Reformulation and Convex Relaxation Techniques for Global Optimization

LEO SERGIO LIBERTI

15th March 2004

A thesis submitted for the degree of Doctor of Philosophy of the University of London
and for the Diploma of Imperial College

Department of Chemical Engineering and Chemical Technology

Imperial College London

South Kensington Campus
London SW7 2AZ

Abstract

Many engineering optimization problems can be formulated as nonconvex nonlinear programming problems (NLPs) involving a nonlinear objective function subject to nonlinear constraints. Such problems may exhibit more than one locally optimal point. However, one is often solely or primarily interested in determining the globally optimal point. This thesis is concerned with techniques for establishing such global optima using spatial Branch-and-Bound (sBB) algorithms.

A key issue in optimization is that of mathematical formulation, as there may be several different ways in which the same engineering problem can be expressed mathematically. This is particularly important in the case of global optimization as the solution of different mathematically equivalent formulations may pose very different computational requirements. Based on the concept of reduction constraints, the thesis presents a set of graph-theoretical algorithms which automatically reformulate large sparse nonconvex NLPs involving linear constraints and bilinear terms. It is shown that the resulting exact reformulations involve fewer bilinear terms and have tighter convex relaxations than the original NLPs. Numerical results illustrating the beneficial effects of applying such automatic reformulations to the well-known pooling and blending problem are presented.

All sBB algorithms rely on the construction of a convex relaxation of the original NLP problem. Relatively tight convex relaxations are known for many categories of algebraic expressions. One notable exception is that of monomials of odd degree, i.e. expressions of the form x^{2n+1} where $n \geq 1$, when the range of the variable x includes zero. These occur often (e.g. as cubic or quintic expressions) in practical applications. The thesis presents a novel method for constructing the convex envelope of such monomials, as well as a tight linear relaxation of this envelope.

Finally, the thesis discusses some of the software engineering issues involved in the design and implementation of codes for sBB, especially in view of the large amounts of both symbolic and numerical information required by these codes. A prototype object-oriented software library, $oo\mathcal{OPS}$, is described.

Acknowledgments

After nearly four years of research, it is hard to think whom *not* to thank in this Ph.D. thesis. Besides, I feel I owe everyone who has been around me during my stay in London a gesture of thanks. However, this would result into a short biography of my last four years rather than a Ph.D. acknowledgment.

I naturally want to thank my supervisor Costas Pantelides for his support, both technical and financial. I would also like to thank: Panagiotis Tsiakis and Benjamin Keeping for producing the $oo\mathcal{MILP}$ software, on which my $oo\mathcal{OPS}$ software is heavily based; Fabrizio Bezzo for listening to many of my woes and lamentations about my latest theoretical ideas leading to nothing, and helping whenever he could; Gerard Gorman for our lengthy talks about C++, coding and debugging techniques; and Panagiotis Karamertzanis for crucial help with the bureaucratic pitfalls of thesis submission. A lot of other people, both in the CPSE and elsewhere, have helped me in my research, but the list is just too long to include it here. As too long would be the list of all people who have extended their moral support and friendship to me during these years: so please excuse me if you are not on this page. You are in my memory nonetheless.

Finally, I would like to thank my family and three dearest friends, Jacopo, Giovanna and Cecilia, for their love and support in times of need.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Optimization is a branch of mathematics that studies the problem of finding the best choice among a set of entities satisfying some imposed requirements.

In general, the problem is formulated in terms of finding the point x in a space set Ω (called the *feasible region*) where a certain function $f : \Omega \to T$ (called the *objective function*), attains a minimum or a maximum. T is any ordered set.

1.1 Basic definitions

In this section we shall introduce some basic definitions which will be used throughout this thesis.

The set $\Omega \subseteq \mathbb{R}^n$ is *convex* if, for all $x, y \in \Omega$ and for all $\lambda \in [0, 1]$, the vector $\lambda x + (1 - \lambda)y$ is also in Ω. The intersection of an arbitrary collection of convex sets is convex. Let $S \subseteq \mathbb{R}^n$. The intersection of all the convex subsets of $\mathbb{R}^n$ containing S is called the *convex hull* of S. The convex hull of a finite subset $\{v_1, \ldots, v_m\}$ of $\mathbb{R}^n$ consists of all the linear combinations $\sum_{i=1}^{m} \lambda_i v_i$ where $\lambda_i \geq 0$ for all $i \leq m$ and $\sum_{i=1}^{m} \lambda_i = 1$. A function $f : \mathbb{R}^n \to \mathbb{R}$ is *convex* if and only if it is defined on a convex set Ω and is such that for all $x, y \in \Omega$ and for all $\lambda \in [0, 1]$ we have

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y).$$

A function f is *concave* if $-f$ is convex.

A function f is *pseudo-convex* if, for all x_1, x_2 such that $f(x_1) < f(x_2)$, we have $\nabla f(x_2)(x_1 - x_2) < 0$.

A function f is *quasi-conovex* if all its sublevel sets $S_\alpha = \{x \mid f(x) \leq \alpha\}$ are convex.

A function $f : \Omega \to \mathbb{R}$ is a *d.c. function* if it is a difference of convex functions, i.e. there exist convex functions $g, h : \Omega \to \mathbb{R}$ such that, for all $x \in \Omega$, we have $f(x) = g(x) - h(x)$. Let C, D be convex sets; then the set $C \backslash D$ is a *d.c. set*[1]. It can be shown that a set M for which there exist convex functions $g, h : \mathbb{R}^n$ such that $M = \{x \mid g(x) \leq 0 \wedge h(x) \geq 0\}$ is a d.c. set. D.c. functions and sets have many interesting properties. See Section 2.1.7 for a more thorough discussion.

Let $f(x)$ be a nonconvex function. A *convex relaxation* of $f(x)$ is a convex function $\underline{f}(x)$ such that, for all x, we have $\underline{f}(x) \leq f(x)$. Likewise, a *concave relaxation* of $f(x)$ is a concave function $\bar{f}(x)$ such that for all x we have $\bar{f}(x) \geq f(x)$. Let $\underline{F}$ be the set of all convex relaxations of f and $\bar{F}$ the set of all concave relaxations. The *convex envelope* of f is $\max\{g(x) \mid g \in \underline{F}\}$; the *concave envelope* of f is $\min\{g(x) \mid g \in \bar{F}\}$.

Given an inequality $g(x) \leq 0$, the set $\{x \mid g(x) \leq 0\}$ is convex if $g(x)$ is a convex function. Notice that, if $g(x)$ is concave, then $g(x) \geq 0$ is a convex inequality.

1.2 Classification of optimization problems

The field of optimization can be subdivided into many categories according to the different possible formulations of the problem.

- Classification with respect to the type of optimization decisions.
  - **Continuous optimization**. The set Ω is a subset of a Euclidean space.
  - **Integer optimization**. Ω is finite or countable.
  - **Mixed integer optimization**. $\Omega = U \times V$ where U is a subset of a Euclidean space and V is finite or countable.
- Classification with respect to the type of objective function and constraints.
  - **Constrained optimization**. Ω is defined by a set of constraints C to be satisfied.

[1] I.e. a difference of convex sets; here $C \backslash D$ is taken to be all the elements in C which are not in D.

- **Unconstrained optimization**. The set of constraints C to be satisfied is empty. However, problems with variable ranges as their only constraints are sometimes also classified as "unconstrained problems".
- **Linear optimization**. The function f and the constraints in C are linear functions of x.
- **Nonlinear optimization**. The function f and the constraints in C may be nonlinear functions of x.
- **Convex optimization**. The function f is convex and feasible region Ω is convex. In particular, a problem with convex constraints belongs to this class.
- **Concave optimization**. The function f is concave (usually, but not always, the constraints in C are convex).
- **D.c. optimization**. The function f is d.c. and the feasible region Ω is a d.c. set.
- **Nonconvex optimization**. The function f and the constraints in C may be general nonconvex functions.

• Classification with respect to the type of solution obtained.

- **Local optimization**. The solution x is such that there is a subset $V \subseteq \Omega$, such that $x \in V$, where $\forall y \in V \ (f(x) \leq f(y))$. The point x is called a *local minimizer*, or a *local optimum* of f with respect to V. It is nearly always the case that V is a neighbourhood of x in the topological sense.
- **Global optimization**. The solution x is such that $\forall y \in \Omega \ (f(x) \leq f(y))$. x is called a *global minimizer*, or a *global optimum* of f.

One of the most interesting and most challenging classes of problems is global nonlinear constrained optimization. Problems belonging to this class are called nonlinear programs (NLPs).

1.3 Algorithms for global optimization of NLPs

Algorithmic methods for globally solving NLPs are usually divided into two main categories: deterministic and nondeterministic (or stochastic). In deterministic optimization, the solution method never employs random choices, and convergence theorems do not use probability arguments. In stochastic optimization[2] algorithms, the solution methods employ random[3] choices. Convergence is proved through arguments based on probability.

1.3.1 Formulation of the NLP

In this thesis, we are interested in constrained NLPs of the following form:

$$\left.\begin{array}{rcl} \min_x & f(x) & \\ \alpha & \leq g(x) \leq & \beta \\ a & \leq x \leq & b \end{array}\right\} \tag{1.1}$$

where $x \in \mathbb{R}^n$ are the (continuous) decision variables, α, β are the lower and upper bounds of the constraints, and a, b are the lower and upper bounds of the variables. The function $f : \mathbb{R}^n \to \mathbb{R}$ is called the *objective function* and $g : \mathbb{R}^n \to \mathbb{R}^m$ are the *constraints* of the problem. We limit the discussion to the minimization of the problem; the maximization is equivalent to solving $\min_x -f(x)$ subject to the same constraints.

Notice that formulation (1.1) is slightly unusual because of the form of the constraints $\alpha \leq g(x) \leq \beta$. The most common constraint formulation in the literature is $h(x) = 0 \wedge g(x) \leq 0$. We employ the former because it makes the software representation of an optimization problem more compact.

1.3.2 A brief history of global optimization

Generic optimization problems have been important throughout history in engineering applications. The first significant work in optimization was carried out by Lagrange in 1797 [66]. Nevertheless, before the introduction and extensive usage of electronic computers, the requirement for global optimization was not even an issue given the tremendous amount of computational effort it requires. Global optimality of solutions was guaranteed only when locally optimizing convex functions, a rather limited class of problems.

The methods that were first used in global optimization were deterministic techniques, mostly based on the divide-and-conquer principle. This was introduced in the late 1950s with the ad-

[2]Albeit an abuse of terminology, the term “nondeterministic optimization” is sometimes also used.

[3]The term “random” is used here in the special sense of pseudo-random.

vent of the first electronic computers into the research environment. In contrast with the computational principles employed before the introduction of computers, whereby people would try to minimize the amount of actual computations to perform with respect to the theoretical framework, the divide-and-conquer principle applies to the intrinsic iterative nature of methods used when working with electronic computers: keep the complexity of the theoretical structures involved to a minimum, and rely on intensive computation to explore the solution space. Thus, instead of trying to locate a minimum by solving a set of equations by symbolic/algebraic methods, one would try to construct a sequence of approximate solutions which would converge to the true solution, by dividing the problem into smaller subproblems. One typical algorithm which embodies the divide-and-conquer principle is the Branch-and-Bound algorithm (BB). Because of the nature of the algorithm, where the subproblems are produced by branching a problem entity (e.g. variable) into its possible instances, the BB algorithm applies very well to cases where problem entities are discrete in nature. Thus, the first applications of BB to global optimization problems were devoted to discrete problems such as the Travelling Salesman Problem (TSP).

The first paper concerning continuous global optimization with a BB (deterministic) technique dates from 1969 [34]. 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 global optimization to very specific cases, or with theoretical results concerning convergence proofs. One notable exception was the work of McCormick [80] who considered symbolic transformations of problems: his methods are such that they can, in theory, be carried out automatically by a computer.

A major reason for the slow pace of progress in continuous global optimization is that it is computationally very expensive, and it was not until the 1990s that computer hardware with the necessary power became available. Toward the end of the 1980s, however, the first elementary textbooks began to appear on the subject. They explained the basics of local optimization (Lagrange multipliers and Karush-Kuhn-Tucker conditions) and some of the early techniques in deterministic [92] and stochastic [131] global optimization. Early topics in deterministic global optimization include convex optimization [94], Branch-and-Bound techniques restricted to particular classes of problems (e.g. concave minimization problems [53]) and some theoretical and complexity-related studies [135, 97]. Stochastic algorithms based on adaptive random search appeared between the 1970s and early 1980s [74, 77]. It is worth noting that the first deterministic global optimization technique that was able to deal with generic nonconvex continuous NLPs was Interval Optimization [48, 49]). Unfortunately, it often has slow convergence due to the fact that Interval Arithmetic generally provides very wide intervals for the objective function value.

Toward the end of the 1980s, many articles on deterministic global optimization started to appear in the literature. They were mostly concerned with iterative methods applied to particular classes of problems [89, 81, 54, 56, 134, 63, 82], but there were also theoretical studies [50, 55] and parallel implementation studies and reports [90, 42, 33]. In the area of stochastic optimization, some of the first algorithms employing "tunnelling" [145] began to appear in the same period.

Since the beginning of the 1990s, the optimization research community has witnessed an explosion of papers, books, algorithms, software packages and resources concerning deterministic and stochastic global optimization. In the early 1990s, most of the articles were still concerned with applications of global optimization or algorithms which perform well on particular classes of problems. It is significant that one of the first and most widely used book in global optimization [59], first published in 1990 and successively re-published in 1993 and 1996, does not even mention general nonconvex NLPs in the generic form (1.1). The same is true even for a "survey book" ([57]) which appeared in 1995.

The first method that was able to deal directly with the generic nonconvex NLPs in the form (1.1) was Ryoo and Sahinidis' Branch-and-Reduce algorithm which appeared in an paper published in May 1995 [99, 100]. Shortly afterward, Floudas' team published their first article on the αBB branch-and-bound method [13] which was then thoroughly explored and analysed in several subsequent papers by Adjiman [7, 6, 3, 4, 5, 8]. The first αBB variant that addressed problems in the form (1.1) appeared in 1997 [3].

One notable limitation of the αBB algorithm is that it relies on the functions being twice differentiable in the continuous variables. Since the inception of the αBB algorithm, a number of Branch-and-Select algorithms geared toward the most generic nonconvex MINLP formulation appeared in the literature, like Smith and Pantelides' symbolic reformulation approach [114, 115, 116], Pistikopoulos' Reduced Space Branch-and-Bound approach [32] (which only applies to continuous NLPs), Grossmann's Branch-and-Contract algorithm [146] (which also only applies to continuous NLPs) and Barton's Branch-and-Cut framework [61]. Within the Branch-and-Select strategy, we can also include modern Interval Analysis based global optimization methods [137, 84, 136].

Branch-and-Select is by no means the only available technique for deterministic global optimization, but it seems to be the method which least relies on the problem having a particular structure: this is an advantage insofar as one needs to implement software which solves problems in the general form (1.1). For example, d.c. optimization and the Extended Cutting Planes (ECP) method [143, 142] are structurally different from Branch-and-Select approaches; however, both make use of a problem which is already in a special form[4]. It must be said, however, that "special forms" are sometimes general enough to accommodate large classes of problems. For instance, the ECP method solves mixed-integer NLPs (MINLPs) in the following form: $\min\{f(z) \mid g(z) \leq 0 \wedge Az \leq a \wedge Bz = b \wedge z \in \mathbb{R}^n \times \mathbb{Z}^m\}$, where f is a pseudo-convex function, g is a set of pseudo-convex functions, A, B are matrices and a, b are vectors. Although not all functions are pseudo-convex, the latter do form quite a large class. It is the pseudo-convexity condition that allows the ECP method to derive a new valid cutting plane at each iteration.

In the stochastic global optimization field, recent advances include Differential Evolution [119], Adaptive Lagrange-Multiplier Methods [140], Simulated and Nested Annealing [96], Ant Colony Simulation [78, 25], Quantum Dynamics in complex biological evolution [52], Quantum Thermal Annealing [68], Ruin and Recreate Principle [105] and Tabu Search [23].

1.3.3 Two-phase global optimization algorithms

Most methods for global optimization are based on two phases [104]: a global search phase and a local search phase. The reason for this is that there are very efficient local optimization methods which can be employed in the local search phase, but these methods are not generally capable of determining a global optimum of a problem in its most generic form (1.1). The local phase is nearly always deterministic in nature whereas the global phase may be either deterministic or stochastic.

In the global phase, the problem is limited to a particular region of space where one can prove (or just hopes) that a local search will find the global optimum. This procedure is iterative in nature so that the search space can be explored exhaustively.

Local optimization of nonconvex problems is an NP-hard problem [93]. Because all iterative methods for global optimization rely on a local search phase, under the most general conditions global optimization of NLPs is an NP-hard problem. The only non-iterative method for global optimization, based on Gröbner bases [47], is also NP-hard [26].

This research focuses on deterministic algorithms for global optimization. In particular, the Branch-and-Select family of algorithms promises a good performance in terms of computational

[4]I.e., objective function and constraints must be pseudo-convex. This form is needed to guarantee global optimality.

time, it can be applied to problems in very general form, and is deterministic in nature.

1.4 The branch-and-select strategy

Branch-and-Select algorithms include well-known techniques such as Branch-and-Cut and Branch-and-Bound, and are among the most effective methods for the deterministic solution of global optimization problems. They started out as divide-and-conquer type algorithms to solve combinatorial optimization problems like the Travelling Salesman Problem [9] but were very soon applied to continuous and mixed-integer nonlinear optimization [34, 59, 92]. In this section, we present a general theory of such algorithms (based on material from [133]), together with the necessary convergence proofs.

Branch-and-Select algorithms can be used to solve the widest possible class of optimization problems (1.1) to global optimality, even when objective function and constraint values are provided by black-box procedures. In fact, they can be designed to rely on no particular feature of the problem structure. However, in their basic form and without any acceleration devices (such as pre-processing steps or improved algorithmic steps), they tend to be very inefficient. Furthermore, their performance may depend very much on the formulation of the problem, meaning that a different algebraic form of the same equations might lead to quite different algorithmic performance.

Let $\Omega \subseteq \mathbb{R}^n$. A finite family of sets $\mathcal{S}$ is a *net* for Ω if it is pairwise disjoint and it covers Ω, that is, $\forall s, t \in \mathcal{S}\ (s \cap t = \emptyset)$ and $\Omega \subseteq \bigcup_{s \in \mathcal{S}} s$. A net $\mathcal{S}'$ is a *refinement* of the net $\mathcal{S}$ if there are finitely many pairwise disjoint $s'_i \in \mathcal{S}'$ such that $s = \bigcup_i s'_i \in \mathcal{S}$ and $s \notin \mathcal{S}$. In other words, if $\mathcal{S}'$ is a refinement of $\mathcal{S}$, it has been obtained from $\mathcal{S}$ by finitely partitioning some set s in $\mathcal{S}$ and then replacing s by its partitions.

Let $\mathcal{S}_n$ be an infinite sequence of nets for Ω such that, for all $i \in \mathbb{N}$, $\mathcal{S}_i$ is a refinement of $\mathcal{S}_{i-1}$, and let M_n be an infinite sequence of subsets of Ω such that $M_i \in \mathcal{S}_i$. M_n is a *filter for $\mathcal{S}_n$* if $\forall i \in \mathbb{N}\ (M_i \subseteq M_{i-1})$. Let $M_\infty = \bigcap_{i \in \mathbb{N}} M_i$ be the *limit* of the filter.

We will now present a general framework for Branch-and-Select algorithms that globally solve the generic problem $\min\{f(x) \mid x \in \Omega\}$. Let $\gamma \in \mathbb{R}$. Given any net $\mathcal{S}$ for $\Omega \cap \{x \mid f(x) < \gamma\}$, we consider a selection rule that determines:

1. a distinguished point $\omega(M)$ for every $M \in \mathcal{S}$,

2. a subfamily $\mathcal{R}$ of qualified members of $\mathcal{S}$,

3. a distinguished member $M^*(\mathcal{S})$ of $\mathcal{R}$,

such that, for all $x \in \bigcup_{s \in \mathcal{S} \setminus \mathcal{R}} s$, we have $f(x) \geq \gamma$. Basically, the selection rule rejects regions which cannot contain the global optimum by choosing the qualified members; for each of these members, it calculates a distinguished point (for example, by applying a local optimization procedure to the specified region) and a distinguished member for further net refinement. The prototype algorithm below also relies on a selection rule used to partition a region.

1. (Initialization) Start with a net $\mathcal{S}_1$ for Ω, set $x_0 = \emptyset$ and let γ_0 be any upper bound for $f(\Omega)$. Set $\mathcal{P}_1 = \mathcal{S}_1$, $k = 1$, $\sigma_0 = \emptyset$.

2. (Evaluation) Let $\sigma_k = \{\omega(M) \mid M \in \mathcal{P}_k\}$ be the set of all distinguished points.

3. (Incumbent) Let x_k be the point in $\{x_{k-1}\} \cup \sigma_k$ such that $\gamma_k = f(x_k)$ is lowest.

4. (Screening) Determine the family $\mathcal{R}_k$ of qualified members of $\mathcal{S}_k$ (in other words, reject the unqualified members, i.e. those that can be shown not to contain a solution better than γ_k).

5. (Termination) If $\mathcal{R}_k = \emptyset$, terminate. The problem is infeasible if $\gamma_k \geq \gamma_0$; otherwise x_k is the global optimum.

6. (Selection) Select the distinguished member $M_k = M^*(\mathcal{S}_k) \in \mathcal{R}_k$ and partition M_k according to a pre-specified branching rule. Let $\mathcal{P}_{k+1}$ be the partition of M_k. In $\mathcal{R}_k$, replace M_k by $\mathcal{P}_{k+1}$, thus obtaining a new refinement net $\mathcal{S}_{k+1}$. Set $k \leftarrow k+1$ and go back to Step 2.

A Branch-and-Select algorithm is *convergent* if $\gamma^* = \inf f(\Omega) = \lim_{k\to\infty} \gamma_k$. A selection rule is *exact* if:

1. the infimum objective function value of any region that remains qualified during the whole solution process is greater than or equal to the globally optimal objective function value, i.e.
$$\forall M \in \bigcap_{k=1}^{\infty} \mathcal{R}_k \; (\inf f(\Omega \cap M) \geq \gamma^*)$$

2. the limit M_∞ of any filter M_k is such that $\inf f(\Omega \cap M_\infty) \geq \gamma^*$.

In this theoretical set-up, it is easy to show that a Branch-and-Select algorithm using an exact selection rule converges.

1.4.1 Theorem

A Branch-and-Select algorithm using an exact selection rule converges.

Proof. Suppose, to get a contradiction, that there is $x \in \Omega$ with $f(x) < \gamma^*$. Let $x \in M$ with $M \in \mathcal{R}_n$ for some $n \in \mathbb{N}$. Because of condition (1) above, M cannot remain qualified forever; furthermore, unqualified regions may not, by hypothesis, include points with better objective function values than the current incumbent γ_k. Hence M must necessarily be split at some iteration $n' > n$. So x belongs to every M_n in some filter $\{M_n\}$, thus $x \in \Omega \cap M_\infty$. By condition (2) above, $f(x) \geq \inf f(\Omega \cap M_\infty) \geq \gamma^*$. The result follows. $\square$

It is worth noting that in this algorithmic framework does not provide a guarantee of a finite convergence. Consequently, most Branch-and-Select implementations make use of the concept of ε-optimality, rather than the usual definition of optimality. Recall that $x^* \in \Omega$ is a global optimum if, for all $x \in \Omega$, we have $f(x^*) \leq f(x)$. Given $\varepsilon > 0$, $\bar{x} \in \Omega$ is ε-globally optimal if there exist bounds $m \leq f(x^*) \leq M$ such that $f(\bar{x}) \in [m, M]$ and $M - m < \varepsilon$. By employing this notion and finding converging lower and upper bounds sequences to the incumbent at each step of the algorithm, it is easier to ensure a finite termination with an ε-global optimum. For a theoretically proven finite termination, we would need some additional regularity assumptions (see for example [106, 11]); ε-optimality, however, is sufficient for most practical purposes.

1.4.1 Fathoming

Let $\mathcal{S}_k$ be the net at iteration k. For each region $M \in \mathcal{S}_k$, find the lower bound $l(M)$ of $f(M \cap \Omega)$ and define $\omega(M) = l(M)$. A set $M \in \mathcal{S}_k$ is qualified if $l(M) \leq \gamma_k$. The distinguished region is usually selected as the one with the lowest $l(M)$ (this may vary in some implementations).

The algorithm is accelerated if one also computes an upper bound $u(M)$ of $f(M \cap \Omega)$ and uses it to bound the problem from above by rejecting any M for which $l(M)$ exceeds the *best* upper bound that has been encountered so far. In this case, we say that the rejected regions have been *fathomed* (see Fig. 1.1). This acceleration device has become part of most Branch-and-Bound algorithms in the literature. Usually, the upper bound $u(M)$ is computed by solving the problem to local optimality in the current region: this also represents a practical alternative to the implementation of the evaluation step (step (2) in the algorithm of Section 1.4), since we can take the distinguished points $\omega(M)$ to be the local solutions $u(M)$ of the problem in the current regions. With a good numerical local solver, the accuracy of ε-global optimum is likely to be better than if we simply use $l(M)$ to define the distinguished points.

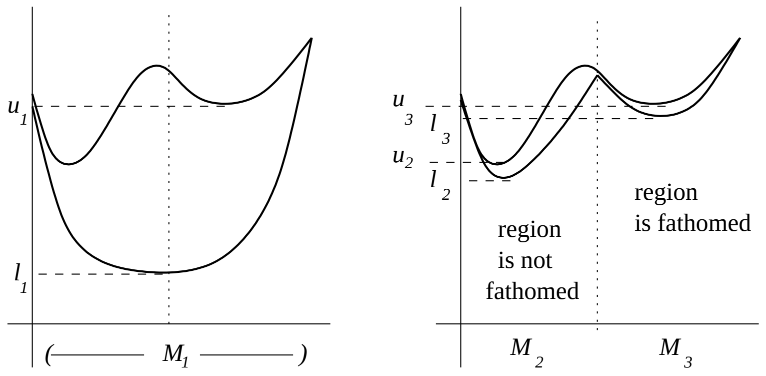


Figure 1.1: Fathoming via upper bound computation.

