an unassigned variable ID and inserting it into the list. Cycling over all variables is equivalent to traversing the list. Similar lists containing variable names, bounds, values and other properties are easily kept synchronized with the "basic" ID list.

Constraints can be dealt with similarly, although symbolic manipulation of constraints is usually much less problematic.

### 4.5 Configuration of solver parameters

The performance of most solver codes is usually hugely conditioned by proper parameter settings (like tolerances, limits on the number of iterations and so on). If a GO algorithm relies on a cascaded sequence of solvers being called, each solver must be properly configured; this, however, poses some problems, because whilst some configuration parameters can be set by the user, others are better left to the higher-level solver, which will set them based on the current performance of the algorithm run. Thus, solvers are first created in memory by a *solver manager*, which pre-configures them; subsequently, the solver and the `problem` object being solved are bound together in a *solver system*, which offers further configuration capabilities and finally offers the API for starting the solution process. A parameter list is stored both in the solver manager and in the solver system (which inherits a pre-configured parameter list from the solver manager). This treatment of parameter setting follows the guidelines of the Global CAPE-Open consortium for chemical engineering software [36, 57, 56].

## 5 Symbolic manipulation of mathematical expressions

Symbolic computation is usually something that the mathematical practitioner employs dedicated software for; software like Maple, Mathematica, Matlab and so on. Numerical methods, and in particular optimization, have always been taught in the spirit of number crunching, the most notable exception being linear algebra, for the simple reason that doing symbolic manipulation on linear expressions is very easy. It is difficult to even envisage how to write a symbolic computation program on a computer whose basic data types are integer and floating point numbers. The following books provide good introductions to symbolic computation methods [76, 15, 14].

Symbolic computation relies on a machine representation of mathematical operations on some numbers or literal symbols (constants, variables, or expressions involving constants and variables). Usually, one of the following techniques is employed to represent these operations:

- binary trees;
- lists;
- $n$-ary trees.

### 5.1 Binary Trees

Binary trees have been proposed as a way of representing mathematical expressions by Knuth [34] and made their way in computational engineering and other fields of scientific computing [13]. This representation is based on the idea that operators, variables and constants are nodes of a digraph; binary operators have two outcoming edges and unary operators only have one; leaf nodes have no outcoming edge. One disadvantage is that binary tree representation makes it cumbersome to implement associativity. For example, the expression $y + x + 2x + 3x$ is represented as $(((y + x) + 2x) + 3x)$, so it would require three recursive steps to lower tree ranks to find out that it is possible to write it as $(y+6x)$. Another disadvantage is that different parsers may have different representations for the same expressions. With the example above, a "left-hand-side-first" parser would create $(y + (x + (2x + 3x)))$ instead of the "right-hand-side-first" $(((y + x) + 2x) + 3x)$.

Where symbolic manipulation is only desired to compute symbolic derivatives and performing little or no symbolic manipulation, this approach may be the best, as it is simpler to implement than the other techniques and generally performs very efficiently [54, 48]. $oo\mathcal{OPS}$ uses binary trees to store expressions.

### 5.2 Lists

The representation of algebraic expressions by lists dates back to the AI-type languages Prolog and Lisp. Lisp, in particular, was so successful at the task that a lot of CASes, today, are still based on Lisp's list manipulation abilities. Prolog has some interesting features in conjunction with symbolic computation, in particular the "computation-reversing" ability, by which if you compute a symbolic derivative and do not bother to simplify it, Prolog lets you integrate it symbolically performing virtually no calculation at all.

Any symbolic computation library written in Prolog/Lisp faces the hard problem of implementing an API which can be used by procedural languages like Fortran, C or C++. Whilst technically not impossible, the architectures and OSes offering stable and compatible Lisp and C/C++ compilers are few. GNU/Linux actually has object-compatible Lisp and C/C++ compilers; however, the GNU Lisp compiler uses an array of internal data structures which are very difficult to read from a C/C++ program, making data interchange between the different modules hard to implement.

There are two other problems faced by Prolog/Lisp programs: portability (many Prolog/Lisp compilers implement different dialects of the languages) and a reduced user base.

### 5.3 $n$-ary Trees

Expression representation by $n$-ary trees can be seen as a combination of the previous two techniques. MORON makes use of this representation. In order to characterize this representation formally, we need some definitions.

An *operator* is a node in a directed tree-like graph. Let $L$ be the set

$$\{+, -, \times, /, \hat{}, (-1)\times, \log, \exp, \sin, \cos, \tan, \cot, \mathrm{VAR}, \mathrm{CONST}\}$$

of *operator labels*. An operator with label VAR is a *variable*, an operator with label CONST is a *constant*. Operator nodes may generally have any number of outcoming edges; variables and constants have no outcoming edges and are called *leaf nodes*. A variable is also characterized by a non-negative integer index $i$, and a constant by a value which is an element of a number field $F$. We shall assume $F = \mathbb{R}$ (or at least, a machine representation of $\mathbb{R}$) in what follows, but this can vary. Let $V$ be the set of all variable-type operator nodes. Let $T_0 = V \cup \mathbb{R}$. This is the set of the terminal (or leaf) nodes, i.e. the variables and constants. Now for each positive integer $i$, define recursively $T_i = L \times (T_{i-1} \cup T_0)^{<\omega}$. Elements of $T_i$ are operator nodes having *rank* $i$. Basically, an element of $T_i$ is made up of an operator label $l \in L$ and a finite number of subnodes. A *subnode* $s$ of $n$ is a node $s$ in the digraph so that there is an edge leaving node $n$ and entering $s$.

The biggest advantage of $n$-ary tree representation is that it makes it very fast and easy to perform expression simplification. Another advantage is that expression evaluation on $n$-ary trees is faster than that obtained with a binary tree structure [40].

### 5.4 Main symbolic manipulation algorithms

In this section we discuss the most important algorithms for symbolic manipulation: allocation and deallocation of memory for recursive data structures like trees, evaluation of an expression at a point, symbolic differentiation and basic simplification rules. Many more symbolic algorithms are actually used in both *oo*$\mathcal{OPS}$ and MORON, including a standard representation for mathematical expressions, equality tests, advanced simplification routines and separation of linear and nonlinear parts of an expression. The latter is particularly important as most nonlinear local solvers require the linear parts of the constraints to be input separately from the nonlinear parts.

#### Allocation and deallocation

One of the biggest challenges in tree handling is memory allocation/deallocation. A tree-node class normally consists of its own semantic data and two (or more, in the case of $n$-ary trees) references to its children subnodes. In this setup, the following memory-related problems arise.

- When a node is allocated, the subnodes are not automatically allocated, so they have to be allocated manually when the need arises (see fig. 7).

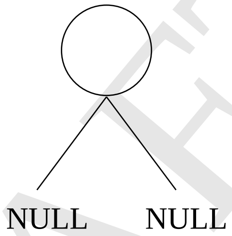


**Fig. 7.** Allocation of a node does not allocate subnodes.

- When a node is copied, the question arises whether all the subnode hierarchy should be copied or just the references to the immediate subnodes (see fig. 8). The two cases must be treated separately.
- Supposing a tree has been allocated, with some of its nodes copied hierarchically and some others just copied as references, how does one deallocate the tree? Just deleting all the nodes will not work because the nodes copied as references are still in use by other trees. A node cannot hold the information about all the trees it belongs to, as it may belong to a huge number of trees. The standard way to deal with this situation is to store a counter in each node that counts the number of times it is copied as a reference. Each time the deallocation of the node is requested the counter is decreased. The node is truly deallocated only when the counter is zero.

#### Evaluation

Normal mathematical operations on expressions amount to the manipulation of nodes. These can be copied to form new expressions, replaced by other nodes to achieve symbolic simplification and so on. Two expressions can be

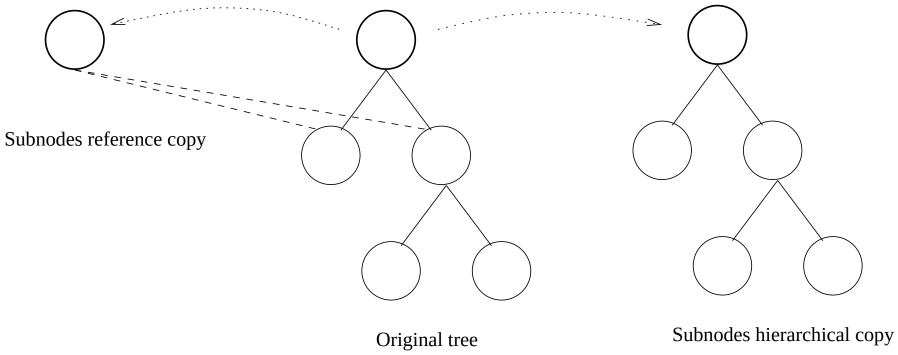


**Fig. 8.** Copy of subnodes and copy of references.

summed (or in fact acted on by any other binary operator) just by creating a new top node and setting the top nodes of the two expressions as its two subnodes. Substitution of a variable can be obtained by replacing the relevant variable indices in the terminal nodes; special care must be taken if these terminal nodes are in use in other trees within the program. If they are, then new terminal nodes should be created before the changes occur. Evaluation of an expression tree is obtained by recursing on each node in the following way.

```
evaluate(node, variable_values) {
  if (node is terminal) {
    return variable value corresponding to variable index;
  } else {
    result = 0;
    for each subnode in node {
      partial_result = evaluate(subnode, variable_values);
      result = result (node.operator) partial_result;
    }
    return result;
  }
}
```

Both `result` and `partial_result` in the above pseudocode indicate real numbers. If the same tree has to be evaluated several times, it may be convenient to store the linear order of recursive evaluation of each node and then call a modified evaluation procedure which is linear in nature rather than recursive. This avoids the computational overhead of recursiveness (although optimizing compilers reduce that overhead considerably nowadays). The details of this strategy are analysed in [35].

### Differentiation

Derivatives of the objective function and constraints are used (or can be used) by most nonlinear deterministic local solvers, so an efficient way to calculate these derivatives gains good computational savings. Ordinarily, derivatives are computed by finite differences or by hard-coding the symbolic derivative inside the program at compile-time. The first method has huge associated computational costs, whilst the second method only targets programs devised to solve one particular optimization problem, which is not suitable for our purposes.

There are two alternatives: automatic differentiation (AD) [25] and symbolic derivative computation. A. Griewank, in his fascinating 1989 paper on AD, claims that AD is better than symbolic differentiation. However, the limitation of the symbolic differentiation methods he was referring to were twofold: hardware-wise, in lack of RAM (he was using a Sun3 with 16MB ram); and software-wise, in having to pass the output of a symbolic derivative computed by Macsyma 1 or Maple 1 to a Fortran compiler before the evaluation, apparently with the overhead of shell-piping mechanisms. Besides, AD inherently has one very stringent limitation which makes it unsuitable for use in a software framework: namely, that AD algorithms automatically generate derivative evaluation *code*, which must be compiled before it can be useful. This would make the optimization framework dependent on a compiler and a linker, which is not usually an acceptable choice for a stand-alone software. For this reason we chose to employ symbolic differentiation techniques.

Once the symbolic derivatives are calculated for the objective function and constraints, they only need to be evaluated in order to produce the derivative value at a point. Other advantages of this approach are that (a) by using expression trees and recursive procedures, symbolic derivatives are not computationally expensive to construct; and (b) the derivative values they provide are exact, whereas finite difference methods can only approximate the values at best.

The following pseudocode shows how to construct a node representing the symbolic derivative of `node` with respect to `variable`.

```
diff(node, variable) {
  retnode = 0;
  for each subnode in node {
    if (subnode depends on variable) {
      if (subnode is terminal) {
        retnode = 1;
      } else {
        case of node.operator {
        case '+':
          retnode = retnode + diff(subnode, variable);
        case '-':
          retnode = retnode - diff(subnode, variable);
        case '*':
```

```
          retnode = retnode + diff(subnode, variable) * node / subnode;
        // and all the other derivative rules...
        }
      }
    }
  }
  return retnode;
}
```

Notice that `retnode` in the above pseudocode indicates a node, so all the operations that act on `retnode` (addition, subtraction, multiplication, division and so on) are to be implemented as procedures which manipulate expression trees.

Derivative rules are the usual ones; the rule for multiplication is expressed in a way that allows for $n$-ary trees to be correctly derived:

$$\frac{\partial}{\partial x}\prod_{i=1}^{n} f_i = \sum_{i=1}^{n}\left(\frac{\partial f_i}{\partial x}\prod_{j\neq i} f_j\right).$$

All valid algebraic simplifications can be used to simplify symbolic expressions. However, all simplifications have an associated computational cost so it is essential to find a balance between the degree of simplification of each expression and the cost of the simplification itself. Simplifying $\sin^2(f(x)) + \cos^2(f(x)) = 1$, for example, involves a tree search that spans 7 nodes (addition, exponentiation and "2" as a constant in two nodes, sine and cosine operators) and is of limited use, so it is not advisable to employ it unless it is known in advance that most of the expressions will involve sines and cosines.