The convergence of the Branch-and-Bound algorithm is ensured if every filter $\{M_k | k \in K\}$ contains an infinite nested sequence $\{M_k | k \in K_1\}$ such that:

$$\lim_{\substack{k\to\infty \\ k\in K_1}} l(M_k) = \gamma_k. \tag{1.2}$$

To establish this, we will show that under such conditions the selection procedure is exact. Let $\{M_k | k \in K\}$ be any filter and M_∞ its limit. Because of equation (1.2), $\inf f(\Omega \cap M_k) \geq l(M_k) \to \gamma^*$, hence $\inf f(\Omega \cap M_\infty) \geq \gamma^*$. Furthermore, if $M \in \mathcal{R}_k$ for all k then $\inf f(\Omega \cap M) \geq l(M) \geq l(M_k) \to \gamma^*$ as $k \to \infty$, i.e. $\inf f(\Omega \cap M) \geq \gamma^*$. Thus the selection procedure is exact and, by theorem 1.4.1, the Branch-and-Bound algorithm converges.

1.5 Mathematical formulation and convex relaxation for nonconvex NLPs

This thesis is concerned with techniques for the determination of global optima of nonconvex NLPs of the general form (1.1). We are particularly interested in improving the performance of branch-and-bound algorithms of the type outlined in Section 1.4. As indicated by the literature review presented in Section 1.3.2 this has been an area of much active research worldwide, especially over the past decade.

Albeit themselves relatively new, branch-and-bound algorithms for global optimization have many similarities to those which have been routinely used for the solution of mixed integer linear programming (MILP) problems for more than four decades – indeed, both classes of algorithms[5] conform to the general Branch-and-Select framework described in Section 1.4. It is, therefore, instructive to draw analogies between these two closely related types of algorithm when trying to identify directions for fruitful research. In particular, the experience of applying branch-and-bound algorithms to MILPs points out the extreme importance of mathematical formulation. More specifically, the same engineering problem (e.g. process scheduling) may often be expressed mathematically in two or more different ways. Although all of these formulations are mathematically completely equivalent, the computational effort required for their solution is very different, often varying by several orders of magnitude. This indicates that the issue of formulation is likely to be central to devising efficient global optimization solution methods, perhaps much more so than, for instance, algorithmic details such as rules for selecting branching variables.

Better mathematical formulations for specific NLPs may sometimes be devised on the basis of insight and intuition. However, it is clear that methods which could *automatically* reformulate wide classes of nonconvex NLPs to a better behaved form would be highly desirable.

A topic that is closely related to that of mathematical formulation is that of the convex relaxation of a given nonconvex NLP. This is essential for all sBB algorithms as a means for establishing a lower bound on the objective function in any given region. The analogous convex relaxation for the case of Branch-and-Bound algorithms for MILPs is the "relaxed" linear programming (LP) problem, i.e. one obtained from the MILP by dropping the integrality requirement on (some of the) integer variables. This convex relaxation is universally used by MILP solution algorithms, and therefore, a "good" mathematical formulation for an MILP is usually considered as one which exhibits the smallest gap between the optimal objective function of the original MILP and that of its LP relaxation.

Interestingly, a significant difference between branch-and-bound algorithms for MILPs and those for nonconvex NLPs is that, in the latter (sBB) case, there is no single convex relaxation of a nonconvex NLP. Consequently, albeit still closely related, the issues of formulation and convex relaxation are no longer identical. For example, it is one important issue how to formulate an engineering optimization problem so that it exhibits the minimum number of nonconvexities –

[5]In this thesis, we shall follow the established practice of referring to branch-and-bound algorithms for global optimization as *spatial* branch-and-bound (sBB) algorithms in order to distinguish them from those traditionally used for MILPs.

and there may be several ways in which to address this requirement, each leading to a different NLP. Now, given any particular nonconvex NLP formulation, there may then also be more than one way in which to construct its convex relaxation, and selecting the best one is quite another issue.

1.6 Outline of this thesis

In view of the discussion in the previous section, it is clear that both the mathematical formulation and the convex relaxation of nonconvex NLPs deserve close attention. Chapter 2 of this thesis presents a review of the related literature, attempting to come up with a rational classification of the quite diverse approaches and techniques that have been reported to date.

Chapter 3 presents a novel automatic reformulation method for nonconvex NLPs that include linear constraints and bilinear terms. The latter are well-known sources of nonconvexity in a wide range of models of engineering systems, for example whenever an extensive variable (e.g. flowrate) multiplies an intensive one (e.g. composition, enthalpy etc.) within a conservation law. The method presented automatically reformulates the nonconvex NLP to one that (a) involves fewer bilinear terms and more linear constraints than the original, and (b) has a tighter convex relaxation when the remaining bilinear terms are convexified using standard techniques.

The objective function and constraints in nonconvex NLPs usually involve a variety of types of sub-expressions (e.g. bilinear, trilinear and linear fractional terms) and other mathematical functions. Relatively tight convex relaxations exist for many of these categories of expressions. One notable exception is that of terms involving monomials of odd degree (i.e. expressions of the form x^n where n is odd) when the range of the variable x includes zero. These occur often (e.g. as cubic or quintic expressions) in practical applications. Chapter 4 presents a novel method for constructing the convex envelope of such monomials, as well as a tight linear relaxation of this envelope.

The automatic reformulation and convex relaxation techniques outlined above fit within the framework of a general sBB algorithm. The form of this algorithm as discussed in the literature is reviewed in Chapter 5. The software implementation of such an algorithm places special demands since, beyond the usual numerical information required by most optimization codes, a substantial amount of symbolic information must also be made available. The architecture of $oo\mathcal{OPS}$, a general object-oriented software framework that attempts to fulfill these, is also presented in Chapter 5.

Finally, Chapter 6 summarizes the main points of the work presented and draws some general conclusions.

Chapter 2

Overview of reformulation techniques in optimization

A key theme of this thesis is the issue of problem reformulations that can be carried out in an automatic manner via symbolic and numerical procedures. Such reformulations can be used for various tasks, both prior to the actual problem solution and during the solution process. Many types of automatic reformulations have appeared in the literature; however, to the best of our knowledge, no attempts have been made, to date, to review them and classify them in a systematic manner. This chapter, therefore, attempts to present such a literature review.

A reformulation of an optimization problem P is a problem P' which shares some mathematical properties with P. A reformulation may be useful as part of an algorithmic procedure to solve P, or if it offers qualitative or quantitative insights regarding the properties of P.

A reformulation P' is *exact* if the global solution $x \in \mathbb{R}^n$ of P can be directly inferred from the global solution x' of P' ("directly inferred" meaning that x can be computed from x' in linear time depending on n). An exact reformulation is *convenient* if finding x' requires less computational resources than finding x.

A useful reformulation which is not exact is usually called a *relaxation* of the problem. For example, many algorithms for the solution of discrete optimization problems involve a continuous relaxation of the problem (i.e. the discrete variables are reformulated to continuous variables). Most Branch-and-Bound techniques for nonconvex optimization problems use a convex relaxation of the problem to compute the lower bound at each iteration.

Reformulations can be constructed by using algebraic-symbolic manipulation of the equations in the original problem, or by using numerical computations. In both cases, it is necessary to prove theoretically, or at least offer strong evidence, that the reformulation is indeed useful. If a reformulation is sometimes useful but it cannot always be decided *a priori* whether it will be useful or not, and to what extent, then it is called *heuristic*.

In the rest of this chapter we shall give a detailed account of the most useful reformulation techniques used in conjunction with deterministic methods for global optimization.

2.1 Reformulations to standard forms

Closely associated with the idea of a reformulation is the concept of a standard form for an optimization problem. Solution algorithms often require the optimization problem to be in a pre-specified form called the *standard form* with respect to that algorithm. Most reformulations are meant to unearth some structural mathematical property of P which was not evident before, or to transform P into a standard form.

In this section we refer to optimization problems in the following form:

$$\min_{x\in\Omega} f(x) \tag{2.1}$$

where Ω is the feasible region.

Notation-wise, by $Ax = b$ we mean a system of m linear equations in n variables, where A is an $m \times n$ matrix, $x \in \mathbb{R}^n$ (unless specified otherwise) and $b \in \mathbb{R}^m$. Likewise, we express a system of linear inequalities as $Ax \le b$. By $x^L \le x \le x^U$ we mean a set of ranges on the variables: $\forall i \le n(x_i^L \le x_i \le x_i^U)$, and $x^L, x^U \in \mathbb{R}^n$.

2.1.1 Box-constrained problems

An optimization problem (2.1) is box-constrained if the only constraints of the problem are the variable ranges (i.e. $\Omega = \{x \mid x^L \le x \le x^U\}$). In other words, no equation or inequality constraints are present (sometimes such problems are called *unconstrained problems*, though in fact a truly unconstrained optimization problem would lack variable bounds as well). This is a very well-studied and interesting class of problems [35, 14, 38], and much effort has gone into reformulations of other types of problems to this type. Moreover, most stochastic optimization methods are originally devised for box-constrained problems, and are then extended to deal with constraints.

2.1.1.1 Penalty and barrier functions

The most widely used reformulation for eliminating equation or inequality constraints from constrained optimization problems is to employ *penalty* or *barrier* functions. This embeds the constraints into the objective function f itself, so that f attains high values at infeasible points; hence a minimization of the objective function automatically excludes the infeasible points.

Consider the optimization problem (2.1) where:

$$\Omega = \{x \in \mathbb{R}^n \mid \forall i \leq m\ (h_i(x) = 0) \wedge \forall i \leq m'\ (g_i(x) \leq 0) \wedge x^L \leq x \leq x^U\}. \tag{2.2}$$

Let $\delta_i, \varepsilon_i : \mathbb{R} \to \{0, \infty\}$ be boolean functions that specify whether a constraint is satisfied: for all $i \leq m$ define $\delta_i(h_i(x)) = 0$ if $h_i(x) = 0$ and $\delta_i(h_i(x)) = \infty$ if $h_i(x) \neq 0$. Likewise for inequality constraints, for all $i \leq m'$ define $\varepsilon_i(g_i(x)) = 0$ if $g_i(x) \leq 0$ and $\varepsilon_i(g_i(x)) = \infty$ otherwise. Now, reformulating the problem to:

$$\min_{x^L \leq x \leq x^U} F(x) = f(x) + \sum_{i=1}^{m} \delta_i(h_i(x)) + \sum_{i=1}^{m'} \varepsilon_i(g_i(x)) \tag{2.3}$$

is an exact reformulation of the original problem to a box-constrained form. If x is feasible in the original problem, then the objective function F of the reformulation reduces to $f(x)$; if it is infeasible, F becomes ∞.

The main disadvantage of this reformulation is that it is highly non-smooth. Furthermore, the introduction of infinity in the definition of δ_i, ε_i is questionable from a numerical point of view. The latter point can be disposed of in case the function f is Lipschitz (i.e. there is a real constant $M > 0$ such that for all $x, y \in \Omega$ we have $|f(x) - f(y)| \leq M||x - y||$), in which case it is possible to find a global upper bound L of f on Ω, and ∞ can be replaced by L. The non-smoothness is a more delicate question. Unless one devises an entirely symbolic method for solving (2.3), one has to replace the functions δ_i, ε_i with smoother functions. Unfortunately, this means that the reformulation may no longer be exact. In [14, 35] a host of different penalty functions are introduced, and their properties are discussed. The simplest "realistic" penalty function involves a reformulated objective function $F(x) = f(x) + \mu\left(\sum_{i=1}^{m} |h_i(x)| + \sum_{i=1}^{m'} \max(0, g_i(x))\right)$. In this case the main drawback is the determination of the parameter μ. There is a value of μ such that this reformulation is exact, but it cannot be known *a priori*. Furthermore, absolute value and pointwise maximum are not smooth functions everywhere. Other penalty and barrier functions involve logarithms, inverses, powers and so on.

2.1.1.2 Lagrangian and Lagrange coefficients

The *Lagrangian* function of an optimization problem in form 2.1, where Ω is defined as in equation (2.2), is defined as $L(x,\lambda,\mu) = f(x) + \sum_{i=1}^{m} \lambda_i h_i(x) + \sum_{i=1}^{m'} \mu_i g_i(x)$ with the requirement that $\mu_i \geq 0$ for all $i \leq m'$. The parameters λ_i, μ_i are called the *Lagrange multipliers*. The Lagrangian is often used in deriving theoretical conditions of optimality in methods of local optimization (Karush-Kuhn-Tucker necessity and sufficiency conditions, see [38, 85]).

In fact, the theory of Lagrange multipliers is also linked to duality theory for general optimization problems. It can be shown that, for a convex problem P, its Lagrangian dual:

$$\max_{\lambda;\mu>0} \min_{x^L \leq x \leq x^U} L(x,\lambda,\mu) \tag{2.4}$$

has exactly the same solution as P. For a nonconvex problem it can be guaranteed only that the solution of the dual problem is a lower bound to the solution of the original problem (2.2). The difference between the value of the dual objective function and the original objective function is called the *duality gap*. In view of the fact that the solution to the dual problem (2.4) is often used as a lower bounding technique in Branch-and-Bound algorithms, it is important to find methods to reduce the duality gap.

There are various such methods: in [17] a suitable partition of the variable ranges is employed. The dual problem is then solved over each subset of the partition. In [70, 69] the Lagrangian $L(x,\lambda,\mu)$ is reformulated to:

$$L_p(x,\lambda,\mu) = f^p(x) + \sum_{i=1}^{m} \lambda_i h_i^p(x) + \sum_{i=1}^{m'} \mu_i g_i^p(x)$$

where $p \geq 1$ is an exponent. The reformulated Lagrangian has local convexity properties in correspondence of local solutions of the original problem. More precisely, under some further regularity conditions, the Hessian of L_p is positive definite at x^* (a locally optimal solution of the original problem (2.2)) for large enough p. This is a sufficient condition for L_p to be locally convex in a neighbourhood of x^*, and hence for L_p to be a valid lower bounding function for the original problem.

2.1.2 Separable problems

Separable programming problems are useful because the objective function can be expressed as a sum of functions of one variable only. Thus, each term in the sum is independent of the other terms. Separable problems have a special structure which offers wide scope for decomposition strategies. If the problem is separable and box-constrained (i.e. $\Omega = \{x \mid x^L \leq x \leq x^U\}$), then it can easily be solved to global optimality via interval analysis (see Section 2.1.2.2). If linear or nonlinear constraints are present, interval analysis offers a very fast way to calculate lower bounds on the objective function: it is no surprise that the first Branch-and-Bound approaches to global optimization were restricted to separable problems [34, 117, 15].

Let $x = (x_1, \ldots, x_n) \in \Omega$. A function $f : \mathbb{R}^n \to \mathbb{R}$ is *separable* if and only if there are $f_1, \ldots, f_n : \mathbb{R} \to \mathbb{R}$ such that for all $x \in \Omega$ we have $f(x) = \sum_{i=1}^n f_i(x_i)$.

If the objective function f of a problem (2.1) is separable, then the problem is separable. The definition of the feasible region Ω varies. Usually [14, 35, 79], Ω is defined by a system of separable inequalities $\sum_{j=1}^n g_{ij}(x_j) \leq b_i$, $\forall i \leq m$, where $g_{ij} : \mathbb{R} \to \mathbb{R}$ for each i, j. Some authors [27, 58] require Ω to be a polytope.

2.1.2.1 Separation of bilinear forms

In the procedure of Section 2.1.8.1, the basic idea is the separation of a bilinear form xy. This idea rests on the relationship $(x+y)^2 = x^2 + 2xy + y^2 \Rightarrow xy = \frac{1}{2}(w^2 - x^2 - y^2)$ and $w = x + y$. It was already known in the 18th century that the quadratic form $\sum_{i,j=1}^n a_{ij} x_i x_j$ could always be reduced to a sum or difference of squares:

$$y_1^2 + \ldots + y_r^2 - y_{r+1}^2 - \ldots - y_{r+t}^2 \tag{2.5}$$

via a real linear transformation $x_i = \sum_j b_{ij} y_j$ (for all $i \leq n$) having non-zero determinant. Cauchy, Sylvester and Jacobi all worked on this problem. If $t = 0$, then the form is called *positive definite*; if $t > 0$ then the form is called *semidefinite* (these terms were introduced by Gauss in his *Disquisitiones arithmeticae* [62]). Their work is also relevant to the theory of convex, concave and d.c. functions, since a positive quadratic term is a convex function and a negative quadratic term is a concave function.

The problem of reducing quadratic forms is tightly linked to the problem of diagonalizing a square matrix. Let $x = (x_1, \ldots, x_n)$, $s = (s_1, \ldots, s_n)$ and $A = (a_{ij})$ be an $n \times n$ matrix. The expression $x^T A s = \sum_{i,j}^n a_{ij} x_i s_j$ is called a *generalized bilinear form*. We are specially interested in the case where $x = s$. To reduce a generalized bilinear form to a semidefinite quadratic form, we have to find a non-singular transformation $x = Py$, where P is an $n \times n$ matrix and $y = (y_1, \ldots, y_n)$ such that $x^T A x = y^T P^T A P y$ and $P^T A P$ is diagonal. Constructing such a matrix P is a classical problem in matrix theory (see for example [76], p. 322).

Reformulating generalized bilinear forms to semidefinite quadratic forms is usually convenient. It is an exact reformulation and it often gives rise to better convex relaxations (see Section 2.3).

2.1.2.2 Global solution of separable box-constrained problems

Finding the global optimum of the separable box-constrained problem

$$\min_{x^L \leq x \leq x^U} \sum_{i=1}^{n} f_i(x_i)$$

reduces to finding the global optimum of each of the one-dimensional problems

$$\min_{x_i^L \leq x_i \leq x_i^U} f_i(x_i),$$

and can therefore be totally decomposed. Interval arithmetic provides an extremely fast and effective way to find the bounds on a univariate function $f_i(x_i)$ given the range $[x_i^L, x_i^U]$ of x_i.

Let $\mathcal{N} = \{N_1, \ldots, N_m\}$ be a partition of $\{1, \ldots, n\}$, and for each $i \leq m$ let $x^{[i]} = \{x_j \mid j \in N_i\}$. The function $f : \mathbb{R}^n \to \mathbb{R}$ is *semi-separable* if and only if for each $i \leq m$ there exist functions $f_i : \mathbb{R}^{|N_i|} \to \mathbb{R}$ such that $f(x) = \sum_{i=1}^m f_i(x^{[i]})$. The method described in the example below also works when applied to problems with semi-separable objective functions.

2.1.1 Example

Find the global minima of the problem:

$$\begin{aligned}
\min_{x_1,x_2,x_3,x_4} \quad f(x) = \quad & x_1x_2 + x_3^4 - 10x_3^3 + \cos(e^{x_4}) \\
& -1 \leq x_1, x_2 \leq 1 \\
& -5 \leq x_3 \leq 10 \\
& 0 \leq x_4 \leq 2.
\end{aligned}$$

Notice f is semi-separable, with $f_1(x_1, x_2) = x_1x_2$, $f_2(x_3) = x_3^4 - 10x_3^3$ and $f_3(x_4) = \cos(e^{x_4})$. Interval analysis on the product x_1x_2 shows that the global minima of $f_1(x_1, x_2)$ in the specified *range are attained at* $(x_1, x_2) = (-1, 1)$ *and* $(x_1, x_2) = (1, -1)$. *The global minimum of* $f_2(x_3)$ *can be found either algebraically, as* $f_2'(x_3) = 4x_3^3 - 30x_3^2 = 2x_3^2(2x_3 - 15)$ *implies a global minimum at* $x_3 = \frac{15}{2}$, *or numerically, by applying Newton's method in one dimension for finding the roots of the derivative. Finally, the global minimum of* $f_3(x_4) = \cos(e^{x_4})$ *can be found by setting* $\cos(e^{x_4}) = -1$, *which implies* $e^{x_4} = (2k+1)\pi$ *for all* $k \in \mathbb{Z}$, *i.e.* $x_4 = \log((2k+1)\pi)$. *Given the range of* x_4, *we have to impose* $k = 0$ *and so* $x_4 = \log(\pi) = 1.14473$. *Thus the global solutions to the problem are at* $(x_1, x_2, x_3, x_4) = (\pm 1, \mp 1, \frac{15}{2}, \log(\pi))$.

2.1.3 Linear problems

A linear optimization problem is such that both the objective function and the constraints are linear functions in the problem variables. A very efficient global solution method (called the *simplex method*) for linear continuous optimization problems was proposed in the 1940s, by Dantzig [27], and has been further refined since then. Nowadays there are many efficient software codes to solve large-scale linear problems.

Unfortunately, linear reformulations are very rarely exact. In fact, most linear reformulations are linear relaxations and are used within more complex methods for the solution of nonlinear problems (like e.g. Branch-and-Bound). We will analyze them in Section 2.4.

2.1.3.1 Reformulating quadratic binary problems to linear binary problems

Although this reformulation only applies to a very special class of optimization problems, it is one of the very few exact linear[1] reformulations [16, 83]. Any unconstrained quadratic binary problem $\max_{x \in \{0,1\}^n} x^T Q x$ can be reformulated exactly to:

$$\begin{array}{rl}
\max & q^T y \\
s.t. & \forall i, j \le n \; (y_{ij} \le x_i) \\
& \forall i, j \le n \; (y_{ij} \le x_j) \\
& \forall i, j \le n \; (y_{ij} \ge x_i + x_j - 1) \\
& x \in \{0,1\}^n, y \in \{0,1\}^{n^2},
\end{array}$$

[1]Since the considered problems are binary, calling this a linear relaxation is a slight abuse of notation. What we mean is that the objective functions and the constraints are linear functions of the decision variables.

where q is a vector consisting of the entries of the $n \times n$ matrix Q ordered by column. Such a reformulation belongs to a class of reformulations called *liftings* because of the fact that they “lift” the geometry of the problem into a higher dimensional Euclidean space (i.e. they add new variables to the problem). Although usually a problem with more variables is more difficult to solve, liftings can nonetheless be useful.

2.1.4 Convex problems

An optimization problem is convex if both the objective function f and the feasible region Ω are convex. This is an interesting class of problems as it is possible to show that any local solution of a convex problem is also a global solution [85, 98]. As in the linear case, convex reformulations are rarely exact. However, convex relaxations (which will be analysed in more detail in Section 2.3) are used within more complex procedures for the solution of nonlinear problems. Their global minimality property makes it possible to compute lower bounds of the objective function of nonconvex problems in given subregions of the feasible region (this notion is used in most spatial Branch-and-Bound algorithms for the solution of NLPs and MINLPs [116, 4]).

One exact convex reformulation can be obtained when the objective function is a positive definite bilinear form (see Section 2.1.2.1). If $t = 0$ in equation (2.5), then the reformulation is exact and it makes the function completely convex.

Sometimes a nonconvex function can be reformulated exactly to a convex function by a nonlinear change of variables [98]. The function $f(x) = ax_1 \cdots x_n$ is nonconvex. For all $i \leq n$ let $X_i = \log x_i$; then $f(X) = ae^{X_1 + \ldots + X_n}$, which is convex. This, with a suitable adaptation, also applies if the variables are discrete [95]. The following, for example, is a mixed-integer reformulation of the function $ax_1^{r_1} \cdots x_n^{r_n}$ where x_i are discrete variables for all $i \leq n$:

$$\left\{ \begin{array}{l} ax_1^{r_1} \cdots x_n^{r_n} \\ a > 0, \forall i \leq n \ (r_i \in \mathbb{R}) \\ \forall i \leq n \ (x_i \text{ discrete}) \end{array} \right. \quad \Leftrightarrow \quad \left\{ \begin{array}{ll} ae^{r_1 X_1 + \ldots + r_n X_n} & \\ X_i = \log d_{i1} + \sum_{j=1}^{n_i - 1} \beta_{ij} (\log d_{i,j+1} - \log d_{i1}) & \qquad (*) \\ \sum_{j=1}^{n_i - 1} \beta_{ij} \leq 1 & \\ X_i \in \mathbb{R}, \beta_{ij} \in \{0, 1\}, & \end{array} \right.$$

where the discrete variable x_i can take values in $\{d_{i1}, \ldots, d_{in_i}\}$. For an explanation of the equality constraints $(*)$, see Section 2.1.5.1.

In [95], another type of reformulation is presented, that deals with terms of the form $-ax^p y^q$ with $a, p, q > 0$. If $p + q \leq 1$ the term is convex, therefore we only need to apply the reformulation when $p + q > 1$. If we let $x = X^{\frac{1}{p+q}}$ and $y = Y^{\frac{1}{p+q}}$, the original term is transformed in

$-aX^{\frac{p}{p+q}}Y^{\frac{q}{p+q}}$ which is convex as the sum of the exponents is obviously less than or equal to 1. If x, y were discrete variables taking values in $\{d_1, \ldots, d_n\}$ and $\{c_1, \ldots, c_m\}$ respectively, then the following integrality constraints would also be necessary:

$$\begin{aligned}
X &= d_1^{p+q} + \sum_{i=1}^{n-1} \beta_i (d_{i+1}^{p+q} - d_1^{p+q}) \\
Y &= c_1^{p+q} + \sum_{i=1}^{n-1} \alpha_i (c_{i+1}^{p+q} - c_1^{p+q}) \\
\sum_{i=1}^{n-1} \beta_i &\leq 1 \\
\sum_{i=1}^{m-1} \alpha_i &\leq 1 \\
\alpha_i, \beta_i &\in \{0, 1\}.
\end{aligned}$$

2.1.5 Binary problems

A binary optimization problem is such that the feasible region Ω is a subset of B^n, where $B = \{0, 1\}$ (i.e. the problem variables can take only values 0 and 1). The solution space of these problems is well suited to Branch-and-Select type searches. Each node of the search tree has exactly two subnodes [9]. From a theoretical point of view, it is important because it is the simplest type of discrete problem, and theorems can be stated and proved more simply if variables can only take two values (see e.g. [109], p. 1277, Theorem 1).