#### Simplification

The basic simplifications which should be carried out are the following:

$$
\begin{aligned}
constant\ (operator)\ constant &= constant(resultofoperation) \\
-(-f(x)) &= f(x) \\
f(x) + 0 &= f(x) \\
f(x) - 0 &= f(x) \\
0 - f(x) &= -f(x) \\
f(x) \times 1 &= f(x) \\
\frac{f(x)}{1} &= f(x) \\
(f(x))^0 &= 1 \\
(f(x))^1 &= f(x) \\
f(x) + f(x) &= 2f(x) \\
f(x) - f(x) &= 0 \\
f(x) \times f(x) &= (f(x))^2 \\
\frac{f(x)}{f(x)} &= 1
\end{aligned}
$$

Note that binary trees are not commutative, so the commuted simplifications

$$
\begin{aligned}
0 + f(x) &= 0 \\
1 \times f(x) &= f(x)
\end{aligned}
$$

should also be carried out. Note also that after a tree has been simplified once there is scope for further simplification. For example applying the above rules in succession to $x^{2y-2y} - 1$ would gather $x^0 - 1$, but it is evident that the expression can be simplified even more. A second application of the rules would gather $1 - 1$ and a third application would finally gather 0. Ideally, thus, simplification should be carried out repeatedly until the expression does not change under the simplification rules. Where this is too computationally expensive, a compromise may be enforced.

## 6 Local solvers

By "local solvers" we mean here those solvers which decide on the local or global optimality of a point by performing an analysis of a point neighbourhood. As such, local solvers may implement local solution algorithms (for NLPs) and global solution algorithms (for LPs). Mostly, local solvers are used within global solvers as black-box calls to solve the main problem, or auxiliary problems, locally. In most GO algorithms, the global phase has a limited numerical knowledge of the problem structure; the "dirty work" is usually performed by the local solvers. In the case of LPs, the local solver needs to know the linear coefficients of all the variables in the objective and the constraints, as well as the variable and constraint bounds. In the case of NLPs, the local solver needs to evaluate objective function, constraints, first derivatives of both, and optionally also second derivatives, for any given point. Large-scale local NLP solvers need to be explicitly told about linear and nonlinear parts of each expression in the problem. As most local solvers are in fact "solver libraries", with varying degrees of user-friendliness, sometimes the problem variables and/or the constraints need to be re-ordered. This is very time-consuming and error-prone, as the inverse re-ordering needs to be applied to the solution vector.

The task of interfacing a local solver with the rest of the optimization system is carried out by the solver wrapper. In this author's experience, solver wrappers for existing local NLP solvers (specially those requiring variable and constraint re-ordering) are the most frustrating source of software bugs. Interfacing with local LP solvers is easier; however, since many local LP solvers do not accept constraints in "double bounded format" ($\mathit{LowerBound} \leq g(x) \leq \mathit{UpperBound}$), preferring the "single bounded format" instead ($g(x) \underset{\leq}{\overset{\geq}{=}} \mathit{Bound}$), some of the constraints might have to be replicated with different directions and bounds.

One word should be spent about the reliability of local solver codes. Whereas LP solvers are next to 100% reliable, some of the most efficient algorithms for the local solution of nonconvex NLPs are inherently unreliable. Sequential Quadratic Programming (SQP), for example, is a standard and widely used technique for locally solving constrained NLPs in general form. SQP might fail (rather spectacularly in certain cases) if a feasible starting point cannot be provided, or if the linearized constraints are infeasible (even though the original nonlinear constraints may be feasible). Both these occurrences are far from rare, so local NLP solvers are rarely reliable. Since most GO algorithms delegate the numerical work to the local solvers, a global solver is only as reliable as its local sub-solver, that is to say, not very reliable at all. Therefore, it is always a good idea for the wrapper to be able to deal properly with all the return messages of the local solver; in our opinion, it is also a good idea to have the wrapper double-check on the feasibility of the solution provided by the local solver.

$oo\mathcal{OPS}$ at the moment has three local solvers: NPSOL (or rather, the VCF optimization code in the NAG library), a rather old version of SNOPT (which is a large-scale modification of NPSOL), and `lp_solve`, which is a free LP solver.

MORON is interfaced to a recent version of SNOPT and the GLPK [49] local LP solver.

## 7 Global solvers

GO algorithms mostly require very high-level steps, like local solution of subproblems, symbolic manipulation of mathematical expressions, and so on.

Embedding global solvers within optimization environment which offer these possibilities makes it possible to implement and test a global solver in a very short time. In this section we shall describe the three global solvers found in $oo\mathcal{OPS}$. At this stage, MORON only has a very preliminary version of a sBB solver, which is not discussed here.

### 7.1 SobolOpt multistart algorithm

SobolOpt[9] (also see Section 2.1) is an implementation of a Multi-Level Single Linkage (MLSL) algorithm; its main strength is that it employs certain Low-Discrepancy Sequences (LDSs) of sampling points called Sobol' sequences whose distributions in Euclidean space have very desirable uniformity properties. Let $Q$ be the set of pairs of sampled points $q$ together with their evaluation $f(q)$ (where $f$ is the objective function). Let $S$ be the list of all local minima found up to now.

1. (Initialization) Let $Q = \emptyset$, $S = \emptyset$, $k = 1$ and set $\varepsilon > 0$.
2. (Termination) If a pre-determined termination condition is verified, stop.
3. (Sampling) Sample a point $q_k$ from a Sobol' sequence; add $(q_k, f(q_k))$ to $Q$.
4. (Clustering distance) Compute a distance $r_k$ (which is a function of $k$ and $n$; there are various ways to compute this distance, so this is considered as an "implementation detail" — one possibility is $r_k = \beta k^{-\frac{1}{n}}$, where $\beta$ is a known parameter and $n$ is the number of variables).
5. (Local phase) If there is no previously sampled point $q_j \in Q$ (with $j < k$) such that $||q_k - q_j|| < r_k$ and $f(q_j) \leq f(q_k) - \varepsilon$, solve problem (1) locally with $q_k$ as a starting point to find a solution $y$ with value $f(y)$. If $y \notin S$, add $y$ to $S$. Set $k \leftarrow k + 1$ and repeat from step 2.

The algorithm terminates with a list $S$ of all the local minima found. Finding the global minimum is then a trivial matter of identifying the minimum with lowest objective function value $f(y)$. Two of the most common termination conditions are (a) maximum number of sampled points and (b) maximum time limit exceeded. A discussion of how Sobol' sequences can be generated is beyond the scope of this paper. A good reference is [59], p.311.

The implementation of the SobolOpt algorithm, which is very robust, was carried out by S. Kucherenko and Yu. Sytsko and successively adapted to the $oo\mathcal{OPS}$ framework. For more details about this algorithm, see [37].



[9] The SobolOpt solver within $oo\mathcal{OPS}$ shares the same code as the implementation described in Chapter 5.

### 7.2 Variable Neighbourhood Search

In this section we discuss the implementation of the Variable Neighbourhood Search[10] algorithm for GO presented in Section 2.2. The search space is defined as the hypercube given by the set of variable ranges $x^L \leq x \leq x^U$. At first we pick a random point $\tilde{x}$ in the search space, we start a local search and we store the local optimum $x^*$. Then, until $k$ does not exceed a pre-set $k_{\max}$, we iteratively select new starting points $\tilde{x}$ in a neighbourhood $N_k(x^*)$ and start new local searches from $\tilde{x}$ leading to local optima $x'$. As soon as we find a local minimum $x'$ better than $x^*$, we update $x^* = x'$, re-set $k = 1$ and repeat. Otherwise the algorithm terminates.

For each $k \leq k_{\max}$ consider hyper-rectangles $R_k(x^*)$ similar to $x^L \leq x \leq x^U$, centered at $x^*$, whose sides have been scaled by $\frac{k}{k_{\max}}$. More formally, let $R_k(x^*)$ be the hyper-rectangle $y^L \leq x \leq y^U$ where, for all $i \leq n$:

$$
\begin{aligned}
y_i^L &= x_i^* - \frac{k}{k_{\max}}(x_i^* - x_i^L) \\
y_i^U &= x_i^* + \frac{k}{k_{\max}}(x_i^U - x_i^*).
\end{aligned}
$$

This construction forms a set of hyper-rectangular "shells" centered at $x^*$. For $k > 0$, we define the neighbourhoods $N_k(x^*)$ as $R_k(x^*) \backslash R_{k-1}(x^*)$ (observe that $R_0(x^*) = \emptyset$). The neighbourhoods are disjoint, which gives the VNS algorithm a higher probability not to fall in local optima that have already been located. Furthermore, the union of all the neighbourhoods $N_k(x^*)$ is the whole space $x^L \leq x \leq x^U$. This is a rather unusual features in VNS implementation, specially when VNS is applied to combinatorial problems. Here it is justified by the (partial) continuity of the search space. The neighbourhoods are obviously just used for sampling; the local search itself is performed in the whole space.

Sampling in the neighbourhoods $N_k(x^*)$ is a non-trivial task. Since sampling in hyper-rectangles is easy, one possible solution would be to sample a point in $R_k(x^*)$ and reject it if it is in $R_{k-1}(x^*)$, but this would be highly inefficient. A different strategy was preferred in this implementation.

1. Choose an index $j \leq n$ randomly.
2. Sample a random value $x'_j$ in the one-dimensional interval given by projection of $R_k(x^*)$ on the $j$-th coordinate.
3. Let the projection of $R_{k-1}(x^*)$ on the $j$-th coordinate be the interval $\rho = [y_j^L, y_j^U]$. If $x'_j \in \rho$, then: if $|x'_j - y_j^L| \leq |x'_j - y_j^U|$ let $x'_j = y_j^L$, else let $x'_j = y_j^U$.
4. For all $i \leq n$ such that $i \neq j$, sample a random value $x'_i$ in the projection of $R_k(x^*)$ on the $i$-th coordinate.

[10] The implementation of the VNS solver in $oo\mathcal{OPS}$ is fundamentally different from the GLOB implementation described in Chapter 6.

The above procedure generates a point $x' = (x'_1, \ldots, x'_n)$ which is guaranteed to be in $N_k(x^*)$. Evidently, $x' \in R_k(x^*)$. Now suppose, to get a contradiction, that $x' \in R_{k-1}(x^*)$. But then for each $i \leq n$ we have $y_i^L \leq x'_i \leq y_i^U$. Since we specifically set one of the $x'_i$ to be outside this interval in steps 1-3 of the algorithm, $x' \notin R_{k-1}(x^*)$ as claimed. Therefore $x' \in N_k(x^*)$. The only problem with this method is that the sampling is not uniformly distributed anymore, as there are areas of $N_k(x^*)$ where there is zero probability of sampling $x'$. This, unfortunately, affects the convergence proof of the algorithm, since there are unexplored areas. A simpler, more robust strategy is to define each neighbourhood $N_k(x^*)$ as the hyper-rectangle $R_k(x^*)$. This is wasteful (a point might be sampled in $N_{k-1}(x^*)$, which had already been explored at the previous iteration), but the convergence proof holds. Both approaches have been coded in the solver, and selection occurs by modifying an appropriate parameter.

The other main solver parameters control: the minimum $k$ to start the VNS from, the number of sampling points and local searches started in each neighbourhood, an $\varepsilon$ tolerance to allow moving to a new $x^*$ only when the improvement was sufficiently high, and the maximum CPU time allowed for the search.

### 7.3 Spatial Branch-and-Bound

The overall sBB algorithm was discussed in Section 2.3. Below, we consider some of the key steps of the algorithm in more detail.

#### Bounds tightening

These procedures appear in steps 1 and 2 of the algorithm structure outlined in Section 2.3. They are optional in the sense that the algorithm will, in principle, converge even without them. Depending on how computationally expensive and how effective these procedures are, in some cases convergence might actually be faster if these optional steps are not performed. In the great majority of cases, however, the bounds tightening steps are essential to achieve fast convergence. Two major bounds tightening schemes have been proposed in the literature: optimization-based and feasibility-based.

The optimization-based bounds tightening procedure identifies the smallest range of each variables subject to the convex relaxation of the problem to remain feasible. This ensures that the sBB algorithm will not have to explore hyper-rectangles which do not actually contain any feasible point. Unfortunately, this is a computationally expensive procedure which involves solving at least $2n$ convex NLPs (or LPs if a linear convex relaxation is employed) where $n$ is the number of problem variables. Let $\alpha \leq \bar{g}(x) \leq \beta$ be the set of constraints in the relaxed (convex) problem ($\alpha, \beta$ are the constraint limits). The following procedure will construct sequences $x^{L,k}, x^{U,k}$ of lower and upper variable bounds which converge to new variable bounds that are at least as tight as, and possibly tighter than $x^L, x^U$.

1. Set $x^{L,0} \leftarrow x^L$, $x^{U,0} \leftarrow x^U$, $k \leftarrow 0$.
2. Repeat

$$\begin{aligned}
x_i^{L,k} &\leftarrow \min\{x_i \mid \alpha \le \bar{g}(x) \le \beta \wedge x^{L,k-1} \le x \le x^{U,k-1}\} \quad \forall i \le n;\\
x_i^{U,k} &\leftarrow \max\{x_i \mid \alpha \le \bar{g}(x) \le \beta \wedge x^{L,k-1} \le x \le x^{U,k-1}\} \quad \forall i \le n;\\
k &\leftarrow k+1.
\end{aligned}$$

until $x^{L,k} = x^{L,k-1}$ and $x^{U,k} = x^{U,k-1}$.

Because of the associated cost, this type of tightening is normally performed only once, at the first step of the algorithm.

Feasibility-based bounds tightening is computationally cheaper than the one described above, and as such it can be applied at each and every region considered by the algorithm. Variable bounds are tightened by using the problem constraints to calculate extremal values attainable by the variables. This is done by isolating a variable on the left hand side of a constraint and evaluating the right hand side extremal values by means of interval arithmetic.