2.1.5.1 Reformulating discrete problems to binary problems

A discrete optimization problem is such that the problem variables can take only discrete values. It is possible to reformulate discrete problems to binary problems exactly [27, 144, 110]. Let v be a discrete problem variables taking values in the set $\{v_1, \ldots, v_d\}$. By introducing d new binary variables $\beta_1 \ldots \beta_d$, we can replace v by $\sum_{i=1}^{d} v_i \beta_i$ and add a linear constraint $\sum_{i=1}^{d} \beta_i = 1$ to the definition of the feasible region Ω. This is an exact reformulation: the linear constraint ensures that exactly one binary variable β_i takes the value of 1, so that the expression for v takes exactly one value in $\{v_1, \ldots, v_d\}$. This reformulation involves only linear functions, so in this sense it does not add significant complexity to the problem. This reformulation is a lifting.

A slightly better reformulation (in the sense that it requires one less binary variable) is the following [95]:

$$v = v_1 + \sum_{i=1}^{n-1} \beta_i (d_{i+1} - d_1) \quad \wedge \quad \sum_{i=1}^{n-1} \beta_i \leq 1,$$

where $\beta_i \in \{0, 1\}$ for any $i \leq n - 1$.

2.1.5.2 Reformulating binary problems to continuous problems

Sometimes it is more convenient to approach a discrete problem with continuous methods. To this end, the following quadratic exact reformulation can be used: substitute each binary variable $y \in \{0, 1\}$ with a continuous variable $\bar{y} \in [0, 1]$ and add the equality constraint $\bar{y} = \bar{y}^2$ to the formulation of the problem. This constraint, called an *integrality enforcing constraint*, is equivalent to the equation $\bar{y}(\bar{y} - 1) = 0$ which has solutions 0 and 1. Hence, any feasible solution to the continuous problem satisfying the integrality enforcing constraint will be such that $\bar{y} = 0$ or $\bar{y} = 1$.

This reasoning can be generalized to any discrete variable $y \in S = \{a_i \mid i \in \mathbb{N}\}$. Thus, we can relax y to a continuous variable $\bar{y} \in [\min S, \max S]$ and add an equality constraint $g(\bar{y}) = 0$ having S as solution set. If S is finite, a possible function is $g(\bar{y}) = \prod_{i=1}^{|S|} (\bar{y} - a_i)$.

It is worth mentioning here that when trying to incorporate this relaxation within a Branch-and-Bound algorithm which requires convex relaxations to find the lower bounds of the objective function in each region, one needs to keep in mind that the function $g(\bar{y})$ will be replaced by its convex and concave relaxations, so g should be chosen so that the "convexity gap" between g and its relaxations is minimal (see also Section 2.3 about convex relaxations). In particular, Smith showed ([114], p. 209-210) that the simplest linear relaxation of the quadratic integrality enforcing constraint $\bar{y} = \bar{y}^2$ just reduces to $\bar{y} \in [0, 1]$ and hence no integrality is enforced on $\bar{y}$ when solving the relaxation.

Numerically, integrality enforcing constraints like $y = y^2$ are usually problematic for most nonlinear local solver codes. Thus, the idea of reformulating a large-scale binary problem to a continuous problem having high numbers of these constraints is not a workable one. However, such constraints are sometimes useful in MINLPs having few integer variables.

2.1.6 Concave problems

A concave optimization problem in form 2.1 is such that the objective function f is concave and the feasible region Ω is convex. Concave optimization is, in a certain sense, the simplest case of a complicated optimization problem: it is multi-extremal (i.e. in general, it has many local minima) so that the techniques of local optimization are not sufficient to find the global optimum. Its complexity is high enough to allow many different optimization problem families to be reformulated exactly to a concave problem. However, its formulation is simple enough that efficient algorithms for its global solution can be designed [58, 18].

2.1.6.1 Reformulating binary problems to concave problems

Let C be a convex set (which may be defined by a set of constraints), $B = \{0,1\}$, $\bar{B} = [0,1]$ and consider a binary problem in the form $\min\{f(x) \mid x \in C \cap B^n\}$, where f is Lipschitz and twice continuously differentiable on $\bar{B}^n$. Then there exists $\mu_0 \in \mathbb{R}$ such that, for all $\mu > \mu_0$, the binary problem above can be reformulated exactly to:

$$\min_{x \in C \cap \bar{B}^n} F(x) = f(x) + \mu \sum_{i=1}^{n} x_i(1 - x_i),$$

and $F(x)$ is concave on $\bar{B}^n$. For the proof of this theorem see [58], p. 15. One convenient feature of this exact reformulation is that it adds no new variables to the problem. Furthermore, it succeeds in relaxing the binary variables to continuous whilst still keeping the reformulation exact.

2.1.6.2 Reformulating bilinear problems to concave problems

The bilinear programming problem is stated in general terms as:

$$\min_{x \in X, y \in Y} f(x,y) = px + xQy + qy \tag{2.6}$$

where X, Y are convex polyhedral sets (defined by sets of linear constraints). Assume Y has at least one vertex and for every $x \in X$ the problem $\min_{y \in Y} f(x,y)$ has a solution. Then problem (2.6) can be reformulated exactly to a concave minimization problem with piecewise linear objective function and linear constraints [58]. In particular, let $V(Y)$ be the set of vertices of Y.

Since the solution of a linear problem is attained at least at one point in $V(Y)$, we have

$$\begin{aligned}\min_{x\in X, y\in Y} f(x,y) &= \min_{x\in X}\min_{y\in V(Y)} f(x,y) = \\ &= \min_{x\in X} f(x),\end{aligned}$$

where $f(x) = \min_{y\in V(Y)} f(x,y)$. The set $V(Y)$ is finite and for each $y \in V(Y)$, $f(x,y)$ is a linear function of x. Thus $f(x)$ is the pointwise minimum of a finite family of linear functions, and hence is concave and piecewise linear.

2.1.6.3 Reformulating complementarity problems to concave problems

Complementarity problems are feasibility problems rather than optimization problems. The problem is to find $x \in \Omega \subseteq \mathbb{R}^n$ such that for all $i \le m$ we have $g_i(x) \ge 0$, $h_i(x) \ge 0$ and $g_i(x)h_i(x) = 0$, where $g_i, h_i : \mathbb{R}^n \to \mathbb{R}$. Complementarity problems arise in the study of other optimization problems, in the analysis of the computation of equilibria in fixed-point problems (theory of games) and in modelling the logical notion of disjunction (which concerns the modelling of constraints in certain combinatorial problems where a constraint is to be enforced only if a certain binary variable is true) [86].

Assume Ω is convex and g_i, h_i are concave functions for each $i \le m$, and that the complementarity problem has a solution x^*. In this case, the complementarity problem can be reformulated exactly to the following concave optimization problem:

$$\left.\begin{array}{rcl}\min_{x\in\Omega} f(x) &=& \sum_{i=1}^m \min\{g_i(x), h_i(x)\} \\ \text{s.t.} && \forall i \le m\ (g_i(x) \ge 0, h_i(x) \ge 0)\end{array}\right\}$$

For the proof of this statement, see [58], p. 24.

2.1.6.4 Reformulating max-min problems to concave problems

A max-min optimization problem is stated as:

$$\left.\begin{array}{rl}\max_{x\ge 0}\min_{y\ge 0} & ax + by \\ \text{s.t.} & Ax + By \le c\end{array}\right\}$$

where $x, a \in \mathbb{R}^n$, $y, b \in \mathbb{R}^m$, A is an $s\times n$ matrix, B is an $s\times m$ matrix and $c \in \mathbb{R}^s$. This problem can be reformulated exactly to a concave optimization problem $-\min_{x\in P}(-f(x) - ax)$, where:

$$\begin{aligned}P &= \{x \ge 0 \mid \exists y \le 0\ (Ax + By \le c)\} \\ f(x) &= \min\{by \mid By \le c - Ax \wedge y \ge 0\}.\end{aligned}$$

Note that $f(x)$ is a convex piecewise linear function on P, so that $-f(x) - ax$ is concave and piecewise linear.

2.1.7 D.c. problems

An optimization problem is *d.c.* if the objective function is d.c. and Ω is a d.c. set (see Section 1.1) [132, 121, 120]; in fact in [121], the objective function is only required to be upper semicontinuous (i.e. for all x in the domain of f we have $f(x) = \lim_{y\to x} \sup f(y)$; the definition of lower semicontinuity is similar, with sup replaced by inf, and both semicontinuities at one point imply ordinary continuity at that point). In [59], the feasible region is defined as $C \cap M$, where $M = \{x \mid \forall i \le m\ (g_i(x) \le 0 \wedge g_i \text{ is d.c.})\}$.

D.c. programming problems have two fundamental properties. The first is that the space of all d.c. functions is dense in the space of all continuous functions. This implies that any continuous optimization problem can be approximated as closely as desired, in the uniform convergence topology, by a d.c. optimization problem [132, 59]. The second property is that it is possible to give explicit necessary and sufficient global optimality conditions for certain types of d.c. problems [132, 121]. Some formulations of these global optimality conditions [120] also exhibit a very useful algorithmic property: if at a feasible point x the optimality conditions do not hold, then the optimality conditions themselves can be used to construct a better point x'.

2.1.7.1 Reformulating continuous functions to d.c. functions

Each twice differentiable function $f : \mathbb{R}^n \to \mathbb{R}$ is d.c, since $g(x) = f(x) + \rho x^T x$ is a convex function for all sufficiently large ρ. However, since finding a good estimate for ρ is itself a difficult problem [7], there is to date no automatic efficient procedure to reformulate any given continuous problem to a d.c. problem. However, some progress in this field has been made for the case of separable functions [67]. Assuming $f(x) = \sum_{i=1}^n f_i(x_i)$ and each f_i is continuous but neither concave nor convex, then it is possible to reformulate f exactly to a d.c. function. Because f is separable and since the procedure is applied to each f_i individually, we shall dispose of the index i here and just consider $f(x)$ as a function of only one variable x. We suppose that $x \in [a, b]$ and aim to find functions $p, q : \mathbb{R} \to \mathbb{R}$ such that p is concave, q is convex and $f(x) = p(x) + q(x)$.

First of all find a partition $\mathcal{N} = \{[a_i, b_i] \mid i = 1, \ldots, r\}$ of $[a, b]$ such that:

1. $a_1 = a$, $b_r = b$;
2. for all $i < r$, we have $b_i = a_{i+1}$;
3. for all $i < r$, either f restricted to $[a_i, b_i]$ is convex and f restricted to $[a_{i+1}, b_{i+1}]$ is concave, or the reverse is true (i.e. $f|_{[a_i,b_i]}$ is concave and $f|_{[a_{i+1},b_{i+1}]}$ is convex).

Basically $a_2 = b_1, \ldots, a_r = b_{r-1}$ are the points where f changes between concavity and convexity[2]. Such a partition can be found by studying the behaviour of the derivatives $f'(x), f''(x)$. We assume without loss of generality that f is concave on the first subinterval $[a_1, b_1]$ (if f is convex on the first subinterval just set $a_1 = b_1$). This makes it possible to infer that, if i is odd, then f is concave on $[a_i, b_i]$; if i is even, then f is convex on $[a_i, b_i]$.

Now let:

$$\Delta_i = \begin{cases} \min\{f'_-(b_i), f'_+(b_i)\} & \text{if } i \text{ is odd} \\ \max\{f'_-(b_i), f'_+(b_i)\} & \text{if } i \text{ is even} \end{cases}$$

where $f'_-(x)$ is the left derivative and $f'_+(x)$ is the right derivative of f at x, and define two affine functions $s_i, t_i : [a, b] \to \mathbb{R}$ as follows:

$$\begin{aligned} s_i(x) &= f(b_i) + \Delta_i(x - b_i) \\ t_i(x) &= \sum_{k=1}^{i} (-1)^{k+i} s_k(x). \end{aligned}$$

If we now define:

$$\begin{aligned} p(x) &= \begin{cases} f(x) - t_{i-1}(x) & \text{if } x \in D_i \text{ and } i \text{ is odd} \\ t_{i-1}(x) & \text{if } x \in D_i \text{ and } i \text{ is even} \end{cases} \\ q(x) &= \begin{cases} t_{i-1}(x) & \text{if } x \in D_i \text{ and } i \text{ is odd} \\ f(x) - t_{i-1}(x) & \text{if } x \in D_i \text{ and } i \text{ is even,} \end{cases} \end{aligned}$$

then it can be shown [67] that p is concave and q is convex, and that $p(x) + q(x) = f(x)$, so that this reformulation is exact.

2.1.8 Factorable problems

Most common functions can be expressed in factorable form [80]; this form is desirable because it makes it relatively easy to construct convex relaxations in a recursive way [126, 141].

[2]If there are any cusp points, the function might fail to alternate in such a way; in this case we simply define an empty interval where $a_i = b_i$.

A mathematical programming problem is in factorable form if it is written in the following way:

$$\left.\begin{array}{lll} \min_{x\in\mathbb{R}^n} & X^N(x) & \\ -\infty < a_i & \le X^i(x) \le & b_i < \infty,\ i \in [1, N-1] \end{array}\right\} \tag{2.7}$$

where $x = (x_1, \ldots, x_n)$, $X^i(x) = x_i$ for $1 \le i \le n$ and the other X^i expressions are defined recursively as follows: given $X^p(x)$ for $p = 1, \ldots, i-1$ then for $i = n+1, \ldots, N$

$$X^i(x) = \sum_{p=1}^{i-1} T_p^i(X^p(x)) + \sum_{p=1}^{i-1}\sum_{q=1}^{p} V_{q,p}^i(X^p(x))U_{p,q}^i(X^q(x))$$

where T_p^i, U_p^i, V_p^i are continuous functions of one variable.

The algebraic properties of the factorable form make it possible to substitute each nonconvex term (univariate functions and bilinear products) with a convex relaxation of it [126]. The αBB method [13, 2, 6, 5] is based on a variation of the factorable form problem, with extensions to include fractional terms and general twice-differentiable nonconvex terms (which are underestimated by means of a quadratic convex relaxation).

Sherali [113] applied the RLT (Reformulation-Linearization Technique) relaxation technique (see Section 2.4.1) to factorable problems.

2.1.8.1 Reformulation of factorable problems to separable form

In this reformulation [79], each nonseparable term in the objective function and the constraints is recursively replaced by an equivalent separable expression, until no nonseparable terms remain. The two steps to repeat are:

1. Replace any product term of the form $q_1(x)q_2(x)$ by $y_1^2 - y_2^2$ and add the constraints $q_1(x) = y_1 - y_2$ and $q_2(x) = y_1 + y_2$ to the definition of Ω.
2. Replace any term of the form $T(t(x))$ by $T(y)$ and add the constraint $t(x) = y$ to the definition of Ω.

It can be shown that the class of problems that can be reformulated to separability via the procedure above loosely corresponds to the class of factorable problems. In fact, it is possible to expand this class to include terms like $q_1(x)^{q_2(x)}$: replace this term by a new variable y and add the following constraints to the definition of Ω:

$$\begin{aligned} y_1 &= q_1(x) \\ y_2 &= q_2(x) \\ y_3 &= \log(y_2) \\ y_4 &= y_1 + y_3 \\ y_5 &= \frac{1}{2}(y_4^2 - y_1^2 - y_3^2) \\ y &= e^{y_5}. \end{aligned}$$

2.1.9 Smith's standard form

This standard form, a symbolic exact reformulation for NLPs and MINLPs in general form, was first proposed in [114]. It will be discussed in detail in Section 5.2.2.1 as it forms a fundamental part of Smith's Branch-and-Bound algorithm, on which a considerable proportion of this thesis is based. This reformulation is an automatic process that isolates the nonlinear terms of the problem (1.1) in a list of simple constraints which are easy to deal with algorithmically. The following example will suffice, for now, to explain the essence of this reformulation.

2.1.2 Example

In order to reduce the following problem:

$$\min_{x\in\Omega} f(x_1, x_2) = \log(x_1 x_2)e^{(x_1+x_2)} + 2x_1$$

to Smith's standard form, we define new problem variables $w_1, \ldots, w_5$ *via the following list of constraints:*

$$\begin{aligned} w_1 &= x_1 x_2 \\ w_2 &= x_1 + x_2 \\ w_3 &= \log(w_1) \\ w_4 &= e^{w_2} \\ w_5 &= w_3 w_4. \end{aligned}$$

We can now express the objective function f *in terms of these new variables* $f(w_5, x_1) = w_5 + 2x_1$*, by adding the above list of constraints to the definition of the feasible region* Ω*.*

Smith's standard form is a lifting (it adds new variables to the problem), so it may not always be convenient; however, the advantage of Smith's standard form is that it is easier for an automatic algorithm to deal with a list of simple constraints like the one above, rather than with a nonlinear problem in the most general formulation (2.1). Such a reformulation makes it easy to implement symbolic algorithms: in short, it is a sort of "starting step" for more complex reformulations.

Smith's standard form can be carried out algorithmically in an extremely efficient way (see section 5.2.2.1). It can be shown that this reformulation is exact: solving a problem in Smith's standard form to global optimality will produce the same solutions as solving the original problem.

2.2 Exact reformulations

In this section, we shall present some exact reformulations which are not considered as standard forms.

2.2.1 Equality/inequality constrained problems

Sometimes it is convenient, for algorithmic reasons, to express an inequality constraint as an equality constraint, and vice versa. Given an inequality $g(x) \leq 0$, it can be reformulated exactly as an equality via the introduction of a *slack variable*. Write $g(x) \leq 0$ as $g(x) + s = 0$ where s is a new problem variable (the slack variable) such that $s \geq 0$. This reformulation is a lifting.

Vice versa, any equation $g(x) = 0$ can be reformulated exactly to a pair of inequalities $g(x) \leq 0$ and $g(x) \geq 0$, without adding any new variable to the problem.

2.2.2 Dimensionality reduction

Cantor proved that $\mathbb{R}^n$ and $\mathbb{R}$ have the same cardinality c (the cardinality of the continuum) by showing that it was possible to construct a bijection between the two sets. Cantor's bijection, however, is not continuous in the usual topology. Peano and Hilbert suggested bijections (known as Peano-type space-filling curves) that were everywhere continuous and nowhere differentiable. These bijections make it possible to devise an exact reformulation of a problem in $\mathbb{R}^n$ to a problem in $\mathbb{R}$, i.e. having a single problem variable.

Given a space-filling curve $g : \mathbb{R} \to [0,1]^n$ (which exists by transfinite cardinality considerations [65, 24]), the dimensionality of the problem can be reduced from n to 1 by solving the reduced problem $\min_{y\in\mathbb{R}} f(g(y))$. The function $f \circ g$ maps $\mathbb{R}$ into $\mathbb{R}$, and may thus be minimized via efficient 1-dimensional optimization methods. If y is a minimizer of $f \circ g$, then $g(y)$ is a minimizer for f.

The difficult part is the choice of an appropriate space-filling function. These functions are usually a set of theoretical tools used to prove existence theorems, and as such are even difficult to describe explicitly. This approach may be worth investigating when the Euclidean space is approximated by a rational or integer lattice. There are some references in the literature which explain how to generate space-filling curves algorithmically [37]. One example is the Hilbert space-filling curve (see Fig. 2.1).

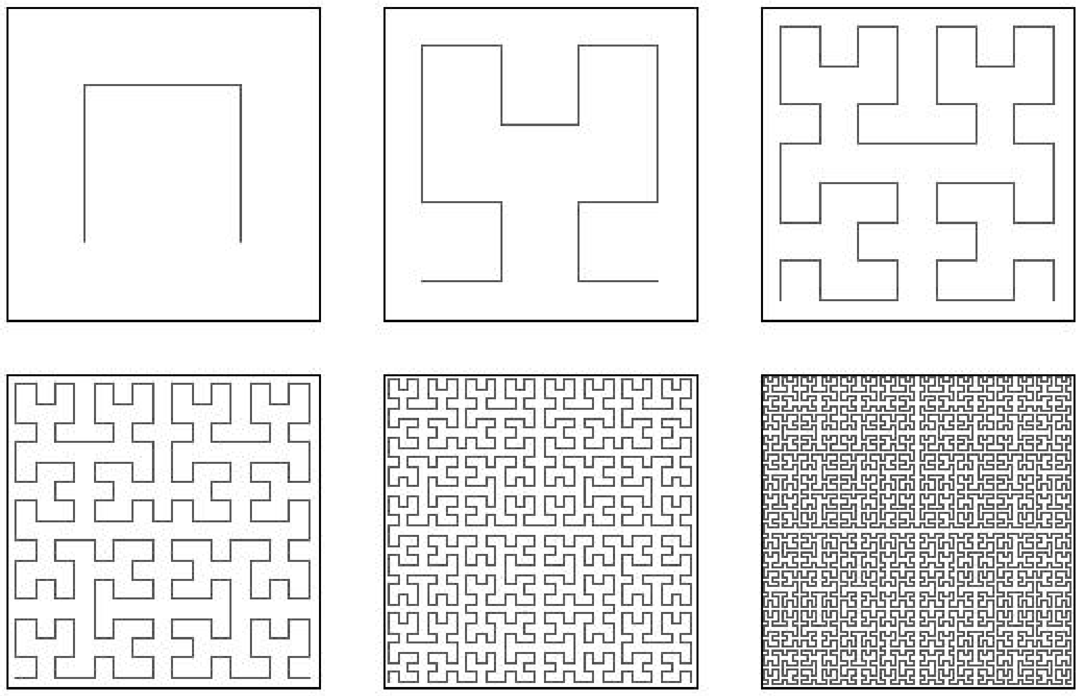


Figure 2.1: The Hilbert space-filling curve in 2-dimensional space.

This technique is based on ideas proposed in the 1970s [122]. Interest in the use of space filling curves for global optimization waned during the 80s, but recently there have been renewed efforts in dimensionality reduction [123, 44, 124].

2.3 Convex relaxations

A relaxation cannot be used to solve a difficult problem directly because the solution of the original problem cannot, in general, be directly inferred from the solution of the relaxation. Relaxations are, however, very important in the field of deterministic global optimization. One of the most important tools in this field is the Branch-and-Bound algorithm, which uses a convex (or linear) relaxation at each step to calculate the lower bound in a region.

Convex relaxations for nonconvex problems in form (2.1) are obtained by substituting the (nonconvex) objective function $f(x)$ with a convex relaxation $\underline{f}(x)$ and the (nonconvex) feasible region Ω with a convex set $\bar{\Omega}$ such that $\Omega \subseteq \bar{\Omega}$. Because every local minimizer of a convex problem is also a global minimizer, solving the convex relaxation:

$$\min_{x \in \bar{\Omega}} \underline{f}(x) \tag{2.8}$$

with a local optimization algorithm will obtain the global solution of problem (2.8) which is guaranteed to be a valid lower bound for the global solution of the original problem (2.1). When the feasible region Ω is defined by equality and inequality constraints, as in equation (2.2), any convex relaxation $\bar{\Omega} \supseteq \Omega$ must be such that all the equality constraints are linear and all inequality constraints are convex[3].

In general, there is more than one possible convex relaxation for any given problem; we therefore look for the best one, i.e. generally the one that gives rise to the greatest possible lower bound. Unfortunately, finding convex relaxations and convex envelopes of arbitrary subsets of $\mathbb{R}^n$ is not an easy task. Usually generic problems (1.1) need to be reformulated to some standard form before it is possible to construct a convex relaxation.

2.3.1 αBB convex relaxation

Floudas and co-workers have proposed a Branch-and-Bound algorithm (called αBB [2, 4, 5, 8]) for general nonconvex twice-differentiable problems. The algorithm aims to solve a problem in form (2.1) where the feasible region Ω is defined as in (2.2). Any nonlinear, twice-differentiable

[3]A more precise characterization is that inequality constraints should be quasi-convex.

function $f(x)$ in the problem, be it the objective function or one of the constraints, can be reformulated exactly as:

$$\begin{aligned} f(x) &= c^T x + f_C(x) + \sum_i b_i x_{B_1(i)} x_{B_2(i)} + \sum_i t_i x_{T_1(i)} x_{T_2(i)} x_{T_3(i)} + \\ &\sum_i d_i \frac{x_{F_1(i)}}{x_{F_2(i)}} + \sum_i r_i \frac{x_{R_1(i)} x_{R_2(i)}}{x_{R_3(i)}} + \sum_i f_{U(i)}(x_i) + \sum_i f_{N(i)}(x), \end{aligned}$$

where:

- $c, x \in \mathbb{R}^n$ and each b_i, t_i, d_i, r_i is a real constant;
- $f_C(x)$ is a general convex function;
- $B_j, T_j, F_j, R_j, U_j, N_j$ are integer functions $\mathbb{N} \to \{1, \ldots, n\}$.
- each $f_{U(i)}$ is a concave univariate function term;
- each $f_{N(i)}$ is a general nonconvex function term.

For a bilinear term xy, McCormick's underestimators [80] are used. A new variable w_B is added to the problem (it replaces the bilinear term xy) and the following inequality constraints are inserted in the relaxed problem:

$$\begin{aligned} w_B &\geq x^L y + y^L x - x^L y^L \\ w_B &\geq x^U y + y^U x - x^U y^U \\ w_B &\leq x^U y + y^L x - x^U y^L \\ w_B &\leq x^L y + y^U x - x^L y^U. \end{aligned}$$

The above linear inequalities have been shown to be the convex envelope of a bilinear term [10]. The maximum separation of the bilinear term xy from its convex envelope $\max(x^L y + y^L x - x^L y^L, x^U y + y^U x - x^U y^U)$ inside the rectangle $[x^L, x^U] \times [y^L, y^U]$ occurs at the middle point $(\frac{x^L + x^U}{2}, \frac{y^L + y^U}{2})$ and is equal to $\frac{(x^U - x^L)(y^U - y^L)}{4}$ [13].