Feasibility-based bounds tightening is trivially easy for the case of linear constraints. Given linear constraints in the form $l \le Ax \le u$ where $A = (a_{ij})$, it can be shown that, for all $1 \le j \le n$:

$$\begin{aligned}
x_j \in &\left[\max\left(x_j^L, \min_i\left(\frac{1}{a_{ij}}\left(l_i - \sum_{k\ne j}\max(a_{ik}x_k^L, a_{ik}x_k^U)\right)\right)\right)\right., \\
&\left.\min\left(x_j^U, \max_i\left(\frac{1}{a_{ij}}\left(u_i - \sum_{k\ne j}\min(a_{ik}x_k^L, a_{ik}x_k^U)\right)\right)\right)\right] \text{ if } a_{ij} > 0 \\
x_j \in &\left[\max\left(x_j^L, \min_i\left(\frac{1}{a_{ij}}\left(l_i - \sum_{k\ne j}\min(a_{ik}x_k^L, a_{ik}x_k^U)\right)\right)\right)\right., \\
&\left.\min\left(x_j^U, \max_i\left(\frac{1}{a_{ij}}\left(u_i - \sum_{k\ne j}\max(a_{ik}x_k^L, a_{ik}x_k^U)\right)\right)\right)\right] \text{ if } a_{ij} < 0.
\end{aligned}$$

As pointed out in [71] p.202, feasibility-based bounds tightening can also be carried out for certain types of nonlinear constraints.

### Choice of region

The region selection at step 2 follows the simple policy of choosing the region in the list with the lowest lower objective function bound as the one which is most promising for further consideration (recall that the lower bound $l$ calculated in each region is associated to the subregions after branching — see step 7 of the sBB algorithm).

### Local solution of the original problem

The most computationally expensive step in the sBB algorithm is typically the call to the local NLP solver to find the upper bound to the objective function value relative to the current region. The two methods described below should at least halve the number of upper bounding problems that are solved during the sBB algorithm. Note that a distinction is made between the variables that are present in the original NLP ("original variables") and those added by the standardization procedure ("added variables" — see Section 2.3).

1. *Branching on added variables.* Suppose that in the sBB algorithm an added variable $w$ is chosen as the branch variable. The current region is then partitioned into two sub-regions along the $w$ axis, the convex relaxations are modified to take the new variable ranges into account, and lower bounds are found for each sub-region. The upper bounds, however, are found by solving the original problem which is not dependent on the added variables. Thus the same exact original problem is solved at least three times in the course of the algorithm (i.e. once for the original region and once for each of its two sub-regions). The obvious solution is for the algorithm to record the objective function upper bounds in each region. Whenever the branch variable is an added variable, avoid solving the original (upper bounding) problem and use the stored values instead.
2. *Branching on original variables.* Even when the branching occurs on an original problem variable, there are some considerations that help avoid solving local optimization problems unnecessarily. Suppose that the original variable $x$ is selected for branching in a certain region. Then its range $[x^L, x^U]$ is partitioned into $[x^L, x']$ and $[x', x^U]$. If the solution of the upper bounding problem in $[x^L, x^U]$ is $x^*$, and $x^* \in [x^L, x']$, then it is unnecessary to solve the upper bounding problem again in the sub-region $[x^L, x']$ as an upper bound is already available at $x^*$. Of course, the upper bounding problem still needs to be solved for the other subregion $[x', x^U]$ (see Fig. 9).

### Branching

There are many branching strategies [18] available for use in spatial Branch-and-Bound algorithms. Generally, branching involves two steps, namely determining the point (i.e. set of variable values) on which to branch, and finding the variable whose domain is to be sub-divided by the branching operation. Here, we use the solution $\tilde{x}$ of the upper bounding problem (step 4) as the branching point, if such a solution is found; otherwise the solution of the lower bounding problem $\bar{x}$ (step 3) is used. We then use the standard form to identify the nonlinear term with the largest error with respect to its convex relaxation. By definition of the standard form (see Section 2.3), this is equivalent to evaluating the defining constraints at $\bar{x}$ and choosing the one

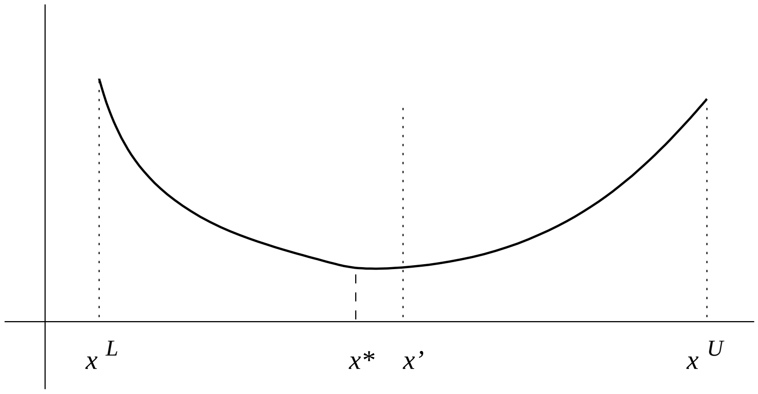


**Fig. 9.** If the locally optimal solution in $[x^L, x^U]$ has already been determined to be at $x^*$, solving in $[x^L, x']$ is unnecessary.

giving rise to the largest error in absolute value. In case the chosen defining constraint represents a unary operator, the only variable operand is chosen as the branch variable; if it represents a binary operator, the branch variable is chosen as the one whose value at the branching point is nearest to the midpoint of its range (see [71], p. 205-207).

### Generation of convex relaxation

As has been explained in Section 2.3, the automatic generation of the convex relaxation entails putting the problem in standard form and then replacing each nonlinear defining constraint with a linear convexification thereof.

The problem in standard form consists of:

- the objective function, consisting of one linearizing variable only (a defining constraint for this equation also exist in the problem, obviously);
- the linear constraints, represented by a matrix;
- the nonlinear constraints, represented by triplets of variable indices and an operator label;
- the constraint bounds;
- the variable ranges;
- the variable values.

Since the nonlinear constraints are isolated in the form $x_i = x_j \otimes x_k$ where $\otimes$ is an operator, they can be efficiently represented by the triplet $(i, j, k)$ and an operator label, $\Lambda$, which indicates what operator acts on $x_j, x_k$ to produce $x_i$. If $\Lambda$ is a unary operator, the index $k$ is set to a dummy value representing "not a variable". If one of the operands is a constant, the respective variable index is set to "not a variable" and the constant value should be stored in a special purpose data field. If both operands are constants, the triplet can be evaluated and discarded. The result of the evaluation replaces all instances of the variable with index $i$; this implies the elimination of the problem variable with index $i$.

The procedure that transforms a problem in standard form is based on the following steps:

1. copy original variable values and bounds, original constraint bounds and linear constraint coefficients from the original problem to initialize the respective standard form data structures;
2. add a constraint $x_i = objectivefunction$ to the problem (separating linear and nonlinear parts) and set $i$ as the variable index representing the objective function;
3. cycle over the original nonlinear parts of the constraints:
   a) recursively split the nonlinear part of the current constraint into triplets;
   b) check if the current triplet already exists. If not, store it, otherwise use the existing triplet and discard it;
4. store all linear triplets (i.e. all triplets where the operand is linear) as linear constraints into the linear constraint matrix and then discard them;
5. eliminate the one-variable constraints (i.e. constraints where the algebraic expression only consists of a one-variable term) and use the bounds to update the respective variable range;
6. eliminate constraints of the form $0 \leq x_i - x_j \leq 0$ and substitute the variable $x_j$ by the variable $x_i$ throughout.

Note that the variable elimination schemes are applied to added variables only. The original variables remain unchanged. This is an important issue as it makes it easy to map original problem variables to relaxed problem variables throughout the sBB execution.

Note also that when linear triplets are stored in the linear constraint matrix, particular care should be taken that the minimum amount of new linear constraints is added to the problem. This is best explained with an example. Let $w_1 = w_2 + w_3$, $w_4 = -w_5$ and $w_2 = w_6 - w_7$; the first triplet is reformulated as the linear constraint $0 \leq w_1 - w_2 - w_3 \leq 0$; the second triplet is reformulated as a new linear constraint $0 \leq w_4 + w_5 \leq 0$ because it is independent of the first one. But the third triplet should *not* be reformulated as a new constraint because $w_2$ already appears in the first constraint. Instead, $w_2$ in the first constraint is replaced by $w_6 - w_7$ and the variable $w_2$ (if it does not appear anywhere else in the problem) is eliminated.

The convexified problem can be described by the same data structure used to represent the original problem. In *oo*$\mathcal{OPS}$, because the implementation uses linear relaxations only, the structure can be simplified by removing all references to nonlinear expressions; also note that since the convexified problem is actually of a different type to that of the original problem, it is stored in a separate object instance. The convexification algorithm is as follows:

1. copy variable bounds, constraint bounds and linear constraint matrix from the problem in standard form;
2. cycle on the standard form triplets:
   a) analyse the current triplet and add its convex envelope constraints to the convexified problem.

Although the above process explained in this section involves quite a lot of data copying between data structures, it is not in fact data replication as the procedures operating on the structures may need to change some of the copied values. The purpose of having different sets of copied values is to allow for full code re-entrancy.

### Region storage scheme

In abstract terms, a region in the sBB algorithm is a hypercube in Euclidean space; thus, it is characterized by a list of $n$ variable ranges. However, this characterization means that, to store $n$ variable ranges explicitly one must allocate and manage memory of size $2n$. This means that to create a new region, we have to repeatedly copy $2n$ memory units from the old region to the new one. Because the partitioning always acts on just one branch variable, all of the other variable ranges would be copied unchanged. Furthermore, because the partitioning always produces two subregions, the waste would be doubled.

In view of the above, we only store the new range for each child subregion together with a pointer to the parent region in order to retrieve the other ranges. This gives rise to a tree of regions where each node contains:

- a pointer to the parent region;
- the branch variable when the region was created;
- the branch variable range of this region;
- the branch point of the parent variable range (one of the endpoints of the branch variable range, it indicates whether this is the “upper” region or the “lower” region);
- the objective function lower and upper bounds;
- a flag that signals whether an upper bound is already available for the region prior to calculation (see Section 7.3).

Starting from any particular region, we can derive the complete set of variable bounds for it by ascending the tree via its parent. In order also to allow traversing the tree in a downwards direction (see below), we add another piece of information:

- pointers to the children subregions.

Also note that some of the regions in the list may be discarded from further consideration at some point in the algorithm. However, we cannot just delete the discarded regions from the tree because they hold information about the variable ranges, so we need a discarding scheme which involves no actual deletion. This is very easy to accomplish with a boolean flag that indicates whether a region is active or inactive (discarded):

- a flag that indicates whether the region is active or not.

#### Control flow in the sBB solver

This section refers to the control flow of the sBB solver in $oo\mathcal{OPS}$ (also see Sections 3.4, 3.4). At the outset, we assume that the user code has created an `opssolvermanager` for the sBB solver and an `opssystem` binding the sBB solver and the problem. The list below starts after the `Solve()` method has been called.

1. Creation of the `opssolvermanager` for the local solver that will be used to solve the upper bounding problem.
2. Creation of an `opssystem` using the `opssolvermanager` just created and the original problem.
3. Creation of the `convexifiermanager` acting on the original problem.
4. Generation of the convex (linear) relaxation. This is held in a modified `ops` class object which only includes data structures for storing linear objective function and coefficients.
5. Creation of the `opssolvermanager` for the local solver used to solve the lower bounding problem.
6. Creation of the `opssystem` using the `opssolvermanager` just created and the convex relaxation.
7. Iterative process:
   a) Follow the sBB algorithm repeatedly calling the `opssystem` acting on upper and lower bounding problems.
   b) On changing the variable ranges update the lower bounding problem on-the-fly (via the `UpdateConvexVarBounds()` method in the `convex-ifiermanager`).
8. Deallocation of objects created by the global solver code.

## 8 Conclusion

In this paper we discussed various aspects of writing general-purpose global optimization software. We first performed a literature review of existing global optimization algorithms and existing global optimization software packages targeted at solving problems in general form (1). The most important issue is that of a sound software architecture and design, which makes it possible to implement very high-level algorithms (including those that call whole sub-algorithms as black box procedures, and those based on symbolic manipulation of mathematical expressions) fairly easily. We suggested a possible standard file format for describing MINLP problems in flat form. Various symbolic computation algorithms have been discussed. We then discussed local solvers generally, and performed an in-depth analysis of the three global solvers implemented within *ooOPS*.

## Acknowledgments

I would like to express the deepest thanks to Dr. Maria Elena Bruni (author of Chapter 4) for valuable suggestions, and to Dr. Sonia Cafieri for detailed information about the SIF format file. Prof. M. Drazić (co-author of Chapter 6) helped with some parts of the implementation of the VNS solver within *ooOPS*.