For a trilinear term xyz a new variable w_T is introduced, to replace the trilinear term xyz, together with the following constraints [75]:

$$
\begin{array}{rcl}
w_T & \geq & xy^Lz^L + x^Lyz^L + x^Ly^Lz - 2x^Ly^Lz^L \\
w_T & \geq & xy^Uz^U + x^Uyz^L + x^Uy^Lz - x^Uy^Lz^L - x^Uy^Uz^U \\
w_T & \geq & xy^Lz^L + x^Lyz^U + x^Ly^Uz - x^Ly^Uz^U - x^Ly^Lz^L \\
w_T & \geq & xy^Uz^L + x^Uyz^U + x^Ly^Uz - x^Ly^Uz^L - x^Uy^Uz^U \\
w_T & \geq & xy^Lz^U + x^Lyz^L + x^Uy^Lz - x^Uy^Lz^U - x^Ly^Lz^L \\
w_T & \geq & xy^Lz^U + x^Lyz^U + x^Uy^Uz - x^Ly^Lz^U - x^Uy^Uz^U \\
w_T & \geq & xy^Uz^L + x^Uyz^L + x^Ly^Lz - x^Uy^Uz^L - x^Ly^Lz^L \\
w_T & \geq & xy^Uz^U + x^Uyz^U + x^Uy^Uz - 2x^Uy^Uz^U
\end{array}
$$

Fractional terms $\frac{x}{y}$ are underestimated by replacing them with a new variable w_F and adding two new constraints to the problem:

$$
w_F \geq \begin{cases} \frac{x^L}{y} + \frac{x}{y^U} - \frac{x^L}{y^U} & \text{if } x^L \geq 0 \\ \frac{x}{y^U} - \frac{x^Ly}{y^Ly^U} + \frac{x^L}{y^L} & \text{if } x^L < 0 \end{cases}
\quad
w_F \geq \begin{cases} \frac{x^U}{y} + \frac{x}{y^L} - \frac{x^U}{y^L} & \text{if } x^U \geq 0 \\ \frac{x}{y^L} - \frac{x^Uy}{y^Ly^U} + \frac{x^U}{y^U} & \text{if } x^U < 0 \end{cases}
$$

Fractional trilinear terms $\frac{xy}{z}$ can be underestimated by replacing them by a new variable w_{FT} and adding the new constraints (for $x^L, y^L, z^L \geq 0$):

$$
\begin{array}{rcl}
w_{FT} & \geq & \frac{xy^L}{z^U} + \frac{x^Ly}{z^U} + \frac{x^Ly^L}{z} - 2\frac{x^Ly^L}{z^U} \\
w_{FT} & \geq & \frac{xy^L}{z^U} + \frac{x^Ly}{z^L} + \frac{x^Ly^U}{z} - \frac{x^Ly^U}{z^L} - \frac{x^Ly^L}{z^U} \\
w_{FT} & \geq & \frac{xy^U}{z^L} + \frac{x^Uy}{z^U} + \frac{x^Uy^L}{z} - \frac{x^Uy^L}{z^U} - \frac{x^Uy^U}{z^L} \\
w_{FT} & \geq & \frac{xy^U}{z^U} + \frac{x^Uy}{z^L} + \frac{x^Ly^U}{z} - \frac{x^Ly^U}{z^U} - \frac{x^Uy^U}{z^L} \\
w_{FT} & \geq & \frac{xy^L}{z^U} + \frac{x^Ly}{z^L} + \frac{x^Uy^L}{z} - \frac{x^Uy^L}{z^L} - \frac{x^Ly^L}{z^U} \\
w_{FT} & \geq & \frac{xy^U}{z^U} + \frac{x^Uy}{z^L} + \frac{x^Ly^U}{z} - \frac{x^Ly^U}{z^U} - \frac{x^Uy^U}{z^L} \\
w_{FT} & \geq & \frac{xy^L}{z^U} + \frac{x^Ly}{z^L} + \frac{x^Uy^L}{z} - \frac{x^Uy^L}{z^L} - \frac{x^Ly^L}{z^U} \\
w_{FT} & \geq & \frac{xy^U}{z^L} + \frac{x^Uy}{z^L} + \frac{x^Uy^U}{z} - 2\frac{x^Uy^U}{z^L}
\end{array}
$$

To relax a concave univariate function $f_{U(i)}(x)$ over $[x^L, x^U]$, the αBB algorithm uses a chord underestimator:

$$f(x^L) + \frac{f(x^U) - f(x^L)}{x^U - x^L}(x - x^L).$$

The main innovation of the αBB algorithm is in the underestimation of a general nonconvex function term $f_{N(i)}(x)$. This 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 α_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 α_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 $\Delta \equiv \text{Diag}_{i=1}^{n}(\alpha_i)$ is the matrix with α_i as diagonal entries and all zeroes elsewhere (diagonal shift matrix). Thus the main focus of the theoretical studies concerning all αBB variants is on the determination of the α_i parameters. Some methods are based on the simplifying requirement that the α_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 α 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 [39].

Thus, having constructed a convex underestimating function for the reformulated function $f(x)$, the relaxation of the problem is carried out accordingly, bearing in mind that:

- the objective function is replaced by its convex underestimator;
- any nonlinear equality constraint $h_i(x) = 0$ is replaced by the two inequality constraints $h_i(x) \leq 0$ and $-h_i(x) \leq 0$;
- any nonconvex inequality constraint $g_i(x) \leq 0$ is reformulated to $\underline{g}_i(x) \leq 0$ where $\underline{g}_i$ is the convex underestimator for $g_i(x)$.

2.3.2 Smith's convex relaxation

In the work of Smith [114, 116], Smith's standard form (see sections 2.1.9, 5.2.2.1) is used as a starting point for forming the convex relaxation. All the nonconvex terms are isolated in constraints of the forms $z = f(x)$ or $z = g(x, y)$, where f is a univariate nonlinear function, g is a bivariate nonlinear function and $x, y, z \in \mathbb{R}$ are single problem variables. The smallest convex set containing $F = \{(x, z) \mid z = f(x)\} \subseteq \mathbb{R}^2$ is given by $\bar{F} = \{(x, z) \mid z \geq \underline{f}(x) \wedge z \leq \bar{f}(x)\}$ where $\underline{f}, \bar{f}$ are respectively the convex and the concave envelopes of f. Similarly for g, the smallest convex set containing $G = \{(x, y, z) \mid z = g(x, y)\} \subseteq \mathbb{R}^3$ is given by $\bar{G} = \{(x, y, z) \mid z \geq \underline{g}(x, y) \wedge z \leq \bar{g}(x, y)\}$ where $\underline{g}, \bar{g}$ are respectively the convex and the concave envelopes of g. When the functions f are convex, concave univariate terms and the functions g are bilinear, trilinear or fractional terms, the convex envelopes are the same as those listed in Section 2.3.1. However, Smith's reformulation makes it possible to extend convex relaxations to nonlinear terms which are not twice-differentiable (as opposed to the αBB relaxation).

Smith, in his work, did not construct the convex/concave envelopes of a piecewise convex and concave term (like, for example, a monomial of odd power when the range of the variable includes zero); Chapter 4 addresses this problem.

The disadvantage of Smith's relaxation is that it is based on Smith's standard form, which is a lifting, and therefore adds new variables to the problem (one for each nonlinear term in the problem). This may result in an excessively high number of problem variables.

2.3.3 BARON's convex relaxation

BARON stands for "Branch And Reduce Optimization Navigator". It is a global optimization software written by Sahinidis and co-workers that relies on a Branch-and-Bound algorithm to solve factorable problems to global optimality [102]. The lower bound to the objective function in each region of the Branch-and-Bound tree is calculated by means of a convex relaxation [126]. The techniques used to form the nonlinear convex relaxation of factorable problems include all the standard convex envelopes for nonconvex factorable terms found in Section 2.3.1 (apart from the α-parameter underestimation for general twice-differentiable nonconvex terms, which is not part of BARON). The most innovative features of BARON in terms of convex relaxations are the following.

- Specific mention of piecewise convex and piecewise concave univariate terms (called *concavoconvex* by the authors) and the respective convex and concave envelopes (see also the detailed study in Chapter 4) [126]. 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 [102].

- Convex and concave envelopes are suggested for various types of fractional terms, based on the theory of convex extensions [127, 128]. 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:

$$\left.\begin{array}{c} z \geq \frac{x^L}{y_a}(1-\lambda) + \frac{x^U}{y_b}\lambda \\ y^L \leq y_a \leq y^U \\ y^L \leq y_b \leq y^U \\ y = (1-\lambda)y_a + \lambda y_b \\ x = x^L + (x^U - x^L)\lambda \\ 0 \leq \lambda \leq 1 \end{array}\right\} \tag{2.9}$$

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

$$\left.\begin{array}{c} z \geq \frac{x^L(y^L+y^U-y_a)}{y^L y^U}(1-\lambda) + \frac{x^U}{y_b}\lambda \\ y^L \leq y_a \leq y^U \\ y^L \leq y_b \leq y^U \\ y = (1-\lambda)y_a + \lambda y_b \\ x = x^L + (x^U - x^L)\lambda \\ 0 \leq \lambda \leq 1 \end{array}\right\} \tag{2.10}$$

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

- the bilinear envelope:

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

- the nonlinear envelope:

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

- Having constructed a reasonably tight nonlinear convex relaxation, the authors discuss various outer approximation techniques to reformulate this to a linear relaxation which can be solved by using a very fast and efficient LP software [126].

BARON is a very efficient global optimization software, paying particular attention to several important implementational aspects, such as: generating valid cuts during pre-processing (optimality bounds tightening [102, 126]) and during execution (feasibility bounds tightening and range reduction techniques [99, 100]); improving the branching scheme [102, 126, 130, 101]; and most importantly of all, targeting particular problem formulations with specialized solvers [126]. 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.

2.4 Linear relaxations

It is possible to use linear over- and underestimators for each nonlinear term in a nonconvex NLP in order to obtain a linear relaxation of the problem. Because a linear problem is always convex, the convexity properties that guarantee the validity of a lower bound remain true. The advantage of a linear relaxation with respect to a convex (possibly nonlinear) relaxation is that linear optimization software can be employed to solve the relaxed underestimating problem. Linear optimization codes are much more efficient than nonlinear optimization software; hence the overall run of a Branch-and-Bound algorithm might be faster. The disadvantage, however, is that a linear relaxation might not be a convex envelope: hence the convexity gap might be increased and the lower bound to the original problem might not be the best possible.

2.4.1 Reformulation-linearization technique

The basic idea of the Reformulation-Linearization Technique (RLT), proposed by Sherali and co-workers in a number of papers [109, 111, 107, 113, 112, 110, 108], consists in deriving valid cuts to the problem by multiplying together various factors involving variables and constraints. This technique was initially proposed in conjunction with combinatorial problems with binary variables, and then extended to continuous bilinear problems, polynomial problems and factorable problems.

In this section we shall describe the RLT applied to bilinear problems [111] of the form:

$$\left.\begin{array}{rcl} \min_x \; x^TQx & + & c^Tx \\ Ax & = & b \\ x^L \leq & x & \leq x^U, \end{array}\right\} \qquad (2.11)$$

where $c, x \in \mathbb{R}^n$, Q is an $n \times n$ matrix, A is an $m \times n$ matrix and $b \in \mathbb{R}^m$.

In order to form a linear (convex) relaxation of problem (2.11), the RLT applied to bilinear problems considers the following sets of algebraic expressions:

- the bound factor set $B_F = \{x_i - x_i^L \mid i \leq n\} \cup \{x_i^U - x_i \mid i \leq n\}$;
- the constraint factor set $C_F = \{\sum_{j=1}^n a_{ij}x_j - b_i \mid i \leq m\}$.

Note that for each $\beta \in B_F$ the constraint $\beta \geq 0$ is a valid problem constraint, and so is $\gamma = 0$ for all $\gamma \in C_F$.

The RLT procedure for forming the convex relaxation consists in creating new linear valid constraints (reformulation step) by multiplying together bound factors and constraint factors as follows:

1. for all $\beta_1, \beta_2 \in B_F$, $\beta_1\beta_2 \geq 0$ is a valid constraint (generation via bound factors);
2. for all $\beta \in B_F$ and for all $\gamma \in C_F$, $\beta\gamma = 0$ is a valid constraint (mixed generation);
3. for all $\gamma_1, \gamma_2 \in C_F$, $\gamma_1\gamma_2 = 0$ is a valid constraint (generation via constraint factors).

Having created all these new constraints, we define new variables $w^i_j = x_i x_j$ for all i, j between 1 and n, and use them to replace the corresponding bilinear products appearing in problem (2.11) or in the newly generated constraints (linearization step). Assuming there are t distinct bilinear terms[4], we end up with a linear relaxation whose variable vector (x, w) is in $\mathbb{R}^{n+t}$.

Let S_F be the region defined by the newly generated constraints. The linear relaxation of problem (2.11) is as follows:

$$\left.\begin{array}{rcl}\min_x \quad c^T x & + & p^T w \\ Ax & = & b \\ (x, w) & \in & S_F \\ x^L \leq & x & \leq x^U \\ w^L \leq & w & \leq w^U\end{array}\right\} \tag{2.12}$$

where $w^L, w^U \in \mathbb{R}^t$ are the variable bounds on the w variables (obtained by simple interval arithmetic on the bounds of the x variables via the defining relations $w^i_j = x_i x_j$).

It is also possible to derive RLT constraints from a system of inequalities $Hx \leq d$, where H is an $m' \times n$ matrix and $d \in \mathbb{R}^{m'}$. Define the inequality constraint factor set $C'_F = \{\sum_{j=1}^n h_{ij}x_j - d_i \mid i \leq m'\}$ and notice that for all $\gamma' \in C'_F$, we have $\gamma' \leq 0$. Since all the bound factors $\beta \in B_F$ are nonnegative, any constraint of the form $\beta\gamma' \leq 0$ is a valid problem constraint.

When the RLT is applied to general polynomial programming problems, arbitrary multilinear products:

$$\prod_j \beta_{i_j} \prod_j \gamma_{i_j}$$

may arise between the bound factors β_{i_j} and constraint factors γ_{i_j}.

[4]Note that $t \leq \frac{n(n+1)}{2}$.

The RLT gives rise to very tight convex relaxations but it has a disadvantage: it is a heuristic reformulation. If implemented to its full extent (i.e. adding all possible constraints deriving from all possible bound and constraint factor products), it gives rise to relaxations which have too many constraints and therefore take inordinate amounts of time to solve. More precisely, with $2n$ bound factors and m constraint factors, we obtain:

- $4n^2$ possible bound factor products;
- m^2 possible constraint factor products;
- $2mn$ possible mixed factor products,

for a total of $4n^2 + 2mn + m^2$ new constraints. When the RLT is applied to polynomial problems, more added constraints can be obtained recursively from the newly formed factor product constraints and the original factors, so this number increases even more.

2.5 Other reformulations

There are several other reformulation studies in the literature. These are not analysed in detail here either because they are specific to a certain type of optimization problem (and, hence, difficult to generalize) or because they address software implementation issues. Here follows a short list of some of the most notable such works.

A qualitative knowledge-based software implementation, REFORM, for the generic reformulation of MINLPs, is proposed in [12]; it is aimed at increasing the general robustness of an optimization problem before submitting it to an optimization software. Frangioni [41] showed how to reformulate a bilevel programming problem (a hierarchical two-stage optimization problem) as a MILP. Fischer [36] reformulated the complementarity problem as a minimization problem with nonnegativity constraints. Some regularity conditions ensure that a stationary point of the reformulation is a solution of the original problem. Bomze and co-workers [20, 21] reformulated the maximum clique problem as a quadratic problem over a standard simplex, and used copositivity-based procedures to solve the standardized quadratic problem. In [19], a general method for handling disjunctive constraints in MINLPs is proposed, which relies on reformulating disjunctive constraints to a mathematical form that can be solved using standard MILP software. Sahinidis and co-workers [1, 129] have studied various reformulations of the pooling problem extensively.

2.6 Conclusion

This chapter has provided a literature review of various techniques for the reformulation of optimization problems, with emphasis on the most useful and generally applicable reformulation techniques for deterministic global optimization. This includes exact reformulations leading to standard forms (box-constrained problems, separable problems, binary problems, concave problems, d.c. problems, factorable problems, generic nonlinear problems), exact reformulations which address a specific feature of a problem, and relaxations (convex and linear).

It is clear from the review presented in this chapter that much progress has been achieved in both exact reformulation and convex relaxation of nonconvex NLPs, especially over the past decade. Equally clearly, there are several areas for which no satisfactory solutions exist to date. Two of these will receive special attention in this thesis; these are respectively:

1. *The tight linear relaxation of bilinear terms.* Bilinear terms are almost ubiquitous in practical applications. The RLT technique described in Section 2.4.1 represents a good attempt to tighten the convex relaxation of NLPs involving such terms. However, the large numbers of constraints and variables that it introduces represents a significant obstacle to its use in large problems. An algorithm which can be applied to large sparse NLPs and which leads to convex relaxations that are both tighter and smaller is presented in Chapter 3.

2. *The tight convex relaxation of monomials of odd power.* Although such terms appear frequently in applications (e.g. as cubics or quintics), no convex envelope has been proposed for them to date. As a result, they are currently treated either as general non-convexities (cf. Section 2.3.1), or via specialized branching schemes (cf. Section 2.3.3). Neither of these two approaches is satisfactory. A convex envelope that addresses this issue in a more efficient manner is presented in Chapter 4 of this thesis.

Chapter 3

Reduction constraints for sparse bilinear programs

We consider the solution of Nonlinear Programs (NLPs) of the following standard form:

$$
[P]: \quad
\begin{array}{rcll}
\min\limits_{z} & z_l & & (3.1)\\
Az & = & b & (3.2)\\
z_i = z_j z_k & & \forall i,j,k \in B & (3.3)\\
z_i = \dfrac{z_j}{z_k} & & \forall i,j,k \in F & (3.4)\\
z_i = f_i(z_j) & & \forall i,j \in N & (3.5)\\
z^L \leq & z & \leq z^U & (3.6)
\end{array}
$$

where $z = (z_1, \ldots, z_p) \in \mathbb{R}^p$ are the problem variables, l is an index in the set $\{1, \ldots, p\}$, $A = (a_{ij})$ is an $m \times p$ matrix of rank m, $b \in \mathbb{R}^m$, B, F are sets of index triplets $\{(i,j,k) \mid 1 \leq i,j,k \leq p\}$, N is a set of index pairs $\{(i,j) \mid 1 \leq i,j \leq p\}$, $f_i : \mathbb{R} \to \mathbb{R}$ are nonlinear univariate functions and $z^L, z^U \in \mathbb{R}^p$ are variable bounds. The above standard form is practically important as it can be shown that all NLPs can automatically be reformulated to it using symbolic manipulations [116, 114] (see Chapter 5). Therefore, any theoretical results, manipulations and solution algorithms derived on the basis of this standard form are generally applicable.

Spatial Branch-and-Bound (sBB) algorithms [133] are among the most effective methods currently available for the global solution of nonconvex NLPs (see Chapter 1). An important requirement for any sBB algorithm is to be able to construct a tight convex underestimator of the NLP within any given region of the space of the variables. For the standard form $[P]$, the lower bound to the objective function can be generated by replacing the nonconvex constraints (3.3), (3.4), (3.5) with their convex relaxations; in this manner, a convex relaxation of the whole problem can be obtained in a simple and completely automatic fashion.

Tight convex relaxations for the most common nonconvex terms are available in the literature. One of the best known is that proposed by McCormick (1976) for the relaxation of bilinear terms. Linear fractional terms like those appearing in constraint (3.4) are reformulated to $z_i z_k = z_j$ and replaced by the McCormick convex relaxation. For the nonlinear terms in constraint (3.5), the convex relaxation depends on the function f_i. When f_i is wholly concave or wholly convex, the function itself and its secant provide the tightest convex relaxation. For functions f_i which are partially concave and partially convex, like e.g. $z_i = z_j^{2k+1}$, where $k \in \mathbb{N}$, building the convex relaxation may not be so straightforward (see, for example, [72]).

3.0.1 Example

Consider the problem

$$\begin{array}{rcl} \displaystyle\min_{x,y,z,w} & z & \\ z - x + w & = & 1 \\ w & = & xy \\ (x^L, y^L, z^L, w^L) & \leq & (x,y,z,w) \leq (x^U, y^U, z^U, w^U) \end{array}$$

in standard form. To obtain the convex relaxation, we replace the nonconvex constraint $w = xy$ *with its McCormick convex relaxation:*

$$\left.\begin{array}{rcl} w & \geq & x^L y + y^L x - x^L y^L \\ w & \geq & x^U y + y^U x - x^U y^U \\ w & \leq & x^U y + y^L x - x^U y^L \\ w & \leq & x^L y + y^U x - x^L y^U \end{array}\right\} . \qquad (3.7)$$

This chapter presents a technique to automatically reformulate optimization problems in standard form $[P]$ in such a way that some of the nonlinear constraints (3.3) are replaced by linear constraints. This is possible because, in certain instances, feasible regions described by nonlinear constraints are, in fact, (linear) hyperplanes. We propose an automatic algorithm to identify such instances in large scale systems. The solution of the reformulated problems by sBB algorithms often requires reduced computational effort, sometimes even by several orders of magnitude. This occurs because replacing nonlinear constraints with linear ones makes the feasible region of the convex relaxation is much tighter.

The creation of new linear constraints via multiplication of existing linear constraints by problem variables was proposed in [111, 108] under the name “reformulation-linearization technique” (RLT). The RLT uses linear constraints built in this way to provide a lower bound to bilinear programming problems. The maximum possible number of new linear constraints is created by multiplying all linear constraints by all problem variables; thus, the method may lead to excessive computational complexity in large problems. In contrast, the algorithm presented in this chapter identifies precisely the set of multiplications of linear constraints by variables that are beneficial.

The rest of this chapter is organized as follows. In Section 3.1, we introduce the basic concepts and ideas behind reduction constraints. Section 3.2 considers the existence of reduction constraints in a more formal manner. This provides the basis of the fast algorithm for the identification of reduction constraints presented in Section 3.3. An example of the application of the algorithm is presented in detail in Section 3.4. The effects of reduction constraints on the global solution of an important class of problems with bilinear constraints, namely pooling and blending problems, are considered in Section 3.5. Finally, Section 3.6 proposes an extension of the algorithm of Section 3.3 that may result in better reformulations at the expense of higher computational time and memory.

3.1 Basic concepts

This section introduces in an informal manner the motivation and general ideas behind the work presented in this chapter. Let y be a single problem variable, and let $w = (w_1, \ldots, w_n)$ and $x = (x_1, \ldots, x_n)$ be problem variables (with $n < p$) such that the following constraints exist in the problem:

$$\forall j \leq n \; (w_j = x_j y). \tag{3.8}$$

Now suppose that the problem also involves the linear constraint (cf. equation (3.2)):

$$\sum_{j=1}^{n} a_{ij} x_j \quad = \quad b_i \tag{3.9}$$

Multiplying this constraint by y and making use of (3.8) leads to a new linear constraint:

$$\sum_{j=1}^{n} a_{ij} w_j - b_i y \quad = \quad 0. \tag{3.10}$$

The linear constraint (3.10) is redundant with respect to the original constraints. Indeed, it can be used to replace one of the bilinear constraints $w_j = x_j y$ in problem $[P]$ without affecting the feasible region of the problem. To see this, assume $a_{ik} \neq 0$ for some $k \in \{1, \ldots, n\}$ and discard the nonlinear constraint $w_k = x_k y$ from the set (3.8) above. We now replace b_i in (3.10) with the left hand side of (3.9) to obtain:

$$\sum_{j=1}^{n} a_{ij} w_j - \sum_{j=1}^{n} a_{ij} x_j y = \sum_{j=1}^{n} a_{ij}(w_j - x_j y) = 0.$$

Since $w_j = x_j y$ for all $j \neq k$, the above reduces to $a_{ik}(w_k - x_k y) = 0$, which implies $w_k = x_k y$. We have thus recovered the discarded bilinear constraint from the other $n-1$ bilinear constraints and the new linear constraint (3.10). We call the latter a *reduction constraint* as it reduces the number of bilinear terms in the original problem.

The geometric significance of reduction constraints is that, given any $i \leq m$, the set:

$$\{(w, x, y) \mid \forall j \leq n \ (w_j = x_j y) \wedge \sum_{j=1}^{n} a_{ij} x_j = b_i\} \tag{3.11}$$

is less nonlinear than might appear to be the case: in fact, provided that $a_{ik} \neq 0$ for some $k \leq n$, it is equal to the set:

$$\{(w, x, y) \mid \forall j \neq k \ (w_j = x_j y) \wedge \sum_{j=1}^{n} a_{ij} x_j = b_i \wedge \sum_{j=1}^{n} a_{ij} w_j - b_i y = 0\}, \tag{3.12}$$

where the kth bilinear constraint has been replaced by a linear constraint. Consequently, the convex relaxation of the set (3.12) is tighter than that of set (3.11), which may be important in the context of sBB algorithms.

Below, we consider an extreme example of such a reformulation.

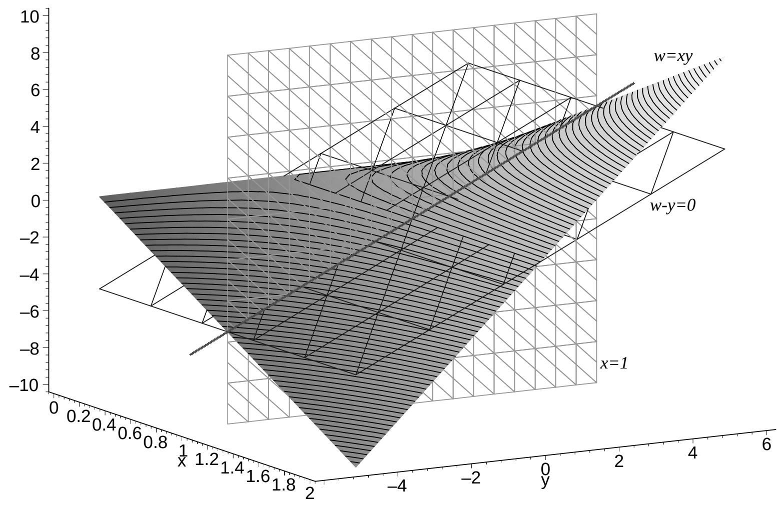


Figure 3.1: Linearity embedded in a bilinear constraint.

3.1.1 Example

Let $(w, x, y) \in \mathbb{R}^3$ and consider the set:

$$A_1 = \{(w, x, y) \mid w = xy \wedge x = 1\}.$$

This set is defined as the intersection of the bilinear surface xy and the vertical plane $x = 1$ (see Fig. 3.1). By multiplying $x = 1$ by y, we get the linear constraint $w - y = 0$ (the skew plane in Fig. 3.1). It is evident that the set

$$A_2 = \{(w, x, y) \mid x = 1 \wedge w - y = 0\}$$

is the same as the set A_1. However, A_2 is defined only by linear relationships whereas A_1 is not. The convexification of A_1 in the context of an sBB algorithm (e.g. by introducing the McCormick relaxation of xy) would be both unnecessary and counter-productive.