## References

1. C.S. Adjiman, I.P. Androulakis, and C.A. Floudas. Global optimization of minlp problems in process synthesis and design. *Computers & Chemical Engineering*, 21:S445–S450, 1997.
2. C. S. Adjiman, I. P. Androulakis, and C. A. Floudas. A global optimization method, $\alpha$bb, for general twice-differentiable constrained nlps: Ii. implementation and computational results. *Computers & Chemical Engineering*, 22(9):1159–1179, 1998.
3. C.S. Adjiman, I.P. Androulakis, C.D. Maranas, and C.A. Floudas. A global optimization method, $\alpha$bb, for process design. *Computers & Chemical Engineering*, 20:S419–S424, 1996.
4. P.L. Toint A.R. Conn, N.I.M. Gould. The sif reference report. `http://www.numerical.rl.ac.uk/lancelot/sif/`.
5. C.S. Adjiman, S. Dallwig, C.A. Floudas, and A. Neumaier. A global optimization method, $\alpha$bb, for general twice-differentiable constrained nlps: I. theoretical advances. *Computers & Chemical Engineering*, 22(9):1137–1158, 1998.
6. C.S. Adjiman. *Global Optimization Techniques for Process Systems Engineering.* PhD thesis, Princeton University, June 1998.
7. C.S. Adjiman and C.A. Floudas. Rigorous convex underestimators for general twice-differentiable problems. *Journal of Global Optimization*, 9(1):23–40, July 1996.
8. F.A. Al-Khayyal and J.E. Falk. Jointly constrained biconvex programming. *Mathematics of Operations Research*, 8(2):273–286, 1983.
9. I. P. Androulakis, C. D. Maranas, and C. A. Floudas. alpha bb: A global optimization method for general constrained nonconvex problems. *Journal of Global Optimization*, 7(4):337–363, December 1995.
10. M. Berkelaar. *LP_SOLVE: Linear Programming Code.* `http://www.cs.sunysb.edu/ algorith/implement/lpsolve/implement.shtml`, 2004.
11. A. Brook, D. Kendrick, and A. Meeraus. Gams, a user's guide. *ACM SIGNUM Newsletter*, 23(3-4):10–11, 1988.

12. J. Brimberg and N. Mladenović. A variable neighbourhood algorithm for solving the continuous location-allocation problem. *Studies in Location Analysis*, 10:1–12, 1996.
13. I.D.L. Bogle and C.C. Pantelides. Sparse nonlinear systems in chemical process simulation. In A.J. Osiadacz, editor, *Simulation and Optimization of Large Systems*, Oxford, 1988. Clarendon Press.
14. J.S. Cohen. *Computer Algebra and Symbolic Computation: Mathematical Methods*. AK Peters, Natick, Massachussetts, 2000.
15. J.S. Cohen. *Computer Algebra and Symbolic Computation: Elementary Algorithms*. AK Peters, Natick, Massachussetts, 2002.
16. Process Systems Enterprise. *gPROMS v2.2 Introductory User Guide*. Process Systems Enterprise, Ltd., London, UK, 2003.
17. T.G.W. Epperly and E.N. Pistikopoulos. A reduced space branch and bound algorithm for global optimization. *Journal of Global Optimization*, 11:287:311, 1997.
18. T.G.W. Epperly. *Global Optimization of Nonconvex Nonlinear Programs using Parallel Branch and Bound*. PhD thesis, University of Winsconsin – Madison, 1995.
19. R. Fourer and D. Gay. *The AMPL Book*.
20. C.A. Floudas. *Deterministic Global Optimization*. Kluwer, Dordrecht, 2000.
21. R. Fourer. Personal communication. 2004.
22. J.E. Falk and R.M. Soland. An algorithm for separable nonconvex programming problems. *Management Science*, 15:550–569, 1969.
23. S. Galli. Parsing ampl internal format for linear and non-linear expressions, 2004. Didactical project, DEI, Politecnico di Milano, Italy.
24. P.E. Gill. *User's Guide for SNOPT 5.3*. Systems Optimization Laboratory, Department of EESOR, Stanford University, California, February 1999.
25. A. Griewank. On automatic differentiation. In Iri and Tanabe [31], pages 83–108.
26. K. Hägglöf, P.O. Lindberg, and L. Svensson. Computing global minima to polynomial optimization problems using gröbner bases. *Journal of Global Optimization*, 7(2):115:125, 1995.
27. P. Hansen and N. Mladenović. Variable neighbourhood search: Principles and applications. *European Journal of Operations Research*, 130:449–467, 2001.
28. P. Hansen and N. Mladenović. Variable neighbourhood search. In P. Pardalos and M. Resende, editors, *Handbook of Applied Optimization*, Oxford, 2002. Oxford University Press.
29. P. Hansen and N. Mladenović. Variable neighbourhood search. In F.W. Glover and G.A. Kochenberger, editors, *Handbook of Metaheuristics*, Dordrecht, 2003. Kluwer.
30. ILOG. *ILOG CPLEX 8.0 User's Manual*. ILOG S.A., Gentilly, France, 2002.
31. M. Iri and K. Tanabe, editors. *Mathematical Programming: Recent Developments and Applications*. Kluwer, Dordrecht, 1989.
32. P. Kesavan and P.I. Barton. Generalized branch-and-cut framework for mixed-integer nonlinear optimization problems. *Computers & Chemical Engineering*, 24:1361–1366, 2000.
33. R. B. Kearfott. *GlobSol User Guide*. `http://interval.louisiana.edu/GLOBSOL/what_is.html`, 1999.
34. D.E. Knuth. *The Art of Computer Programming, Part II: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, 1981.

35. B.R. Keeping and C.C. Pantelides. Novel methods for the efficient evaluation of stored mathematical expressions on scalar and vector computers. *AIChE Annual Meeting*, Paper #204b, nov 1997.
36. B. Keeping, C.C. Pantelides, James Barber, and Panagiotis Tsiakis. Mixed integer linear programming interface specification draft. *Global Cape-Open Deliverable WP2.3-03*, October 2000.
37. S. Kucherenko and Yu. Sytsko. Application of deterministic low-discrepancy sequences to nonlinear global optimization problems. *Computational Optimization and Applications*, (to appear) 2004.
38. V. Kovačević-Vujčić, M. Čangalović, M. Ašić, L. Ivanović, and M. Dražić. Tabu search methodology in global optimization. *Computers and Mathematics with Applications*, 37:125–133, 1999.
39. L. Liberti. *Reformulation and Convex Relaxation Techniques for Global Optimization*. PhD thesis, Imperial College London, UK, March 2004.
40. L. Liberti. Performance comparison of function evaluation methods. *Progress in Computer Science Research*, (to appear) 2004.
41. L. Liberti and S. Kucherenko. Comparison of deterministic and stochastic approaches to global optimization. *DEI, Politecnico di Milano, Technical report n. 2004.25*, July 2004.
42. R. Levine, T. Mason, and D. Brown. *Lex and Yacc*. O'Reilly, Cambridge, second edition, 1995.
43. L. Liberti, N. Maculan, and S. Kucherenko. The kissing number problem: a new result from global optimization. In L. Liberti and F. Maffioli, editors, *CTW04 Workshop on Graphs and Combinatorial Optimization, Menaggio, Italy, June 2004*, volume 17, Amsterdam, 2004. Electronic Notes in Discrete Mathematics, Elsevier.
44. M. Locatelli. Simulated annealing algorithms for global optimization. In Pardalos and Romeijn [58], pages 179–229.
45. L. Liberti and C.C. Pantelides. Convex envelopes of monomials of odd degree. *Journal of Global Optimization*, 25:157–168, 2003.
46. M. Locatelli and F. Schoen. Simple linkage: Analysis of a threshold-accepting global optimization method. *Journal of Global Optimization*, 9:95–111, 1996.
47. M. Locatelli and F. Schoen. Random linkage: a family of acceptance/rejection algorithms for global optimization. *Mathematical Programming*, 85(2):379–396, 1999.
48. L. Liberti, P. Tsiakis, B. Keeping, and C.C. Pantelides. *ooOPS*. Centre for Process Systems Engineering, Chemical Engineering Department, Imperial College, London, UK, 1.24 edition, jan 2001.
49. A. Makhorin. *GNU Linear Programming Kit*. Free Software Foundation, `http://www.gnu.org/software/glpk/`, 2003.
50. G.P. McCormick. Computability of global solutions to factorable nonconvex programs: Part i — convex underestimating problems. *Mathematical Programming*, 10:146–175, 1976.
51. M. Mathur, S.B. Karale, S. Priye, V.K. Jayaraman, and B.D. Kulkarni. Ant colony approach to continuous function optimization. *Industrial and Engineering Chemistry Research*, 39:3814–3822, 2000.
52. N. Mladenović, J. Petrović, V. Kovačević-Vujčić, and M. Čangalović. Solving a spread-spectrum radar polyphase code design problem by tabu search and variable neighbourhood search. *European Journal of Operations Research*, 151:389–399, 2003.

53. Numerical Algorithms Group. *NAG Fortran Library Manual Mark 11.* 1984.
54. C.C. Pantelides. *Symbolic and Numerical Techniques for the Solution of Large Systems of Nonlinear Algebraic Equations.* PhD thesis, Imperial College of Science, Technology and Medicine, University of London, May 1988.
55. Janós Pintér. *LGO: a Model Development System for Continuous Global Optimization. User's Guide.* Pintér Consulting Services, Halifax, NS, Canada, 1999.
56. C.C. Pantelides, L. Liberti, P. Tsiakis, and T. Crombie. Minlp interface specification. *CAPE-OPEN Update*, 2:10–13, March 2002.
57. C.C. Pantelides, L. Liberti, P. Tsiakis, and T. Crombie. Mixed integer linear/nonlinear programming interface specification. *Global Cape-Open Deliverable WP2.3-04*, February 2002.
58. P.M. Pardalos and H.E. Romeijn, editors. *Handbook of Global Optimization*, volume 2. Kluwer, Dordrecht, 2002.
59. W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C, Second Edition.* Cambridge University Press, Cambridge, 1992, reprinted 1997.
60. A.H.G. Rinnooy-Kan and G.T. Timmer. Stochastic global optimization methods; part i: Clustering methods. *Mathematical Programming*, 39:27–56, 1987.
61. A.H.G. Rinnooy-Kan and G.T. Timmer. Stochastic global optimization methods; part ii: Multilevel methods. *Mathematical Programming*, 39:57–78, 1987.
62. H.S. Ryoo and N.V. Sahinidis. Global optimization of nonconvex nlps and minlps with applications in process design. *Computers & Chemical Engineering*, 19(5):551–566, 1995.
63. H. S. Ryoo and N. V. Sahinidis. A branch-and-reduce approach to global optimization. *Journal of Global Optimization*, 8(2):107–138, March 1996.
64. H. Ryoo and N. Sahinidis. Global optimization of multiplicative programs. *Journal of Global Optimization*, 26(4):387–418, 2003.
65. P. RoyChowdury, Y.P. Singh, and R.A. Chansarkar. Hybridization of gradient descent algorithms with dynamic tunneling methods for global optimization. *IEEE Transactions on Systems, Man and Cybernetics—Part A: Systems and Humans*, 30(3):384–390, 2000.
66. N.V. Sahinidis. Baron: Branch and reduce optimization navigator, user's manual, version 4.0. *http://archimedes.scs.uiuc.edu/baron/manuse.pdf*, 1999.
67. F. Schoen. Random and quasi-random linkage methods in global optimization. *Journal of Global Optimization*, 13:445–454, 1998.
68. F. Schoen. Global optimization methods for high-dimensional problems. *European Journal of Operations Research*, 119:345–352, 1999.
69. F. Schoen. Two-phase methods for global optimization. In Pardalos and Romeijn [58], pages 151–177.
70. H. Schichl. *The Coconut API: Reference Manual.* Dept. of Maths, Universität Wien, October 2003.
71. E.M.B. Smith. *On the Optimal Design of Continuous Processes.* PhD thesis, Imperial College of Science, Technology and Medicine, University of London, October 1996.
72. J.E. Smith. Genetic algorithms. In Pardalos and Romeijn [58], pages 275–362.
73. E.M.B. Smith and C.C. Pantelides. Global optimisation of nonconvex minlps. *Computers & Chemical Engineering*, 21:S791–S796, 1997.
74. R. Storn and K. Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11:341–359, 1997.

75. E.M.B. Smith and C.C. Pantelides. A symbolic reformulation/spatial branch-and-bound algorithm for the global optimisation of nonconvex minlps. *Computers & Chemical Engineering*, 23:457–478, 1999.
76. T.K. Shi, W.H. Steeb, and Y. Hardy. *An Introduction to Computer Algebra Using Object-Oriented Programming.* Springer-Verlag, Berlin, second edition, 2000.
77. G. Schrimpf, J. Schneider, H. Stamm-Wilbrandt, and G. Dueck. Record breaking optimization results using the ruin and recreate principle. *Journal of Computational Physics*, 159:139–171, 2000.
78. B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, third edition, 1999.
79. Lindo Systems. *Lindo API: User's Manual.* Lindo Systems, Inc., Chicago, 2004.
80. M. Tawarmalani and N.V. Sahinidis. Semidefinite relaxations of fractional programming via novel techniques for constructing convex envelopes of nonlinear functions. *Journal of Global Optimization*, 20(2):137–158, 2001.
81. M. Tawarmalani and N. Sahinidis. Convex extensions and envelopes of semi-continuous functions. *Mathematical Programming*, 93(2):247–263, 2002.
82. M. Tawarmalani and N.V. Sahinidis. Exact algorithms for global optimization of mixed-integer nonlinear programs. In Pardalos and Romeijn [58], pages 1–63.
83. M. Tawarmalani and N.V. Sahinidis. Global optimization of mixed integer non-linear programs: A theoretical and computational study. *Mathematical Programming*, 99:563–591, 2004.
84. H. Tuy. *Convex Analysis and Global Optimization.* Kluwer, Dodrecht, 1998.
85. B.W. Wah and T. Wang. Efficient and adaptive lagrange-multiplier methods for nonlinear continuous global optimization. *Journal of Global Optimization*, 14:1:25, 1999.
86. J. M. Zamora and I. E. Grossmann. A branch and contract algorithm for problems with concave univariate, bilinear and linear fractional terms. *Journal of Global Optimization*, 14:217:249, 1999.