A limitation of the reformulation technique presented above is that all constraints of the type $w_j = x_j y$ have to exist in the problem (3.1) before the linear constraint (3.10) can be created. Consider, however, a problem that already includes bilinear constraints $w_j = x_j y$ for all $j \leq n-1$, and two linear constraints:

$$\sum_{j=1}^{n-1} a_{ij}x_j + a_{in}x_n = b_i \tag{3.13}$$

$$\sum_{j=1}^{n-1} a_{hj}x_j + a_{hn}x_n = b_h \tag{3.14}$$

where $i, h \leq m$ and a_{in}, a_{hn} are nonzero constants. On multiplying (3.13) and (3.14) by y, we obtain the two linear constraints:

$$\sum_{j=1}^{n-1} a_{ij}w_j + a_{in}w_n - b_iy = 0 \tag{3.15}$$

$$\sum_{j=1}^{n-1} a_{hj}w_j + a_{hn}w_n - b_hy = 0. \tag{3.16}$$

where w_n is a new variable defined via a new bilinear constraint $w_n = x_ny$. By forming a suitable linear combination of the two constraints, the new variable w_n can be eliminated, and we now have a linear constraint similar to (3.10) which can be used to eliminate one of the original bilinear constraints in the manner indicated earlier.

3.2 Fundamental properties

Section 3.1 has shown that it is possible, in some cases, to replace nonconvex nonlinear constraints by linear ones. We now proceed to provide a more rigorous foundation for these ideas.

Consider an optimization problem in standard form $[P]$ and subsets of problem variables $w, x \in \mathbb{R}^n$ and $y \in \mathbb{R}$. Suppose the problem includes the set of m linear equality constraints $Ax = b$, where the matrix A is of full row rank m.

Multiplying $Ax = b$ by y, we create m reduction constraints of the form $Aw - yb = 0$, where the variable vector $w \in \mathbb{R}^n$ is defined as in equation (3.8). The following theorem shows that the reduction constraints can be used to replace m of the n bilinear constraints in (3.8) without changing the feasible region of the problem.

3.2.1 Theorem

Let $J \subseteq \{1, \ldots, n\}$ be an index set of cardinality $|J| = n - m$ and consider the sets:

$$
\begin{aligned}
C &= \{(x, w, y) \mid Ax = b \wedge \forall j \le n \ (w_j = x_j y)\} && (3.17)\\
R_J &= \{(x, w, y) \mid Ax = b \wedge Aw - yb = 0 \wedge \forall j \in J \ (w_j = x_j y)\} && (3.18)
\end{aligned}
$$

Then, there is at least one set J such that $C = R_J$.

Proof. The fact that $C \subseteq R_J$ for any set J is straightforward: if $(x, w, y) \in C$, then it also satisfies the constraints $Aw - yb = 0$, hence $(x, w, y) \in R_J$.

We shall now prove the converse inclusion. Since $Ax = b$, the reduction constraint system $Aw - yb = 0$ implies $Aw - yAx = 0$. If we now define $u_j = w_j - x_j y, \forall j \le n$, this can be written as the linear homogeneous system $Au = 0$. Since rank$(A) = m$, there exists a permutation π of the columns of A such that:

$$
(A_0 \ A_1) \begin{pmatrix} u^{(0)} \\ u^{(1)} \end{pmatrix} = 0,
$$

where A_0 is a non-singular $m \times m$ matrix, and $(u^{(0)^T}, u^{(1)^T})^T$ is the corresponding permutation of u with $u^{(0)} \in \mathbb{R}^m$ and $u^{(1)} \in \mathbb{R}^{n-m}$. Let J be the image of $\{n - m + 1, \ldots, n\}$ under the permutation π, and let $(x, w, y) \in R_J$. Since (x, w, y) satisfies $w_j = x_j y$ for all $j \in J$, then $u^{(1)} = 0$; since (x, w, y) also satisfies the reduction constraint system, we have $Au = 0$, which implies $u^{(0)} = 0$ as well. Hence $w_j = x_j y$ for all $j \le n$. □

The above theorem concerns the multiplication of a set of linear constraints by a single variable y. The latter could, in fact, be any one of the system variables x, so considering multiplications by variable x_k, we can modify the definitions (3.17) and (3.18) to:

$$
\begin{aligned}
C^k &= \{(x, w^k) \mid Ax = b \wedge \forall j \le n \ (w^k_j = x_j x_k)\}, \quad k \le n && (3.19)\\
R^k_J &= \{(x, w^k) \mid Ax = b \wedge Aw^k - x_k b = 0 \wedge \forall j \in J \ (w^k_j = x_j x_k)\}, \quad k \le n && (3.20)
\end{aligned}
$$

where $w^k = (w^k_1, \ldots, w^k_n)$. Now, by virtue of the theorem, we will have $C^k = R^k_J, \forall k \le n$. We note that the index set J appearing in (3.20) is the same for all k: the proof of the theorem indicates that J is fully determined by the nature of the matrix A and does not depend on the specific variable y being used to multiply the constraints. Consequently, the union of the sets C^k over all $k \le n$ must be equal to the corresponding union of sets R^k_J. This proves that the two sets:

$$\begin{aligned} \bar{C} &= \{(x, w) \mid Ax = b \wedge \forall k \leq n \;\; j \leq n \; (w_j^k = x_j x_k)\} && (3.21) \\ \bar{R}_J &= \{(x, w) \mid Ax = b \wedge Aw - x_k b = 0 \wedge \forall k \leq n, \;\; \forall j \in J \; (w_j^k = x_j x_k)\} && (3.22) \end{aligned}$$

are equal to each other.

The geometrical implications of the above results are that the intersection in $\mathbb{R}^n$ of a set of bilinear terms, and the linear form $Ax = b$ is a hypersurface containing a degree of linearity. By exploiting this linearity, we are able to replace some of the bilinear terms with linear constraints.

3.3 An algorithm for the identification of valid reduction constraints

Section 3.2 has established that, in principle, reduction constraints exist for any NLP involving bilinear terms and linear constraints. Here we are concerned with two more practical questions. Section 3.3.1 introduces a criterion for determining whether the introduction of reduction constraints is likely to be beneficial. Then sections 3.3.2 and 3.3.3 present an efficient algorithm for the identification of such reduction constraints based on the above criterion; this is particularly important in the case of large NLPs for which neither identification by inspection nor indiscriminate multiplication of linear constraints by problem variables are practical propositions.

3.3.1 Valid reduction constraint sets

Given an NLP in the standard form [P], consider a subset $\mathcal{L}$ of its linear constraints (3.2). Let the set of variables that occur in these constraints be denoted by $\mathcal{V}(\mathcal{L})$, i.e. $\mathcal{V}(\mathcal{L}) \equiv \{k \mid \exists l \in \mathcal{L} \; (a_{lk} \neq 0)\}$.

Now consider multiplying the linear constraint set $\mathcal{L}$ by a problem variable z_l. This will create bilinear terms of the form $z_l z_k, \forall k \in \mathcal{V}(\mathcal{L})$. Some of these terms will already occur in the NLP, i.e. $\exists i : (i, l, k) \in B$. Let $K(l, \mathcal{L})$ be the subset of $\mathcal{V}(\mathcal{L})$ that leads to *new* bilinear terms, i.e.:

$$K(l, \mathcal{L}) \equiv \{k \mid k \in \mathcal{V}(\mathcal{L}) \wedge \nexists i : (i, l, k) \in B\}$$

The theorem of Section 3.2 indicates that we can now construct a problem that has the same feasible region as $[P]$ by replacing $|\mathcal{L}|$ bilinear constraints by linear constraints. The latter are said to form a *valid reduction constraint system* if the substitution leads to a reduction in the overall number of bilinear constraints in comparison with the original problem $[P]$, i.e. if the number of new bilinear constraints introduced by the multiplication by z_l is smaller than the number of bilinear constraints eliminated by the reduction constraints:

$$|K(l,\mathcal{L})| \quad < \quad |\mathcal{L}| \tag{3.23}$$

Despite the apparent simplicity of this criterion, applying it in a brute force manner as a means of identifying reduction constraint systems is impractical: for an NLP containing M linear constraints, one would have to examine $2^M - 1$ subsets $\mathcal{L}$ for each candidate multiplier variable z_l.

3.3.2 A graph-theoretical representation of linear constraints

Here we consider a fast graph-theoretical algorithm for the identification of linear constraint subsets $\mathcal{L}$ that satisfy (3.23) for a given multiplier variable z_l. We start by constructing a bipartite graph [64, 45, 51] $\mathcal{G}_l$ where the set of nodes is partitioned into two disjoint subsets $\mathcal{N}^L$ and $\mathcal{N}_l^V$. We call these the “constraint” and “variable” nodes respectively. The former correspond to the set of linear constraints in $[P]$ while the latter correspond to those problem variables which do not appear in any bilinear term (3.3) multiplied by z_l, i.e.:

$$\mathcal{N}_l^V \equiv \{k | \nexists i : (i,l,k) \in B\}$$

The set of edges $\mathcal{E}_l$ in graph $\mathcal{G}_l$ is defined as:

$$\mathcal{E}_l \equiv \{(\lambda,k) | \lambda \in \mathcal{N}^L \wedge k \in \mathcal{N}_l^V \wedge a_{\lambda k} \neq 0\}$$

Thus, edge (λ, k) exists if variable z_k occurs in linear constraint λ.

Suppose, now, that $\mathcal{L}$ is a subset of the nodes $\mathcal{N}^L$. Let $\bar{K}(l,\mathcal{L})$ denote the subset of nodes $\mathcal{N}_l^V$ that are connected to nodes in $\mathcal{L}$, i.e.:

$$\bar{K}(l,\mathcal{L}) \equiv \{k \mid k \in \mathcal{N}_l^V \wedge \exists \lambda \in \mathcal{L} : (\lambda,k) \in \mathcal{E}_l\}$$

It can be verified that, in view of the definitions of sets $\mathcal{V}$, $\mathcal{N}_l^V$ and $\mathcal{E}_l$, the two sets $K(l,\mathcal{L})$ and $\bar{K}(l,\mathcal{L})$ are, in fact, identical. Consequently, a valid reduction constraint set (i.e. one that satisfies criterion (3.23)) will correspond to a *dilation* in graph $\mathcal{G}_l$, i.e. a subset $\mathcal{L}$ of, say, m nodes $\mathcal{N}^L$ that are connected to fewer than m distinct nodes $\mathcal{N}_l^V$. This provides a practical basis for the identification of valid reduction constraint sets using efficient existing algorithms for the determination of dilations in bipartite graphs.

3.3.3 Efficient identification of dilations

Dilations in bipartite graphs are closely related to the existence of *output set assignments* (OSA) in such graphs [87]. A subset $\mathcal{A}_l$ of the edges $\mathcal{E}_l$ in graph $\mathcal{G}_l$ is an OSA if no node is incident to more than one edge in $\mathcal{A}_l$. A *complete* OSA is one in which each and every node $\lambda \in \mathcal{N}^L$ is incident to exactly one edge in it; in this case, $|\mathcal{A}_l| = |\mathcal{N}^L| = M$.

Given a non-complete OSA $\mathcal{A}_l$ of cardinality $m < M$, it may be possible to obtain one of cardinality $m + 1$ by identifying an unassigned node $\lambda \in \mathcal{N}^L$ (i.e. one that is not incident to any edge in $\mathcal{A}_l$) and tracing an *augmenting path* emanating from this node [29]. An augmenting path is a sequence of $2\alpha - 1$ edges of the form:

$$(\lambda_1, k_1), (\lambda_2, k_1), (\lambda_2, k_2), \ldots, (\lambda_\alpha, k_{\alpha-1}), (\lambda_\alpha, k_\alpha)$$

such that $(\lambda_\beta, k_\beta) \notin \mathcal{A}_l, \forall \beta = 1, .., \alpha$ and $(\lambda_\beta, k_{\beta-1}) \in \mathcal{A}_l, \forall \beta = 2, .., \alpha$. If such a path can be found, then an OSA of cardinality $m+1$ can be obtained from $\mathcal{A}_l$ simply by replacing the $\alpha - 1$ edges $(\lambda_\beta, k_{\beta-1}), \beta = 2, .., \alpha$ in it by the α edges $(\lambda_\beta, k_\beta), \beta = 1, .., \alpha$.

Since the empty set is a valid (albeit trivial) OSA, the repeated tracing of augmenting paths can be used for the construction of OSAs of maximal cardinality.

A complete output set assignment for the graph $\mathcal{G}_l$ exists if and only if the graph does not contain any dilations. Consequently, dilations can be identified by the application of efficient algorithms designed for the identification of output set assignments such as that proposed by [29]. A basic algorithm [29] for constructing an augmenting path emanating from a node λ of graph $\mathcal{G}_l$ is shown in pseudo-code notation in Figure 3.2. The procedure `AugmentPath` returns a boolean flag $PathFound$ indicating whether or not an augmenting path has been found. $PathFound$ is initially set to false.

The variable $Assigned[k]$ contains the index of the constraint node to which variable node k is currently assigned; a value of 0 indicates a variable node that has not yet been assigned. The boolean variable $ConstraintVisited[\lambda]$ takes a value of true if constraint λ has been visited as part of the attempted construction of the augmenting tree. Similarly, $VariableVisited[k]$ marks whether variable k has been visited. Initially, all elements of the array $Assigned$ are set to 0, and all elements of $ConstraintVisited$ and $VariableVisited$ to false.

Line 2 of the algorithm simply marks the current constraint node λ as “visited”. Then lines 3-7 aim to determine whether node λ is adjacent to any node k that is not yet assigned. If so, an augmenting path has been found (line 4); k is immediately assigned to λ (line 5), and the algorithm terminates.

If no immediate assignment can be found for constraint node λ in lines 3-7, then this means that all variable nodes k adjacent to λ are already assigned to other constraint nodes. Lines 8-14 consider all such nodes k provided they have not already been visited as part of the current search (i.e. $VariableVisited[k]$ is still false, cf. line 8). If such a variable node k exists, then it is immediately marked as “visited” (line 9). Node k must be already assigned to a node $Assigned[k]$ which is different to λ. At line 10, the algorithm attempts to determine an augmenting path starting from this other node via a recursive invocation to procedure `AugmentPath`. If such a path is found (line 11), then node k is re-assigned to λ (line 12), and no further search is necessary (line 13).

If neither of the searches in lines 3-7 and 8-15 manage to find an augmenting path, then the procedure `AugmentPath` terminates at line 16, having left the flag $PathFound$ at its original value of false.

On termination of the algorithm, if $PathFound$ is true, then the above recursive algorithm must have first executed an assignment of a yet-unassigned variable node k at line 5 at the deepest level of the recursion; this will have been followed by exactly one re-assignment (line 12) of an already assigned variable node k at each higher level of the recursion. This will have increased the total number of assignments by one, which is, of course, the intention of tracing an augmenting path.

On the other hand, if, on termination of the algorithm, $PathFound$ is false, then it can be verified that (a) the number of constraint nodes marked as “visited” will exceed the number of variable nodes by exactly one, and (b) the algorithm will have visited each and every variable node adjacent to each visited constraint node. Consequently, the sets of constraint and variable nodes that have been visited form a dilation.

We are now in a position to formulate an algorithm that identifies the set of linear constraints $\mathcal{L}_l$ that lead to valid reduction constraints when multiplied by a given variable z_l. The algorithm is shown, again in pseudo-code notation, in Figure 3.3. The initialization part (lines 2-6) sets the set $\mathcal{L}_l$ to empty, constructs the bipartite graph $\mathcal{G}_l$, and declares all variable nodes in it

```
PROCEDURE AugmentPath($\lambda$, $PathFound$)

Set $ConstraintVisited[\lambda]$ := TRUE

IF $\exists$ a node $k : (\lambda, k) \in \mathcal{E}_l$ AND $Assigned[k] = 0$ THEN
    Set $PathFound$ := TRUE
    Set $Assigned[k]$ := $\lambda$
    RETURN
END IF

FOR every $k : (\lambda, k) \in \mathcal{E}_l$ AND NOT $VariableVisited[k]$ DO
    Set $VariableVisited[k]$ := TRUE
    AugmentPath($Assigned[k]$, $PathFound$)
    IF $PathFound$ THEN
        Set $Assigned[k]$ := $\lambda$
        RETURN
    END IF
END FOR

END AugmentPath
```

Figure 3.2: Algorithm for the construction of an augmenting path emanating from a linear constraint node λ in graph $\mathcal{G}_l$.

```
PROCEDURE ValidReductionConstraints($l$, $\mathcal{L}_l$)

Set $\mathcal{L}_l$ := $\emptyset$
Construct bipartite graph $\mathcal{G}_l$ as described in 3.3.2
FOR each variable node $k$ DO
    Set $Assigned$[$k$] := $0$
END FOR

FOR each constraint node $\lambda$ DO
    FOR each constraint node $\lambda'$ DO
        Set $ConstraintVisited[\lambda']$ := FALSE
    END FOR
    FOR each variable node $k$ DO
        Set $VariableVisited[k]$ := FALSE
    END FOR
    Set $PathFound$ := FALSE

    AugmentPath($\lambda$, $PathFound$)

    IF NOT $PathFound$ THEN
        Set $\mathcal{L}_l$ := $\mathcal{L}_l \cup \{\lambda' | ConstraintVisited[\lambda'] = \text{TRUE}\}$
    END IF
END FOR

END ValidReductionConstraints
```

Figure 3.3: Algorithm for identification of set of valid reduction constraint set $\mathcal{L}$ for variable z_l.

as unassigned. The algorithm then considers each constraint node λ in sequence (lines 7-19), attempting to construct an augmenting path emanating from it. Lines 8-14 perform the initializations necessary for the correct operation of procedure `AugmentPath` invoked on line 15 (cf. Figure 3.2). If an augmenting path is not found, then all constraints visited during the search for one are added to the set of constraints $\mathcal{L}_l$ (lines 16-18).

Procedure `AugmentPath` considers each edge in the bipartite graph at most once. Since procedure `ValidReductionConstraints` invokes `AugmentPath` once for each constraint node, the theoretical complexity of the algorithm is at most of the order (number of constraint nodes $\times$ number of edges). However, as observed by Duff (1981) , in practice, the computational complexity of such OSA algorithms is usually nearer to (number of constraint nodes + number of edges). This results in an efficient algorithm that is practically applicable to large problems.

3.4 A detailed example

In order to illustrate the operation of the algorithm proposed in Section 3.3, we consider the following bilinear optimization problem in standard form:

$$\left.\begin{array}{ll} \min_z & z_{24} \\ & z_{24} = c^T z \\ & Az' = b \\ & z_i = z_j z_k \ \ \forall (i,j,k) \in B \\ & 0 \le z \le 10, \end{array}\right\} \tag{3.24}$$

where:

$$c^T = (0,0,0,0,0,0,1,1,3,1,2,2,1,2,1,-1,2,-1,4,3,6,9,1),$$

$z = (z_1, \dots, z_{23})^T$, $z' = (z_1, \dots, z_6)^T$, $b = (1, 2, -1, 1)^T$,

$$A = \begin{pmatrix} 1 & & 2 & & 1 & 1 \\ 2 & -1 & & 1 & & 3 \\ & 1 & & 6 & 2 & -3 \\ 2 & & & 1 & 3 & \end{pmatrix}$$

and:

$$\begin{aligned} B = \{&(7,1,1),(8,2,2),(9,4,4),(10,5,5),(11,6,6),(12,1,4), \\ &(13,1,5),(14,1,6),(15,2,4),(16,2,5),(17,2,6),(18,3,4), \\ &(19,3,5),(20,3,6),(21,4,5),(22,4,6),(23,5,6)\,\} \end{aligned} \tag{3.25}$$

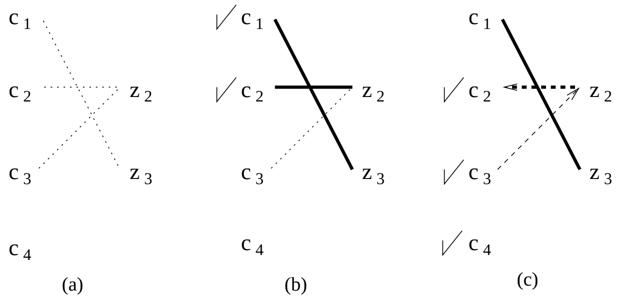


Figure 3.4: Generation of valid reduction constraints for multiplier variable z_1.

The above is, effectively, a bilinear programming problem involving 6 variables $z_1, \ldots, z_6$ which are, in turn, related via 4 linear constraints. The problem involves all bilinear combinations (including simple squares) of these 6 variables apart from $z_1 z_2$, $z_1 z_3$, $z_2 z_3$, z_3^2. New variables $z_7, \ldots, z_{23}$ have been introduced to represent these bilinear terms, and these are linearly combined to form the objective function represented by variable z_{24}.

In this problem, it is quite easy to see that only variables $z_1, \ldots, z_6$ can possibly give rise to reduction constraints when multiplied by the linear constraints $Az' = b$. We consider each of these 6 variables in turn, applying to it procedure `ValidReductionConstraints` of Figure 3.3.

3.4.1 Valid reduction constraints by multiplication with variable z_1

The bipartite graph $\mathcal{G}_1$ constructed as described in Section 3.3.2 for variable z_1 is shown in Figure 3.4a. Nodes $c_1, \ldots, c_4$ correspond to the four linear constraints. Edges correspond to the creation of new bilinear terms if a particular constraint were to be multiplied by z_1. For example, edges (c_1, z_1) and (c_1, z_3) exist because, if constraint c_1 were to be multiplied by z_1, then this would give rise to new terms z_1^2 and $z_1 z_3$ respectively.

As there are initially no assignments of variable nodes to constraint nodes, all edges are shown as dotted lines in Figure 3.4a. We now consider each constraint node in turn (lines 7-19 of Figure 3.3), attempting to construct an augmenting path emanating from it (line 15).

Starting from constraint node c_1, an augmenting path is found immediately to its adjacent variable node z_3. This is assigned to c_1 (cf. line 5 of Figure 3.2); this assignment is indicated as a solid line in Figure 3.4b.

Considering constraint c_2 also results in an augmenting path being found immediately, this time assigning z_2 to c_2. The resulting output set assignment is also shown as a solid line in Figure 3.4b.

So far, we have not identified any dilations. However, if we now attempt to construct an augmenting path emanating from constraint node c_3, we note that c_3 is adjacent to only one variable node, namely z_2 which is already assigned to node c_2. Consequently, the first search loop (lines 3-7) of the `AugmentPath` procedure fails to detect an augmenting path, and we therefore have to enter the second loop (lines 8-15). We mark z_2 as visited (line 9), and then recursively invoke `AugmentPath` (line 10) asking it to search for an augmenting path emanating from the constraint node currently assigned to z_2, namely c_2. However, c_2 is not adjacent to any variables other than z_2 which has already been visited; consequently, the recursive invocation of `AugmentPath` returns $PathFound$ false. As there are no more variable nodes adjacent to c_3, the first invocation of `AugmentPath` also returns $PathFound$ false. Consequently, line 17 of Figure 3.3 adds the two constraint nodes visited during this unsuccessful search for an augmenting path, i.e. c_2 and c_3 to the set of linear constraints $\mathcal{L}_1$ to be multiplied by z_1.

Finally, constraint node c_4 is completely isolated in $\mathcal{G}_1$, i.e. there are no edges incident to it. Thus, the corresponding invocation of procedure `AugmentPath` at line 15 of Figure 3.3 returns immediately with $PathFound$ false. Consequently, c_4 is also added to set $\mathcal{L}_1$.

As there are no more constraint nodes to be considered, we conclude that $\mathcal{L}_1 = \{c_2, c_3, c_4\}$. Essentially, the algorithm has identified automatically that multiplying constraints c_2:

$$2z_1 - z_2 + z_4 + 3z_6 = 2$$

and c_3:

$$z_2 + 6z_4 + 2z_5 - 3z_6 = -1$$

by z_1 creates only one new bilinear term, namely $z_1 z_2$, beyond those that already exist in the problem. Consequently, if we define a new variable $z_{25} \equiv z_1 z_2$, we end up with two linear constraints:

$$2z_7 - z_{25} + z_{12} + 3z_{14} = 2z_1$$

and

$$z_{25} + 6z_{12} + 2z_{13} - 3z_{14} = -z_1$$

respectively that can be used to eliminate two bilinear terms from the problem.

Moreover, the algorithm has also detected that multiplying constraint c_4:

$$2z_1 + z_4 + 3z_5 = 1$$

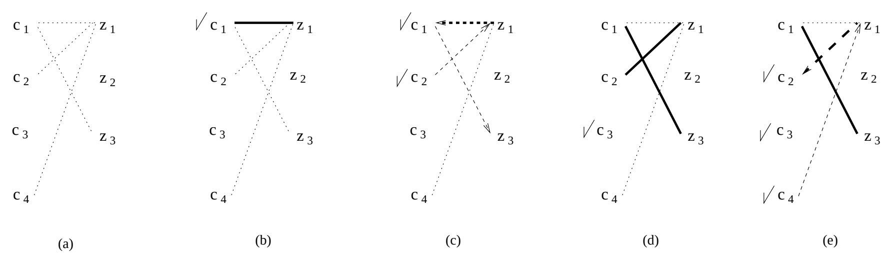


Figure 3.5: Generation of valid reduction constraints for multiplier variable z_2.

by z_1 does not generate any new bilinear terms. The new linear constraint:

$$2z_7 + z_{12} + 3z_{13} = z_1$$

obtained by this multiplication can be used to eliminate one more of the problem's bilinear terms. Overall, the number of bilinear terms in the problem can be reduced by two via the use of valid reduction constraints.

3.4.2 Valid reduction constraints by multiplication with variable z_2

The bipartite graph $\mathcal{G}_2$ is shown in Figure 3.5a. An augmenting path emanating from constraint node c_1 can be found immediately, resulting in the assignment of z_1 to it (see solid line in Figure 3.5b.

We now proceed to consider constraint node c_2. The only variable adjacent to it is z_1 but this is already assigned. Thus, no augmenting path is found in lines 3-7 of Figure 3.2 and we are forced to enter the second search loop (lines 8-15). This considers variable z_1 again and recursively invokes procedure `AugmentPath` (line 10) asking it to search for an augmenting path emanating from the constraint node currently assigned to z_1, namely c_1. This recursive invocation immediately identifies an unassigned variable z_3 as shown in Figure 3.5c. Variable z_3 is assigned to c_1 while z_1 is re-assigned to c_2, resulting in the output set assignment shown in Figure 3.5d.

Constraint node c_3 is completely isolated. Thus, the corresponding invocation of procedure `AugmentPath` at line 15 of Figure 3.3 returns immediately with $PathFound$ false. Consequently, c_3 is added to the set of linear constraints $\mathcal{L}_2$ (line 17).

Finally, attempting to construct an augmenting path emanating from constraint node c_4 visits first variable node z_1 and from there, the constraint node c_2 to which z_1 is currently assigned.

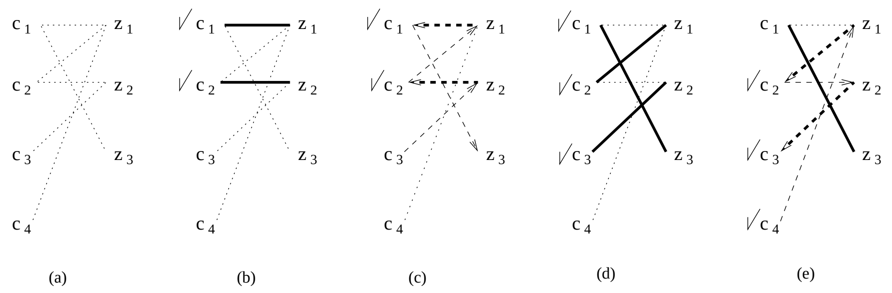


Figure 3.6: Generation of valid reduction constraints for multiplier variable z_3.

No further progress is possible, and therefore `AugmentPath` returns with $PathFound$ false, having visited both c_4 and c_2. Consequently, these are also added to the set $\mathcal{L}$.

We conclude that $\mathcal{L}_2 = \{c_2, c_3, c_4\}$. In this case, the algorithm has detected that c_3 can be multiplied by z_2 without generating any new bilinear terms. Also, the multiplication of c_2 and c_4 by z_2 leads to the creation of only one new bilinear term, z_1z_2 and two linear constraints.

3.4.3 Valid reduction constraints by multiplication with variable z_3

The bipartite graph $\mathcal{G}_3$ is shown in Figure 3.6a. In this case, augmenting paths emanating from both nodes c_1 and c_2 can be found immediately, resulting in the assignment of z_1 to c_1, and z_2 to c_2 respectively (see solid lines in Figure 3.6b).

The application of procedure `AugmentPath` to constraint node c_3 results in tracing an augmenting path $c_3 \to z_2 \to c_2 \to z_1 \to c_1 \to z_3$, as shown in Figure 3.6c. We therefore assign z_3 to c_1, and re-assign z_1 to c_2 and z_2 to c_3. The output set assignment up to this point is shown in Figure 3.6d.

Finally, attempting to locate an augmenting path emanating from c_4 visits nodes z_1, c_2, z_2, c_3. However, no further progress is possible in this case. Procedure `AugmentPath` returns $PathFound$ false. Consequently, all three constraint nodes visited are added to set $\mathcal{L}_3$.

We conclude that $\mathcal{L}_3 = \{c_2, c_3, c_4\}$. Here the algorithm has detected that multiplying these three constraints by z_3 results in the creation of only two new bilinear terms, namely z_1z_3 and z_2z_3.

3.4.4 Valid reduction constraints by multiplication with variables $z_4, \ldots, z_6$

The bipartite graphs $\mathcal{G}_l$ for $l = 4, 5, 6$ are very simple as they contain no variable nodes (i.e. $\mathcal{N}_l^V = \emptyset$). Consequently, the invocation of procedure `AugmentPath` for each and every constraint node c_λ, $\lambda = 1, \ldots, 4$ immediately returns with $PathFound$ false. This results in the constraint sets:

$$\mathcal{L}_l = \{c_1, c_2, c_3, c_4\},\ l = 4, 5, 6$$

This, of course, is a simple consequence of the fact that this problem already contains all possible bilinear terms involving z_4 or z_5 or z_6. Consequently, multiplying any linear constraint by one of these variables does not result in any new bilinear terms and can, therefore, be used to generate a valid reduction constraint.

3.4.5 The reformulated NLP

In summary, the operations described in sections 3.4.1-3.4.4 above result in the following set of 21 reduction constraints:

From multiplication by z_1 :

$$\begin{aligned}
z_1 \times c_2 &\Rightarrow 2z_7 - z_{25} + z_{12} + 3z_{14} - 2z_1 = 0\\
z_1 \times c_3 &\Rightarrow z_{25} + 6z_{12} + 2z_{13} - 3z_{14} + z_1 = 0\\
z_1 \times c_4 &\Rightarrow 2z_7 + z_{12} + 3z_{13} - z_1 = 0
\end{aligned}$$

From multiplication by z_2 :

$$\begin{aligned}
z_2 \times c_2 &\Rightarrow 2z_{25} - z_8 + z_{15} + 3z_{17} - 2z_2 = 0\\
z_2 \times c_3 &\Rightarrow z_8 + 6z_{15} + 2z_{16} - 3z_{17} + z_2 = 0\\
z_2 \times c_4 &\Rightarrow 2z_{25} + z_{15} + 3z_{16} - z_2 = 0
\end{aligned}$$

From multiplication by z_3 :

$$\begin{aligned}
z_3 \times c_2 &\Rightarrow 2z_{26} - z_{27} + z_{18} + 3z_{20} - 2z_3 = 0\\
z_3 \times c_3 &\Rightarrow z_{27} + 6z_{18} + 2z_{19} - 3z_{20} + z_3 = 0\\
z_3 \times c_4 &\Rightarrow 2z_{26} + z_{18} + 3z_{19} - z_3 = 0
\end{aligned}$$

From multiplication by z_4 :

$$\begin{aligned}
z_4 \times c_1 &\Rightarrow z_{12} + 2z_{18} + z_{21} + z_{22} - z_4 = 0\\
z_4 \times c_2 &\Rightarrow 2z_{12} - z_{15} + z_9 + 3z_{22} - 2z_4 = 0\\
z_4 \times c_3 &\Rightarrow z_{15} + 6z_9 + 2z_{21} - 3z_{22} + z_4 = 0\\
z_4 \times c_4 &\Rightarrow 2z_{12} + z_9 + 3z_{21} - z_4 = 0
\end{aligned}$$

From multiplication by z_5 :

$$\begin{aligned}
z_5 \times c_1 &\Rightarrow z_{13} + 2z_{19} + z_{10} + z_{23} - z_5 = 0\\
z_5 \times c_2 &\Rightarrow 2z_{13} - z_{16} + z_{21} + 3z_{23} - 2z_5 = 0\\
z_5 \times c_3 &\Rightarrow z_{16} + 6z_{21} + 2z_{10} - 3z_{23} + z_5 = 0\\
z_5 \times c_4 &\Rightarrow 2z_{13} + z_{21} + 3z_{10} - z_5 = 0
\end{aligned}$$

From multiplication by z_6 :

$$\begin{aligned}
z_6 \times c_1 &\Rightarrow z_{14} + 2z_{20} + z_{23} + z_{11} - z_6 = 0\\
z_6 \times c_2 &\Rightarrow 2z_{14} - z_{17} + z_{22} + 3z_{11} - 2z_6 = 0\\
z_6 \times c_3 &\Rightarrow z_{17} + 6z_{22} + 2z_{23} - 3z_{11} + z_6 = 0\\
z_6 \times c_4 &\Rightarrow 2z_{14} + z_{22} + 3z_{13} - z_6 = 0
\end{aligned}$$

Three new bilinear terms were introduced:

$$\begin{aligned} z_{25} &= z_1 z_2 \\ z_{26} &= z_1 z_3 \\ z_{27} &= z_2 z_3 \end{aligned}$$

augmenting the original set B (cf. equation (3.3)) with the triplets $(25, 1, 2)$, $(26, 1, 3)$ and $(27, 2, 3)$.

In order to determine which bilinear terms may be replaced by these linear constraints, we write the latter in matrix form:

$$Bz'' + Cz' \quad = \quad 0. \tag{3.26}$$

where $z' = (z_1, \ldots, z_6)^T \in \mathbb{R}^6$ and $z'' = (z_7, \ldots, z_{23}, z_{25}, \ldots, z_{27})^T \in \mathbb{R}^{20}$, $B : \mathbb{R}^{20} \to \mathbb{R}^{21}$ and $C : \mathbb{R}^6 \to \mathbb{R}^{21}$. By performing Gaussian elimination with row pivoting on B (and replicating the same row operations on C) we obtain a system of the form:

$$\begin{pmatrix} U & \tilde{B} \\ 0 & 0 \end{pmatrix} \begin{pmatrix} \xi \\ \zeta \end{pmatrix} = \begin{pmatrix} \tilde{C} \\ 0 \end{pmatrix} z'$$

where $\xi = (z_7, \ldots, z_{23}, z_{25})^T \in \mathbb{R}^{18}$, $\zeta = (z_{26}, z_{27})^T \in \mathbb{R}^2$, $U : \mathbb{R}^{18} \to \mathbb{R}^{18}$ is a nonsingular, upper-triangular matrix, $\tilde{B} : \mathbb{R}^2 \to \mathbb{R}^{18}$ and $\tilde{C} : \mathbb{R}^6 \to \mathbb{R}^{18}$. Thus, the reduction constraints determine the variables ξ in terms of ζ and z' via the solution of $U\xi = \tilde{C}z' - \tilde{B}\zeta$, and consequently, the corresponding triplets (i, j, k) can be deleted from the set B in equation (3.25). Overall, the reformulation of the original problem (3.24) involves the introduction of the reduction constraints (3.26) and a much smaller set of just two triplets, namely $B = \{(26, 1, 3), (27, 2, 3)\}$.

3.5 Computational results

We have chosen a selection of bilinear test problems relative to pooling and blending problems to test the efficiency of our algorithm. Pooling and blending involves the determination of optimal amounts of different raw materials that need to be mixed to produce required amounts of end-products with desired properties. Such problems occur frequently in the petrochemical industry and are well known to exhibit multiple local minima.

Here, we use the general blending problem formulation found in [1]:

$$\min_{f,q,x} \sum_{j=1}^{p}\sum_{i=1}^{n_j} c_{ij} f_{ij} - \sum_{k=1}^{r} d_k \sum_{j=1}^{p} x_{jk} \tag{3.27}$$

$$\sum_{i=1}^{n_j} f_{ij} - \sum_{k=1}^{r} x_{jk} = 0, \qquad \forall j \le p \tag{3.28}$$

$$q_{jw} \sum_{k=1}^{r} x_{jk} - \sum_{i=1}^{n_j} \lambda_{ijw} f_{ij} = 0, \qquad \forall j \le p \, \forall w \le l \tag{3.29}$$

$$\sum_{j=1}^{p} x_{jk} \le S_k, \qquad \forall k \le r \tag{3.30}$$

$$\sum_{j=1}^{p} q_{jw} x_{jk} - Z_{kw} \sum_{j=1}^{p} x_{jk} \le 0, \qquad \forall k \le r \, \forall w \le l \tag{3.31}$$

$$f^L \le f \le f^U, q^L \le q \le q^U, x^L \le x \le x^U, \tag{3.32}$$

where f_{ij} is the flow of input stream i into pool j, x_{jk} is the total flow from pool j to product k and q_{jw} is the w-th quality of pool j; p is the number of pools, r the number of products, l the number of qualities, n_j the number of streams; $c_{ij}, d_k, S_k, Z_{kw}, \lambda_{ijw}$ are given parameters.

When the blending problem (3.27)-(3.32) is reformulated to the standard form $[P]$, new variables u_j are created to replace the term $\sum_{k=1}^{r} x_{jk}$ in constraint set (3.29). Thus, we introduce the linear constraints:

$$\sum_{k=1}^{r} x_{jk} - u_j \;=\; 0, \quad \forall j \le p \tag{3.33}$$

and re-write (3.29) as

$$t_{jw} - \sum_{i=1}^{n_j} \lambda_{ijw} f_{ij} = 0, \quad \forall j \le p \;\; \forall w \le l$$

where we have introduced new variables t_{jw} derived via the bilinear constraints:

$$t_{jw} = q_{jw} u_j, \quad \forall j \le p \;\; \forall w \le l$$

More new variables z_{jwk} are created in the standard form to replace the bilinear terms $q_{jw}x_{jk}$ in constraint set (3.31):

$$z_{jwk} = q_{jw} x_{jk}, \quad \forall j \le p \;\; \forall w \le l \;\; \forall k \le r$$

which allows (3.31) to be re-written in linear form as:

$$\sum_{j=1}^{p} z_{jwk} - Z_{kw} \sum_{j=1}^{p} x_{jk} = 0, \quad \forall k \le r \, \forall w \le l.$$

The standard form reformulation of (3.27)-(3.32) is shown below:

$$\min_{f,q,x} \sum_{j=1}^{p} \sum_{i=1}^{n_j} c_{ij} f_{ij} - \sum_{k=1}^{r} d_k \sum_{j=1}^{p} x_{jk}$$

subject to linear constraints:

$$\begin{aligned}
\sum_{i=1}^{n_j} f_{ij} - \sum_{k=1}^{r} x_{jk} &= 0 \qquad \forall j \le p \\
t_{jw} - \sum_{i=1}^{n_j} \lambda_{ijw} f_{ij} &= 0 \qquad \forall j \le p \,\forall w \le l \\
\sum_{k=1}^{r} x_{jk} - u_j &= 0 \qquad \forall j \le p \\
\sum_{j=1}^{p} x_{jk} &\le S_k \qquad \forall k \le r \\
\sum_{j=1}^{p} z_{jwk} - Z_{kw} \sum_{j=1}^{p} x_{jk} &= 0 \qquad \forall k \le r \,\forall w \le l
\end{aligned}$$

with bilinear terms:

$$B = \{(t_{jw}, q_{jw}, u_j), (z_{jwk}, q_{jw}, x_{jk}) \mid j \le p, w \le l, k \le r\}$$

and bounds:

$$f^L \le f \le f^U, \quad q^L \le q \le q^U, \quad x^L \le x \le x^U.$$

We now apply the algorithm of this chapter to the above standard form. This results in a set of reduction constraints derived by multiplying constraints (3.33) by the quality variables q_{jw}:

$$\begin{array}{lll}
\forall j \le p \,\forall w \le l & q_{jw}(\sum_{k=1}^{r} x_{jk} - u_j) & = \\
& \sum_{k=1}^{r} q_{jw} x_{jk} - q_{jw} u_j & = \\
& \sum_{j=1}^{r} z_{jwk} - t_{jw} & = 0.
\end{array}$$

These constraints can be used to eliminate the bilinear constraints $t_{jw} = q_{jw} u_j$.

Another way to interpret what is happening in this particular case is that the constraint sets (3.29) and (3.31) (the bilinear constraints in the general blending problem formulation above) define more bilinear products than is really necessary: if we were to re-write the first term on the left hand side of (3.29) as $\sum_{k=1}^{r} q_{jw} x_{jk}$, we would not need to create all the variables t_{jw}.

	n_{org}	m_{org}	n_{std}	m^l_{std}	$\|BT\|$	$\|RC\|$	$\|NB\|$
Haverly 1	9	8	18	11	6	2	0
Haverly 2	9	8	18	11	6	2	0
Haverly 3	9	8	18	11	6	2	0
Foulds 2	26	16	51	21	20	4	0
Foulds 3	168	48	313	57	136	8	0
Foulds 4	168	48	313	57	136	8	0
Foulds 5	100	40	173	45	68	4	0
Ben-Tal 4	10	8	19	11	6	2	0
Ben-Tal 5	41	27	94	32	48	8	0
example 1	21	30	64	33	40	8	0
example 2	25	42	88	45	60	12	0
example 3	38	49	132	53	90	18	0
example 4	26	35	77	38	48	8	0

Table 3.1: Test problem statistics.

Yet another way of saying this is that distributing products over sums is advantageous. This is in accordance with the considerations found in [130], p. 73. However, it is important to note that, here, this reformulation is determined automatically by a generic algorithm.

Table 3.1 summarizes the main characteristics of the test problems. Here, n_{org}, m_{org} are respectively the number of variables and constraints in the original problem formulation, n_{std} is the number of variables in the standard form $[P]$, m^l_{std} is the number of *linear* constraints in the standard form, $|RC|$ is the number of reduction constraints created, $|BT|$ is the number of bilinear constraints in the standard form before reduction constraint creation and $|NB|$ is the number of new bilinear terms created during the reduction constraint creation procedure. We note that $|NB| = 0$ in all test problems considered.

We now proceed to consider how the addition of reduction constraints affects the tightness of the convex relaxation of the NLP in the context of its solution using deterministic global optimization algorithms. Table 3.2 compares the number of nodes needed to solve the problems of Table 3.1 using eight different codes. The first six are codes described in the literature:

1 = [40] (Foulds, 1992)
2 = [138] (Visweswaran, 1993)
3 = [17] (Ben-Tal, 1994)
4 = [139] (Visweswaran, 1996)
5 = [1] (Adhya, 1999)
6 = [126] (Tawarmalani, 1999)

and their performance is shown in Table 3.2, taken from the corresponding papers. Codes 7 and 8 both correspond to a rather basic implementation of sBB based on the algorithm proposed in [116]. The only difference between them is that code 7 does not incorporate the reduction constraints while code 8 does.

	1	2	3	4	5	6	7	**8**
Haverly 1	5	7	3	12	3	3	31	**1**
Haverly 2		19	3	12	9	9	43	**7**
Haverly 3			3	14	5	3	39	**7**
Foulds 2	9				1	1	131	**7**
Foulds 3	1				1	1	$>$ 20,000	**1**
Foulds 4	25				1	1	$>$ 20,000	**1**
Foulds 5	125				1	1	$>$ 20,000	**1**
Ben-Tal 4		47	25	7	3	3	101	**1**
Ben-Tal 5		42	283	41	1	1	$>$ 200,000	**1**
example 1					6174	1869	11245	**5445**
example 2					10743	2087	83051	**11049**
example 3					79944	7369	$>$ 200,000	**7565**
example 4					1980	157	2887	**1467**

Table 3.2: Numerical results.

As can be seen from the results in column 7 of Table 3.2, the lack of sophisticated algorithmic features regarding, for example, the choice of branching strategy, makes the performance of our basic implementation of the sBB algorithm significantly worse than that of earlier codes presented in the literature. However, the mere addition of the reduction constraints to this basic code improves its performance dramatically, to the extent that it outperforms all but one of the other codes (cf. column 8 of Table 3.2).

It is worth noting that the BARON software (columns 5 and 6 of Table 3.2) implements a Branch-and-Bound method that employs various methods to generate valid cuts; however, none of these is currently equivalent to our method. Our results are in the same order of magnitude as those of BARON (the version published in 1999, fifth column of Table 3.2). However, at the end of 1999 a new version of BARON was described [126] that used an improved branching scheme that cut the number of iterations by an order of magnitude (sixth column). With the notable exception of Example 3, our code cannot generally attain these levels. Nevertheless, our reformulation is complementary to the techniques implemented in BARON and their combination could lead to a further improvement in performance.

3.6 Generalization of the graph-theoretical algorithm

The graph-theoretical algorithm described in Section 3.3 for the identification of useful reduction constraints has some limitations. These arise from the fact that potential multiplications are considered separately for each variable, which may result in some useful multiplications being missed. To understand this point, consider the following very simple example:

$$\left.\begin{array}{rll} \min & x_1^2 + x_2^2 & \\ & x_1 + x_2 & = 1 \\ & 0 \le x_1, x_2 \le 10. & \end{array}\right\} \qquad (3.34)$$

On multiplying the linear constraint $x_1 + x_2 = 1$ by x_1 we obtain $x_1^2 + x_1x_2 = x_1$, which introduces one new bilinear term x_1x_2; thus, this multiplication would not appear to bring any benefit. Similarly, on multiplying the same linear constraint by x_2, we would get $x_1x_2+x_2^2 = x_2$ and thus, again, one new bilinear term x_1x_2; hence this multiplication would not be considered to be beneficial either. Consequently, the algorithm of Section 3.3 applied to this system will not create any reduction constraint.

However, considering the combined effect of the two multiplications, we note that they produce two new linearly independent reduction constraints while introducing only one new bilinear term (namely, x_1x_2). These two reduction constraints can be used to eliminate two bilinear terms from the problem, e.g. x_1^2 and x_2^2, leading to the problem reformulation:

$$\begin{array}{rll} \min & w_1 + w_3 & \\ & x_1 + x_2 = 1 & \\ & w_1 + w_2 = x_1 & \\ & w_2 + w_3 = x_2 & \\ & w_2 = x_1 x_2 & \\ & 0 \le x_1, x_2 \le 10 & \\ & 0 \le w_i \le 100, & i = 1, 2, 3. \end{array}$$

Essentially, the algorithm of Section 3.3 correctly identifies that multiplying the linear constraint by either x_1 or x_2 results in a bilinear term $x_1 x_2$ that did not occur in the original problem. What it misses is the fact that it is the *same* bilinear term in both cases. This is unfortunate as such reformulation may be very beneficial. For instance, we have found that the numerical solution of the example above in its original form with a simple sBB algorithm requires the examination of 255 nodes, while the reformulated one can be solved in a single node.

This motivates an extension to the algorithm of Section 3.3 to consider simultaneously multiplication of linear constraints by all system variables. Instead of creating one bipartite graph for each multiplier variable, we consider one unified bipartite graph comprising two disjoint sets of nodes:

- The ρ-nodes which comprise $m \times n$ nodes ρ_{ij} representing the potential multiplication of constraint i by variable x_j.
- The σ-nodes which comprise n^2 nodes σ_{hk} representing the bilinear term $x_h x_k$.

An edge connecting node ρ_{ij} and σ_{hk} exists if the multiplication of constraint i by variable x_j would give rise to a new bilinear term $x_h x_k$; obviously either $h = j$ or $k = j$ holds.

Having created the bipartite graph, we attempt to trace an augmenting path algorithm similar to that described in Figure 3.2 emanating from each node ρ_{ij}. If no such path is found, then we must have identified a dilation involving ν nodes of type ρ and $\nu - 1$ nodes of type σ. This implies that the variable-constraint multiplications corresponding to these ρ nodes will result in ν new constraints but only $\nu - 1$ new bilinear terms – which is exactly what we are trying to establish.

We apply this generalized algorithm to the simple example problem (3.34). Assuming that the constraint $x_1 + x_2 = 1$ is labelled with index $i = 1$, the unified bipartite graph is shown in

Fig. 3.7. Tracing an augmenting path from node ρ_{11} is , resulting in node σ_{12} being assigned to it. However, no augmenting path emanating from node ρ_{12} can be found. Instead, we identify a dilation comprising nodes ρ_{11}, σ_{12} and ρ_{12}, i.e. in this case, $\nu = 2$. This simply implies that the reformulated problem should involve multiples of the linear constraint by both variables, which introduces a single bilinear term.

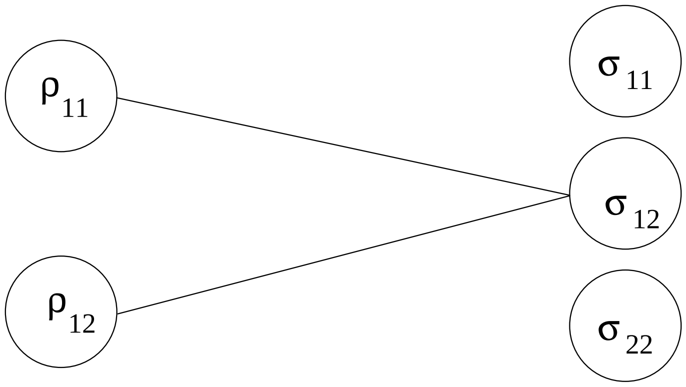


Figure 3.7: Unified bipartite graph for problem (3.34) in the generalized algorithm.

As has been mentioned, the worst-case computational complexity of the dilation-finding algorithm of Section 3.3.3 is proportional to the product of the number of nodes from which the augmenting paths emanate and the number of edges; on the other hand, the average-case complexity is nearer to the sum of these two numbers. The original procedure described in Section 3.3 was applied to a bipartite graph with m linear constraint nodes; in principle, each of these nodes could be connected with each and every one of the n variable nodes; consequently, the number of edges is bounded from above by mn. Therefore, the worst-case and average complexities of this procedure are $O(m^2n)$ and $O(m + mn)$ respectively. Of course, the procedure has to be applied separately for each candidate multiplier variable; therefore, the corresponding total complexities are $O(m^2n^2)$ and $O(mn + mn^2)$ respectively. On the other hand, the procedure described in this section is applied to a bipartite graph with mn ρ-nodes and n^2 σ-nodes. Each ρ-node is potentially connected with up to n σ-nodes, and therefore the number of edges is bounded from above by mn^2 edges. Consequently, the worst-case and average complexities are $O(m^2n^3)$ and $O(mn + mn^2)$ respectively. In conclusion, the procedure of this section has worse worst-case complexity than that of Section 3.3, but similar average complexity.

On the other hand, the memory requirements of the two procedures are very different. With the original algorithm, a graph consisting of $m + n$ nodes and up to mn vertices needs to be stored in memory at any one time, whereas the unified bipartite graph of this section will have

$mn + n^2$ nodes and up to mn^2 edges. For extremely large problems, these requirements may be excessive and special attention may have to be paid to the implementation of the algorithm (one could create the graph “on-the-fly” as the algorithm progresses, deriving the nodes and edges from the problem data at each step).

3.7 Concluding remarks

The work presented in this chapter is based on the fact that geometrical intersections of hyperplanes and nonlinear hypersurfaces corresponding to bilinearities may embed a higher degree of linearity than what is apparent by mere inspection of the defining equations. We have shown that it is possible to exploit this fact so as to reformulate an NLP involving such equality constraints to a form with fewer bilinearities and more linear constraints.

The basic idea of the reformulation is to multiply subsets of the NLP’s linear constraints by one of the system variables x. This creates new linear “reduction” constraints expressed in terms of variables w, each one of which corresponds to a bilinear product, viz. $w_{jk} \equiv x_j x_k$. In general, some of these w variables will already exist in the original NLP while others are new variables introduced by the multiplication.

In principle, any of the original linear constraints in the NLP can be multiplied by any variable; this is the basis of the RLT procedure proposed in [111, 108]. However, here we have focussed on identifying multiplications which result in “valid” sets of reduction constraints, i.e. sets in which the number of constraints exceeds the number of new variables w introduced by the multiplication.

Valid reduction constraints do not affect the feasible region of the original NLP, but they do reduce the feasible region of its convex relaxation, thereby rendering it tighter. To see this, consider the multiplication of a subset of ν of the original linear constraints $Ax = b$ by a single variable x_k. This subset can be written in the form:

$$A'x' + A''x'' = b' \tag{3.35}$$

where the partitioning is such that the bilinear terms $w' \equiv x_k x'$ already occur in the NLP, while $w'' \equiv x_k x''$ do not. Now, if the above is to form a valid reduction constraint set, we must have $x'' \in R^{\nu'}$ where $\nu' < \nu$. We multiply the above constraint by x_k to obtain:

$$A'w' + A''w'' = x_k b' \tag{3.36}$$

If we now apply Gaussian elimination with row pivoting on A'', we can bring the above system to the form:

$$\begin{pmatrix} \tilde{A}' \\ \bar{A}' \end{pmatrix} w' + \begin{pmatrix} \tilde{A}'' \\ 0 \end{pmatrix} w'' = x_k \begin{pmatrix} \tilde{b}' \\ \bar{b}' \end{pmatrix} \tag{3.37}$$

We now note that the bottom block row of the above equation, $\bar{A}'w' = \bar{b}'$ constrains the admissible values of the variables w' which already existed in the original NLP. Although these constraints are redundant with respect to the NLP itself, they are non-redundant with respect to any relaxation which does not enforce the equality $w' = x_k x'$ exactly.

The above argument helps explain the beneficial effects of the proposed reformulation. On one hand, the (linear) valid reduction constraints *always* tighten the convex relaxation of the NLP. On the other hand, the elimination of some of the bilinear terms reduces the size of the convex relaxation (e.g. by obviating the need for McCormick relaxations for these terms). Overall, we have a relaxation that is both tighter and smaller, something that is not always the case with earlier reformulation methods such as RLT.

Finally, it is worth noting that our reformulation relies on the use of graph theoretical algorithms for the identification of valid reduction constraint sets. These algorithms have the advantage of being quite fast even when applied to relatively large systems. On the other hand, they may fail to identify some valid reduction constraint sets. This may occur in cases in which the matrix A'' in equation (3.35) is numerically singular but structurally non-singular.

Chapter 4

A convex relaxation for monomials of odd degree

One of the most effective techniques for the solution of nonlinear programming problems (NLPs) to global optimality is the spatial Branch-and-Bound (sBB) method. This requires the computation of a lower bound to the solution, usually obtained by solving a convex relaxation of the original NLP. The formation and tightness of such a convex relaxation are critical issues in any sBB implementation.

As will be shown in sections 5.2.2.1 and 5.2.2.2, it is possible to form a convex relaxation of any NLP by isolating the nonconvex terms and replacing them with their convex relaxation. Tight convex underestimators are already available for many types of nonconvex term, including bilinear and trilinear products, linear fractional terms, and concave and convex univariate functions. However, terms which are piecewise concave and convex are not explicitly catered for. A frequently occurring example of such a term is x^{2k+1}, where $k \in \mathbb{N}$ and the range of x includes zero. A detailed analysis of the conditions required for concavity and convexity of polynomial functions has been given in [75]; however, the results obtained therein only apply to the convex underestimation of multivariate polynomials with positive variable values. For monomials of odd degree, where the variable ranges over both negative and positive values, no special convex envelopes have been proposed in the literature, and one therefore has to rely either on generic convex relaxations such as those given by Floudas and co-workers (see [13, 7]) or on reformulation in terms of other types of terms for which convex relaxations are available.

In this chapter, we propose convex/concave nonlinear envelopes for odd power terms of the form x^{2k+1} ($k \in \mathbb{N}$), where $x \in [a, b]$ and $a < 0 < b$. These envelopes are continuous and differentiable everywhere in $[a, b]$. We also derive tight linear relaxations. We compare both of these relaxations with relaxations for the same terms derived using other methods. We shall, with a slight abuse of notation, speak about "envelope" to mean the region enclosed between the convex and the concave envelopes.

4.1 Statement of the problem

In [75], the generation of convex envelopes for general univariate functions was discussed. Here we consider the monomial x^{2k+1} in the range $x \in [a, b]$ where $a < 0 < b$. Let c, d be the x-coordinates of the points C, D where the tangents from points A and B respectively meet the curve (see Figure 4.1 below). The shape of the convex underestimator of x^{2k+1} depends on the relative magnitude of b and c. In particular, if $c < b$ (as is the case in Figure 4.1), a convex underestimator can be formed from the tangent from $x = a$ to $x = c$ followed by the curve x^{2k+1} from $x = c$ to $x = b$. On the other hand, if $c > b$ (cf. Figure 4.2), a convex underestimator is simply the straight line passing through A and B.

The situation is similar for the concave overestimator of x^{2k+1} in the range $x \in [a, b]$. If $d > a$, the overestimator is given by the upper tangent from B to D followed by the curve x^{2k+1} from D to A, as shown in Figure 4.1. On the other hand, if $d < a$, the overestimator is just the straight line from A and B. It should be noted that the conditions $c > b$ and $d < a$ cannot both hold simultaneously.

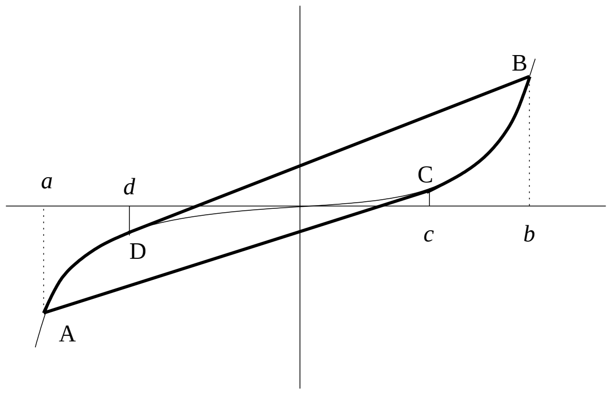


Figure 4.1: Convex envelope of x^{2k+1}.

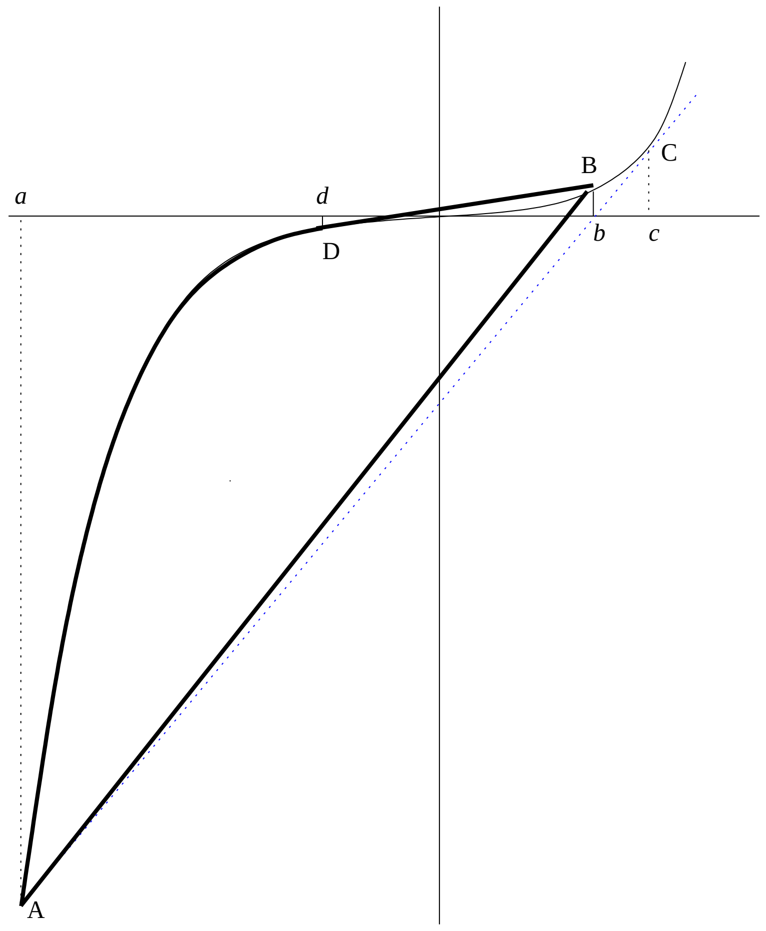


Figure 4.2: The case when $c > b$.

4.2 The tangent equations

The discussion in Section 4.1 indicates that forming the envelope of x^{2k+1} requires the determination of the tangents that pass through points A, C and B, D. Considering the first of these two tangents and equating the slope of the line $\overline{AC}$ to the gradient of x^{2k+1} at $x = c$, we derive the tangency condition:

$$\frac{c^{2k+1} - a^{2k+1}}{c - a} = (2k+1)c^{2k} \tag{4.1}$$

Hence c is a root of the polynomial:

$$P^k(x, a) \equiv (2k)x^{2k+1} - a(2k+1)x^{2k} + a^{2k+1} \tag{4.2}$$

It can be shown by induction on k that:

$$P^k(x,a) = a^{2k-1}(x-a)^2 Q^k(\frac{x}{a}) \tag{4.3}$$

where the polynomial $Q^k(x)$ is defined as:

$$Q^k(x) \equiv 1 + \sum_{i=2}^{2k} i x^{i-1}. \tag{4.4}$$

Thus, the roots of $P^k(x,a)$ can be obtained from the roots[1] of $Q^k(x)$. Unfortunately, polynomials of degree greater than 4 cannot generally be solved by radicals (what is usually called an "analytic solution"). This is the case for $Q^k(x)$ for $k > 2$. For example, the Galois group of $Q^3(x) = 6x^5 + 5x^4 + 4x^3 + 3x^2 + 2x + 1$ is isomorphic to S_5 (i.e. the symmetric group of order 5) which is not soluble since its biggest proper normal subgroup is A_5, the smallest non-soluble group. For details on Galois theory and the solvability of polynomials, see [118].

4.3 The roots of $Q^k(x)$ and their uniqueness

Unlike $P^k(x,a)$, the polynomial $Q^k(x)$ does not depend on the range of x being considered. Moreover, as shown formally in Section 4.3.1 below, $Q^k(x)$ has exactly one real root, r_k, for any $k \geq 1$, and this lies in $[-1 + 1/2k, -0.5]$. Hence, the roots of $Q^k(x)$ for different k can be computed *a priori* to arbitrary precision using simple numerical schemes (e.g. bisection). A table of these roots is presented in Table 4.1 for $k \leq 10$.

k	r_k	k	r_k
1	-0.5000000000	6	-0.7721416355
2	-0.6058295862	7	-0.7921778546
3	-0.6703320476	8	-0.8086048979
4	-0.7145377272	9	-0.8223534102
5	-0.7470540749	10	-0.8340533676

Table 4.1: Numerical values of the roots of $Q^k(x)$ for $k = 1, .., 10$ (to 10 significant digits).

[1]Although $P^k(x,a)$ has the additional root $x = a$, this is not of practical interest.

4.3.1 Bounding the roots of $Q^k(x)$

In this section, we show that $Q^k(x)$ has exactly one real root, which lies in the interval $[-1 + \frac{1}{2k}, -\frac{1}{2}]$.

4.3.1 Proposition

For all $k \in \mathbb{N}$, the following properties hold:

$$\left.\begin{array}{rcl} Q^k(0) & = & 1 \\ Q^k(-1) & = & -k \end{array}\right\} \tag{4.5}$$

$$\forall x > 0 \ \left(\frac{dQ^k(x)}{dx} > 0 \right) \tag{4.6}$$

$$\forall x \leq -1 \ \left(Q^k(x) < 0 \right) \tag{4.7}$$

Proof. (4.5): $Q^k(0) = 1$ by direct substitution in (4.4). Also $Q^k(-1) = 1 + \sum_{i=2}^{2k} i(-1)^{i-1} = \sum_{i=1}^{k}(2i-1) - \sum_{i=1}^{k} 2i = -k$.
(4.6): $\frac{dQ^k(x)}{dx} = \sum_{i=1}^{2k-1} i(i+1)x^{i-1}$, hence it is greater than zero whenever $x > 0$.
(4.7): For $x \neq 0$, we can rewrite $Q^k(x)$ as $\sum_{i=1}^{k} x^{2i-2}[2i(x+1)-1]$. For $x \leq -1$, we have $x^{2i-2} > 0$ and $[2i(x+1)-1] < 0$, thus each term of the sum is negative. □

From the above proposition and the continuity of x^{2k+1}, we can conclude that:

1. there is at least one root between -1 and 0 (property (4.5));
2. there are no roots for $x \geq 0$ (property (4.6) and the fact that $Q^k(0) > 0$);
3. there are no roots for $x \leq -1$ (property (4.7)).

4.3.2 Lemma

For all $k \in \mathbb{N}$, the real roots of $Q^k(x)$ lie in the interval $[-1 + \frac{1}{2k}, -\frac{1}{2}]$.

Proof. This is proved by induction on k. For $k = 1$, $Q^1(x) \equiv 1 + 2x$ has one real root at $x = -\frac{1}{2}$ which lies in the set $[-1 + \frac{1}{2}, -\frac{1}{2}]$. In particular, $Q^1(x) < 0$ for all $x < -1 + \frac{1}{2}$ and $Q^1(x) > 0$ for all $x > -\frac{1}{2}$.

We now make the inductive hypothesis that, for all $j < k$, $Q^j(x) > 0$ for all $x > -\frac{1}{2}$ and $Q^j(x) < 0$ for all $x < -1 + \frac{1}{2j}$ and prove that the same holds for $j = k$. Using (4.4), we can write $Q^k(x) = Q^{k-1}(x) + x^{2k-2}(2kx + 2k - 1)$ for all $k > 1$. Since x^{2k-2} is always positive, we have that:

$$\begin{aligned} Q^k(x) > Q^{k-1}(x) \quad &\text{if} \quad x > -1 + \frac{1}{2k} \\ Q^k(x) < Q^{k-1}(x) \quad &\text{if} \quad x < -1 + \frac{1}{2k} \end{aligned}$$

for all $k > 1$. Now, since $-\frac{1}{2} > -1 + \frac{1}{2k}$ for all $k > 1$, $Q^k(x) > Q^{k-1}(x) > 0$ for all $x > -\frac{1}{2}$ by inductive hypothesis.

Furthermore, by the inductive hypothesis, $Q^{k-1}(x) < 0$ for all $x < -1 + \frac{1}{2(k-1)}$; since $\frac{1}{2k} < \frac{1}{2(k-1)}$, it is also true that $Q^{k-1}(x) < 0$ for all $x < -1 + \frac{1}{2k}$. But since, as shown above, $Q^k(x) < Q^{k-1}(x)$ for all $x < -1 + \frac{1}{2k}$, we can deduce that $Q^k(x) < 0$ for all $x < -1 + \frac{1}{2k}$.

We have thus proved that, for all $k > 0$,

$$Q^k(x) > 0 \quad \text{if} \quad x > -\frac{1}{2} \tag{4.8}$$

$$Q^k(x) < 0 \quad \text{if} \quad x < -1 + \frac{1}{2k}. \tag{4.9}$$

The proof of the lemma follows from (4.8), (4.9) and the continuity of $Q^k(x)$. □

4.3.3 Theorem

For all $k \in \mathbb{N}$, $Q^k(x)$ has exactly one real root, which lies in the interval $[-1 + \frac{1}{2k}, -\frac{1}{2}]$.

Proof. Consider the polynomial $P^k(x, 1) = 2kx^{2k+1} - (2k+1)x^{2k} + 1$ defined by (4.2). By virtue of (4.3), we have the relation $P^k(x, 1) = (x-1)^2 Q^k(x)$. Consequently, $P^k(x, 1)$ and $Q^k(x)$ have exactly the same roots for $x < 1$. Therefore (Lemma 4.3.2), all negative real roots of $P^k(x, 1)$ lie in the interval $[-1 + 1/2k, -1/2]$, and there is at least one such root.

Now, $P^k(x, 1)$ can be written as $P^k(x, 1) = q_1^k(x) + q_2^k(x) + 1$, where $q_1^k(x) = 2kx^{2k+1}$ and $q_2^k(x) = -(2k+1)x^{2k}$. Since q_1 is a monomial of odd degree, it is monotonically increasing in $[-1, 0]$. Since q_2 is a monomial of even degree with a negative coefficient, it is also monotonically increasing in $[-1, 0]$.

Overall, then, $P^k(x, 1)$ is monotonically increasing in $[-1, 0]$, and consequently in the interval $[-1 + 1/2k, -1/2]$ where all its negative real roots lie. Therefore, there can be only one such root, which proves that $Q^k(x)$ also has a unique root in this interval. □

4.4 Nonlinear convex envelopes

If the roots shown in the second column of Table 4.1 are denoted by r_k, then the tangent points c and d in Figure 4.1 are simply $c = r_k a$ and $d = r_k b$. The lower and upper tangent lines are given respectively by:

$$a^{2k+1} + \frac{c^{2k+1} - a^{2k+1}}{c-a}(x-a) \tag{4.10}$$

$$b^{2k+1} + \frac{d^{2k+1} - b^{2k+1}}{d-b}(x-b) \tag{4.11}$$

Hence, the envelope for $z = x^{2k+1}$ when $x \in [a,b]$ and $a < 0 < b$:

$$l_k(x) \leq \quad z \quad \leq u_k(x) \tag{4.12}$$

is as follows:

- If $c < b$, then:

$$l_k(x) = \begin{cases} a^{2k+1}\left(1 + R_k\left(\frac{x}{a} - 1\right)\right) & \text{if } x < c \\ x^{2k+1} & \text{if } x \geq c \end{cases} \tag{4.13}$$

  otherwise:

$$l_k(x) = a^{2k+1} + \frac{b^{2k+1} - a^{2k+1}}{b-a}(x-a) \tag{4.14}$$

- If $d > a$, then:

$$u_k(x) = \begin{cases} x^{2k+1} & \text{if } x \leq d \\ b^{2k+1}\left(1 + R_k\left(\frac{x}{b} - 1\right)\right) & \text{if } x > d \end{cases} \tag{4.15}$$

  otherwise:

$$u_k(x) = a^{2k+1} + \frac{b^{2k+1} - a^{2k+1}}{b-a}(x-a) \tag{4.16}$$

where we have used the constant $R_k \equiv \frac{r_k^{2k+1}-1}{r_k-1}$. If the range of x is unbounded either below or above we take the limits of $l_k(x)$ and $u_k(x)$ as $a \to \infty$ or $b \to \infty$.

By construction, the above convex underestimators and concave overestimators are continuous and differentiable everywhere. Moreover, they form the convex envelope of of x^{2k+1}, as the following theorem shows.

4.4.1 Theorem

The convex underestimator and concave overestimator of x^{2k+1} for $x \in [a,b]$ where $a < 0 < b$ and $k \in \mathbb{N}$, given in equations (4.12)-(4.16), are as tight as possible.

Proof. First, consider the case where $a < d < 0 < c < b$. As the convex underestimator between c and b is the curve itself, no tighter one can be found in that range. Furthermore, the convex underestimator between a and c is a straight line connecting two points on the original curve, so again it is the tightest possible.

It only remains to show that $l_k(x)$ is convex for any small neighbourhood of c. Consider the open interval $(c-\varepsilon, c+\varepsilon)$, and the straight line segment $\Gamma(c, c+\varepsilon)$ with endpoints $(c, l_k(c))$, $(c+\varepsilon, l_k(c+\varepsilon))$. Because for all $x \geq c$, $l_k(x) \equiv x^{2k+1}$ is convex, all points in $\Gamma(c, c+\varepsilon)$ lie above the underestimator. If we now consider $\Gamma(c-\varepsilon, c+\varepsilon)$, its slope is smaller than the slope of $\Gamma(c, c+\varepsilon)$ (because the point with coordinate $c-\varepsilon$ moves on the tangent of the curve at c), yet the right endpoint $c+\varepsilon$ of the segments is common. Thus all points in $\Gamma(c-\varepsilon, c+\varepsilon)$ also lie above the underestimator $l_k(x)$. Since ε is arbitrary, the claim holds. A similar argument holds for the overestimator between a and d.

The cases where $a < d < 0 < b \leq c$ and $d \leq a < 0 < c < b$ are simpler as the underestimator is a straight line in the whole range of x. □

4.5 Tight linear relaxation

The convex envelope presented in Section 4.4 is nonlinear. As convex relaxations are used to solve a local optimization problem at each node of the search tree examined by the sBB algorithm, using a linear relaxation instead of a nonlinear one may have a significant impact on computational cost. We can relax the nonlinear envelope to a linear relaxation by dropping the “follow the curve” requirements on either side of the tangency points, and using the lower and upper tangent as convex underestimator and concave overestimator respectively, as follows:

$$a^{2k+1}\left(1+R_k\left(\frac{x}{a}-1\right)\right) \quad \leq z \leq \quad b^{2k+1}\left(1+R_k\left(\frac{x}{b}-1\right)\right) \tag{4.17}$$

We can tighten this relaxation further by drawing the tangents to the curve at the endpoints A, B, as shown in Figure 4.3. This is equivalent to employing the following constraints:

$$(2k+1)b^{2k}x - 2kb^{2k+1} \quad \leq z \leq \quad (2k+1)a^{2k}x - 2ka^{2k+1} \tag{4.18}$$

in addition to those in (4.17).

As has been noted in Section 4.1, when $c > b$, the underestimators on the left hand sides of (4.17) and (4.18) should be replaced by the line $a^{2k+1} + \frac{b^{2k+1}-a^{2k+1}}{b-a}(x-a)$ through points A and

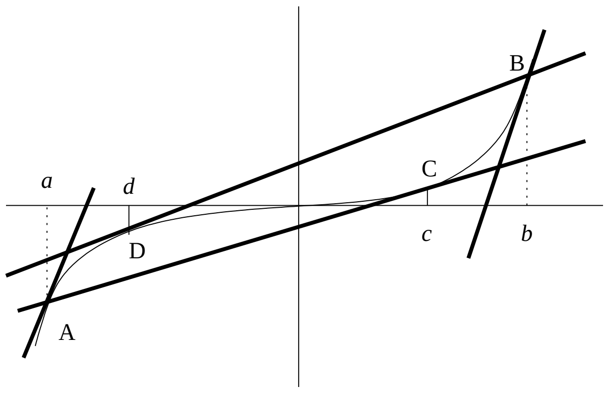


Figure 4.3: Tight linear relaxation of x^{2k+1}.

B (see Figure 4.2). On the other hand, if $d < a$, this line should be used to replace the concave overestimators on the right hand sides of (4.17) and (4.18). The linear relaxation constraints are summarized in Table 4.2.

$c < b$ and $d > a$	$c > b$ and $d > a$	$c < b$ and $d < a$
$z \geq a^{2k+1}(1 + R_k(\frac{x}{a} - 1))$	$z \geq a^{2k+1} + \frac{b^{2k+1}-a^{2k+1}}{b-a}(x-a)$	$z \geq a^{2k+1}(1 + R_k(\frac{x}{a} - 1))$
$z \leq b^{2k+1}(1 + R_k(\frac{x}{b} - 1))$	$z \leq b^{2k+1}(1 + R_k(\frac{x}{b} - 1))$	$z \leq a^{2k+1} + \frac{b^{2k+1}-a^{2k+1}}{b-a}(x-a)$
$z \geq (2k+1)b^{2k}x - 2kb^{2k+1}$	$z \leq (2k+1)a^{2k}x - 2ka^{2k+1}$	–
$z \leq (2k+1)a^{2k}x - 2ka^{2k+1}$	–	$z \geq (2k+1)b^{2k}x - 2kb^{2k+1}$

Table 4.2: Summary of linear relaxations for $z = x^{2k+1}$, $x \in [a, b]$, $a < 0 < b$.

4.6 Comparison to other relaxations

This section considers two alternative convex/concave relaxations of the monomial x^{2k+1} where the range of x includes 0, and compares them with both the nonlinear envelope and linear relaxation proposed in this paper.

4.6.1 Reformulation in terms of bilinear products

One possible way of determining a convex/concave relaxation for $z = x^{2k+1}$, where $a \leq x \leq b$ and $a < 0 < b$, is via its exact reformulation in terms of a bilinear product of x and the convex monomial x^{2k}:

$$\begin{aligned} z &= wx \\ w &= x^{2k} \\ a \leq\ & x \leq b \\ 0 \leq\ & w \leq w^U = \max\{a^{2k}, b^{2k}\} \end{aligned}$$

By replacing the bilinear term wx with the standard linear convex envelope proposed by [80] (see equations (5.10)-(5.13)), and the convex univariate term x^{2k} with the envelope given by the function itself as the underestimator and the secant as the overestimator, we obtain the following constraints:

$$\begin{aligned} aw \leq\ & z \leq bw \\ w^U x + bw - w^U b \leq\ & z \leq w^U x + aw - w^U a \\ x^{2k} \leq\ & w \leq a^{2k} + \frac{b^{2k} - a^{2k}}{b-a}(x-a) \\ a \leq\ & x \leq b \\ 0 \leq\ & w \leq w^U \end{aligned}$$

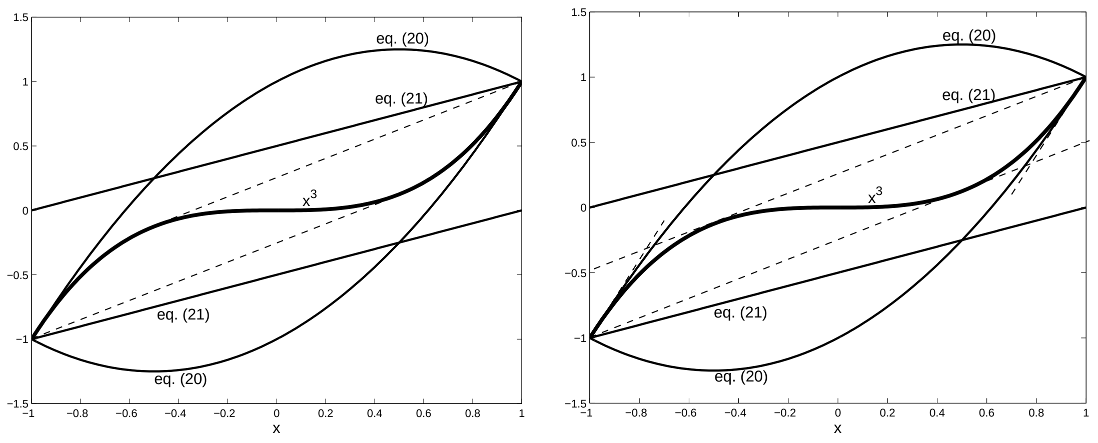
After some algebraic manipulation, we can eliminate w to obtain the following nonlinear convex relaxation for z:

$$\frac{w^U a}{a-b}(x-b) \leq z \leq \frac{w^U b}{b-a}(x-a) \tag{4.19}$$

$$bx^{2k} + w^U(x-b) \leq z \leq ax^{2k} + w^U(x-a) \tag{4.20}$$

$$a\left(a^{2k} + \frac{b^{2k} - a^{2k}}{b-a}(x-a)\right) \leq z \leq b\left(a^{2k} + \frac{b^{2k} - a^{2k}}{b-a}(x-a)\right) \tag{4.21}$$

Figure 4.4 shows the convex/concave relaxation for x^3 for $x \in [-1, 1]$ obtained using (4.19)-(4.21). It also compares it with the nonlinear envelope of Section 4.4 (dashed lines in Figure 4.4a) and the linear relaxation of Section 4.5 (dashed lines in Figure 4.4b).



a. Comparison with nonlinear envelope b. Comparison with tight linear relaxation

Figure 4.4: Convex relaxation of x^3 by reformulation to bilinear product.

As can be seen from Figure 4.4a, the convex relaxation (4.19)-(4.21) is generally similar to that of Section 4.4 (in that both the underestimator and the overestimator consist of a straight line joined to a curve), but not as tight. This is to be expected in view of theorem 4.4.1.

On the other hand, the convex relaxation (4.19)-(4.21) is slightly tighter than the linear relaxation of Section 4.5 in the sub-interval $[a, e]$ where e is the point at which the curve on the right hand side of (4.20) intersects the tangent line on the right hand side of (4.18); and also in the sub-interval $[f, b]$ where f is the point at which the curve on the left hand side of (4.20) intersects the tangent line on the left hand side of (4.18). However, the linear relaxation of Section 4.5 is tighter everywhere else.

4.6.2 Underestimation through α parameter

An alternative approach to deriving convex relaxations of general non-convex functions is the αBB algorithm (see [13], [7]). In this case, the convex underestimator $\mathcal{L}_k(x)$ is given by $x^{2k+1} + \alpha_k(x-a)(x-b)$, where α_k is a positive constant that is sufficiently large to render the second derivative $d^2\mathcal{L}_k(x)/dx^2$ positive for all $x \in [a, b]$. Similarly, the concave overestimator $\mathcal{U}_k(x)$ is given by $x^{2k+1} - \beta_k(x-a)(x-b)$ where β_k is sufficiently large to render $d^2\mathcal{U}_k(x)/dx^2$ negative for all $x \in [a, b]$. It can easily be shown that the above conditions are satisfied by the values:

$$\alpha_k = k(2k+1)|a|^{2k-1} \tag{4.22}$$
$$\beta_k = k(2k+1)b^{2k-1}. \tag{4.23}$$

The convex relaxation for the case of $k = 1$ (i.e. the function x^3) obtained using the above approach in the domain $x \in [-1, +1]$ is shown in Figure 4.5. It is evident that it is looser than those shown in figs. 4.1 and 4.3.

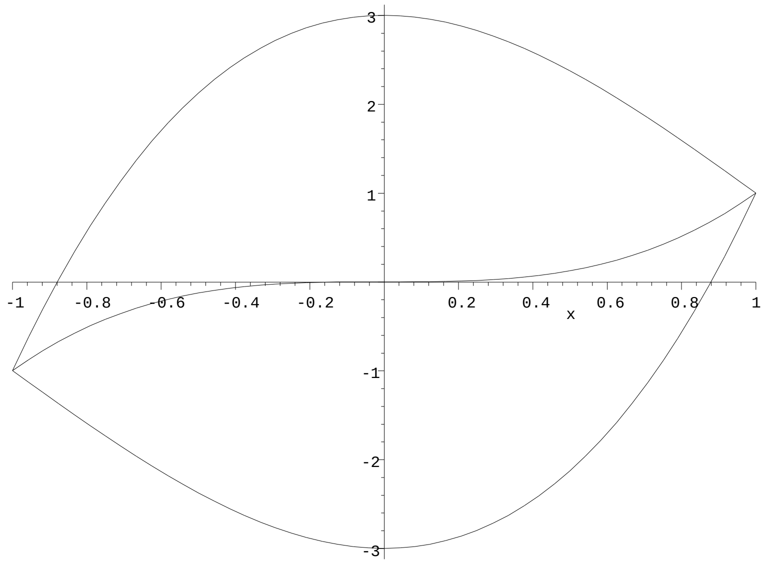


Figure 4.5: Convex relaxation of x^3 by the α method.

4.7 Computational results

In order to illustrate the difference between two of the relaxations described above for x^{2k+1} (those of Section 4.5 and 4.6.1), we solved the problem:

$$\left.\begin{array}{rl} \min_{x,y} & x - y \\ y & = x^{2k+1} \\ -1 & \le x, y \le 1 \end{array}\right\} \tag{4.24}$$

to global optimality using the spatial Branch-and-Bound algorithm described in [116] within the *oo*$\mathcal{OPS}$ implementation [73], both with the tight linear relaxation (equation (4.17)-(4.18))

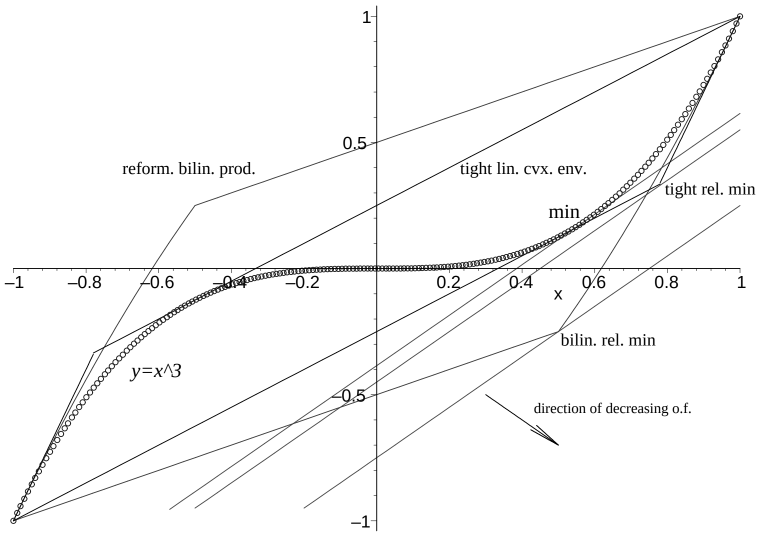


Figure 4.6: Graphical description of simple test problem for $k = 1$ (in 2D).

and with the alternative convex relaxation (equation (4.19)-(4.21)). Table 4.3 lists the number of iterations taken by the algorithm when k varies. The first column lists the results relative to the novel convex relaxation (Section 4.5), the second those relative to the alternative convex relaxation (Section 4.6.1).

The results clearly substantiate the theory: the novel convex relaxation gives better performance in comparison to the alternative relaxation based on reformulation to bilinear product. In the case of x^3, we can see why this happens in figures 4.6 and 4.7. The direction of minimization of the objective function $x - y$ is such that the minimum over the original feasible region is very near the minimum over the novel convex relaxation. However, the minimum over the alternative convex relaxation is further away. Hence the performance gain.

4.8 Conclusion

We have proposed a convex envelope for monomials of odd degree when the range of the defining variable includes zero, i.e. when they are piecewise convex and concave. We have then compared it with other possible relaxations (based on reformulation to bilinear product and on

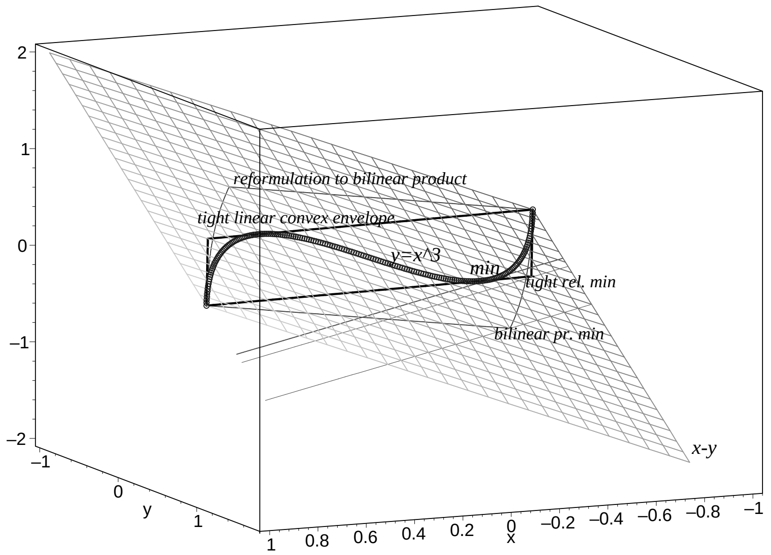


Figure 4.7: Graphical description of simple test problem for $k = 1$ (in 3D).

α parameter under- and overestimators) and shown that the former performs better than the latter when tested in a Branch-and-Bound algorithm.

k	Iterations (novel rel.)	Iterations (alternative rel.)	k	Iterations (novel rel.)	Iterations (alternative rel.)
1	17	21	8	19	27
2	17	31	9	21	27
3	17	27	10	7	25
4	17	29	11	7	25
5	19	31	12	7	25
6	19	27	13	7	25
7	19	27	14	7	23

Table 4.3: Numerical results from the simple test problem.

Chapter 5

Spatial Branch-and-Bound algorithm with symbolic reformulation

The previous two chapters considered issues relating to the (re-)formulation and tight convexification of NLP problems in the context of spatial branch-and-bound (sBB) algorithms for global optimization. We now turn our attention to the sBB algorithm itself and its computer implementation in a form that can be used as a component within larger software systems such as process modelling tools. An important consideration in this context is the large amount of both numerical and symbolic information required by sBB algorithms.

We start with an overview of the common structure of most sBB algorithms reported in the literature. We then provide a more detailed description of Smith's sBB algorithm [114, 116] which forms the basis for our work and propose some algorithmic improvements to improve its performance. Lastly, we describe $oo\mathcal{OPS}$ (object-oriented OPtimization System), a general software framework for defining and solving optimization problems, and the implementation of our algorithm within this framework.

5.1 Overview of spatial Branch-and-Bound algorithms

The spatial Branch-and-Bound algorithm described in this chapter solves NLPs in the following form:

$$\left.\begin{array}{rcl} \min_x & f(x) & \\ \alpha & \leq g(x) \leq & \beta \\ a & \leq x \leq & b \end{array}\right\} \tag{5.1}$$

where $x \in \mathbb{R}^n$ are the (continuous) problem variables, $f : \mathbb{R}^n \to \mathbb{R}$ is the objective function (which may be nonconvex), $g : \mathbb{R}^n \to \mathbb{R}^m$ are a vector of generally nonconvex functions, α, β are the lower and upper bounds of the constraints[1], and a, b are the lower and upper bounds of the variables.

The general mathematical structure and properties of sBB algorithms aimed at solving nonconvex NLPs were studied in Section 1.4, and their convergence proofs are similar to that of theorem 1.4.1. The Branch-and-Reduce method [99] is an sBB algorithm with strong emphasis on variable range reduction. The αBB algorithm [13, 2, 6, 5] is an sBB whose main feature is that the convex underestimators for general twice-differentiable nonconvex terms can be constructed automatically. The reduced-space Branch-and-Bound algorithm [32] identifies *a priori* a reduced set of branching variables so that less branching is required. The generalized Branch-and-Cut framework proposed in [61] derived cuts from violated constraints in three sub-problems related to the original problem.

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

1. (Initialization) Initialize a list of regions to a single region comprising the entire set of variable ranges. Set the convergence tolerance ε and the best objective function value $U := \infty$. Optionally, perform optimization-based bounds tightening.

2. (Choice of Region) If the list of regions is empty, terminate the algorithm with solution U. Otherwise, choose a region (the "current region") from the list. Delete this region from the list. Optionally, perform feasibility-based bounds tightening on this region.

3. (Lower Bound) Generate a convex relaxation of the original problem in the selected region and solve it to obtain an underestimation l of the objective function. If $l > U$ or the relaxed problem is infeasible, go back to step 2.

[1]Equality constraints can be specified by setting $\alpha = \beta$.

4. (Upper Bound) Attempt to solve the original (generally nonconvex) problem in the selected region to obtain a (locally optimal) objective function value u. If this fails, set $u := +\infty$.

5. (Pruning) If $U > u$, set $U := u$. 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.

5.2 Smith's sBB algorithm

The most outstanding feature of Smith's sBB algorithm is the automatic construction of the convex relaxation via symbolic reformulation [114, 116]. This involves identifying all the nonconvex terms in the problem and replacing them with the respective convex relaxations. The algorithm that carries out this task is symbolic in nature as it has to recognize the nonconvex operators in any given function.

Smith assumes that the NLP solved by his algorithm is of the form:

$$\left.\begin{array}{rcl} \min_x & f(x) & \\ \bar{g}(x) & = & 0 \\ a & \leq x \leq & b \end{array}\right\} \tag{5.2}$$

introducing slack variables to convert any inequalities to the equality constraints $\bar{g}(x) = 0$ above.

Below, we consider some of the key steps of the algorithm in more detail.

5.2.1 Choice of region (step 2)

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.

5.2.2 Convex relaxation (step 3)

The convex relaxation solved at step 3 of the algorithm aims to find a guaranteed lower bound to the objective function value. In most global optimization algorithms, convex relaxations are obtained for problems in special (e.g. factorable) forms. In Smith's sBB, the convex relaxation is calculated automatically for a problem in the most generic form (1.1) provided this is available in closed analytic form, which allows symbolic manipulation to be applied to it.

The convex relaxation is built in two stages: first the problem is reduced to a *standard form* where nonlinear terms of the same type are collected in lists; then each nonlinear term is replaced by the corresponding convex under- and overestimators. The standard form also provides ways to simplify branch point calculation, branch variable choice and feasibility based bounds tightening.

5.2.2.1 Reformulation to standard form

This is the first stage toward the construction of the convex relaxation of the original problem via symbolic manipulation of the variables and constraints. In this form, the problem nonlinearities are isolated from the rest of the problem and thus are easy to tackle by symbolic and numerical procedures.

Smith defined the following standard form:

$$\min x_{obj} \tag{5.3}$$
$$l \leq \quad Ax \quad \leq u \tag{5.4}$$
$$x_k \quad = \quad x_i x_j \qquad \forall (i,j,k) \in \mathcal{M} \tag{5.5}$$
$$x_k \quad = \quad \frac{x_i}{x_j} \qquad \forall (i,j,k) \in \mathcal{D} \tag{5.6}$$
$$x_k \quad = \quad x_i^\nu \qquad \forall (i,k,\nu) \in \mathcal{P} \tag{5.7}$$
$$x_k \quad = \quad f_\mu(x_i) \qquad \forall (i,k,\mu) \in \mathcal{U} \tag{5.8}$$
$$x^L \leq \quad x \quad \leq x^U \tag{5.9}$$

where $x = (x_1, \ldots, x_n)$ are the problem variables, A is a constant, usually sparse, matrix, l, u are the linear constraint bounds, and x^L, x^U are the variable bounds. Constraints (5.5)–

(5.8), called the *defining constraints*, include all the problem nonlinearities: $\mathcal{M}, \mathcal{D}$ are sets of triplets of variable indices which define bilinear and linear fractional terms respectively. $\mathcal{P}$ is a set of triplets defining power terms; each triplet comprises two variable indices i, k and the corresponding power exponent $\nu \in \mathbb{R}$. $\mathcal{U}$ is a set of triplets which define terms involving univariate functions f_μ; each triplet comprises two variable indices i, k and a third index μ that identifies the type of the univariate function being applied (e.g. $\exp(.)$, $\ln(.)$).

Each of the constraints (5.5)-(5.8) has one of the forms:

$$\begin{aligned} added\, variable &= operand\ (binary\ operator)\ operand \\ added\, variable &= (unary\ operator)\ operand \end{aligned}$$

where *operand* is an original or an “added” variable i.e. one added to the original set of variables by the standardization procedure.

In the interests of simplicity, the objective function is replaced by the added variable x_{obj}, and a constraint of the form:

$$x_{obj} = objective\ function$$

is added to the linear constraints.

An efficient algorithm for reducing an NLP to standard form has been described in detail in [114, 116]. It works by recursively tackling nonlinear terms in the problem and forming linear and defining constraints as it goes along, always aiming at introducing the minimal number of new variables.

5.2.2.2 Convexification

This is the second stage of the process where the actual convex relaxation of the original problem within the current region $[x^L, x^U]$ is built. The algorithm for convexification is entirely symbolic (as opposed to numeric) and hence performs very efficiently even in the presence of very complicated mathematical expressions.

Having reduced a problem to standard form, we replace every nonconvex term with a convex envelope consisting of convex over- and underestimating inequality constraints. The rules we follow to construct over- and underestimators are as follows:

1. $x_i = x_j x_k$ is replaced by four linear inequalities (McCormick's envelopes, [80]):

$$\begin{aligned} x_i &\geq x_j^L x_k + x_k^L x_j - x_j^L x_k^L && (5.10) \\ x_i &\geq x_j^U x_k + x_k^U x_j - x_j^U x_k^U && (5.11) \\ x_i &\leq x_j^L x_k + x_k^U x_j - x_j^L x_k^U && (5.12) \\ x_i &\leq x_j^U x_k + x_k^L x_j - x_j^U x_k^L && (5.13) \end{aligned}$$

2. $x_i = x_j / x_k$ is reformulated to $x_i x_k = x_j$ and the convexification rules for bilinear terms (5.10)-(5.13) are applied.

3. $x_i = f_\mu(x_j)$ where f_μ is concave univariate is replaced by two inequalities: the function itself and the secant:

$$\begin{aligned} x_i &\leq f_\mu(x_j) && (5.14) \\ x_i &\geq f_\mu(x_j^L) + \frac{f_\mu(x_j^U) - f_\mu(x_j^L)}{x_j^U - x_j^L}(x_j - x_j^L) && (5.15) \end{aligned}$$

4. $x_i = f_\mu(x_j)$ where f_μ is convex univariate is replaced by:

$$\begin{aligned} x_i &\leq f_\mu(x_j^L) + \frac{f_\mu(x_j^U) - f_\mu(x_j^L)}{x_j^U - x_j^L}(x_j - x_j^L) && (5.16) \\ x_i &\geq f_\mu(x_j) && (5.17) \end{aligned}$$

5. $x_i = x_j^\nu$ where $0 < \nu < 1$ is treated as a concave univariate function in the manner described in point 3 above.

6. $x_i = x_j^{2m}$ for any $m \in \mathbb{N}$ is treated as a convex univariate function in the manner described in point 4 above.

7. $x_i = x_j^{2m+1}$ for any $m \in \mathbb{N}$ can be convex, concave, or piecewise convex and concave with a turning point at 0. If the range of x_j does not include 0, the function is convex or concave and falls into a category described above. Smith does not specify a convex envelope for the case where the range includes 0; however, it is desirable to use the linear convex and concave relaxations described in Chapter 4.

5.2.3 Branching (step 7)

There are many branching strategies [31] 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. Smith uses the solution 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 (step 3) is used. He then identifies the nonlinear (nonconvex) term with the largest error with respect to its convex relaxation. The branch variable is chosen as the variable whose value at the branching point is nearest to the midpoint of its range.

5.2.4 Bounds tightening

These procedures arrear in steps 1 and 2 of the algorithm structure outlined in Section 5.1. 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.

5.2.4.1 Optimization-based bounds tightening (step 1)

This is a computationally expensive procedure which involves solving at least $2n$ convex NLPs (or LPs if a linear 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. The following procedure will construct sequences $\{x^L\}^{[k]}$, $\{x^U\}^{[k]}$ of lower and upper bounds which converge to bounds that are at least as tight as, and probably 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

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

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.

5.2.4.2 Feasibility-based bounds tightening (step 2)

This procedure 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 $l \leq Ax \leq b$ where $A = (a_{ij})$, it can be shown that, for all $1 \leq j \leq n$:

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

As pointed out by Smith ([114], p.202), feasibility-based bounds tightening can also be carried out for certain types of nonlinear constraints.

5.3 Improvements to Smith's sBB algorithm

In this section we present some simple improvements to Smith's work based on observations about the theory of sBB. All of these leave the convergence properties of the algorithm intact but reduce its cost.

5.3.1 Avoiding the introduction of slack variables

The computational efficiency of optimization codes is adversely affected by the size of the problem in terms of the number of variables. It therefore makes sense to try to reduce this number, either by dimensionality arguments (see Section 2.2.2) or by reducing the number of intermediate variables generated by the symbolic manipulation procedures. Here, we use the latter approach.

One of the prerequisites of the standardization procedure stated by Smith is that the original problem consists only of equality constraints of the form $g(x) = 0$ (see [114], p.190). All the inequality constraints should therefore be transformed into equality constraints via the introduction of slack variables as described in Section 5.2.2.1 (also see section 2.2.1). This produces a number of new problem variables equal to the number of inequality constraints. For cases where all or most of the constraints are inequalities, the additional computational overhead resulting from this step is substantial. If one takes into account the fact that the most generic problem form 5.1 has two inequalities for each constraint expression (i.e. each constraint has a lower and an upper bound, as in $-1 \leq x + y \leq 3$, for example), the actual number of slack variables is doubled.

Fortunately, it turns out that slack variables are not needed for the standardization process. More precisely, the standardization process only acts on the expression $g(x)$ within a constraint of the form $l \leq g(x) \leq u$; at no point is the fact that $g(x) = 0$ used. Thus, the standardization can be applied directly to inequality constraints, so that slack variables are not needed.

5.3.2 Avoiding unnecessary local optimizations

The most computationally expensive steps in the sBB algorithms are typically the calls to the local optimization procedures to find lower and upper bounds to the problem at hand. This normally involves the numerical solution of a general non-convex NLP, which can be relatively expensive. If a good lower or upper bound can be found for a region without resorting to the local optimization procedure, then it should be used without question.

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 were present in the original NLP ("original variables") and those that were added by the standardization procedure ("added variables", cf. section 5.2.2.1).

5.3.2.1 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.

5.3.2.2 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. 5.1).

5.4 The $oo\mathcal{OPS}$ software framework for optimization

Any computer implementation of general-purpose spatial Branch-and-Bound algorithms for global optimization requires a large amount of information on the problem being solved. This includes:

- numerical information of the type required by local optimization solvers, e.g. values of the objective function and the residuals of the constraints for given values of the problem variables;

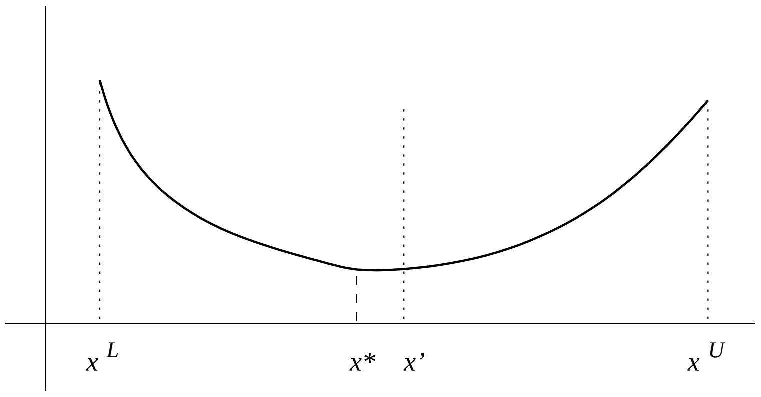


Figure 5.1: If the locally optimal solution in $[x^L, x^U]$ has already been determined to be at x^*, solving in $[x^L, x']$ is unnecessary.

- structural information, e.g. regarding the sparsity pattern of the constraints; this is essential not only for improving the code efficiency (a feature that is also shared with local optimization codes), but also in implementing advanced techniques such as the model reformulation algorithm of chapter 3;
- symbolic information on the objective function and constraints, needed for implementing general reformulation and/or convexification techniques.

Most implementations of sBB-type algorithms in existence today have been either standalone software codes (e.g. the αBB system [4, 5]) with their own language for defining optimization problems, or have been embedded within existing modelling tools which provide them with the necessary information; for example, Smith's [114] code was embedded in the gPROMS modelling tool [30] while the BARON code [102] has recently been made available within GAMS [22]. Whilst these developments are undeniably useful from the immediate practical point of view, the wider dissemination of global optimization technology requires a different approach which allows the software to be directly embedded within larger software systems such as mathematical modelling tools, domain-specific optimization codes (e.g. for pooling and blending) and so on.

The above considerations provided the motivation for the development of the $oo\mathcal{OPS}$ (object-oriented OPtimization System) system in the context of the work described in this thesis. $oo\mathcal{OPS}$ is a comprehensive library of callable procedures for the definition, manipulation and solution of large, sparse nonlinear programming problems[2].

$oo\mathcal{OPS}$ is completely coded in C++ [125], [103]. The C++ language is object-oriented but low-level enough to leave memory management to the programmer; C++ compilers are very efficient at optimizing the code and very portable across different architectures.

$oo\mathcal{OPS}$ is described in detail in Appendix A of this thesis. The rest of this chapter provides an overview of its design and usage.

5.4.1 Main features of $oo\mathcal{OPS}$

Rather than being a stand-alone code, $oo\mathcal{OPS}$ is designed to provide a number of high-level services to a client software code (e.g. written in C++, C or FORTRAN) via an Application Programming Interface (API).

A complete description of $oo\mathcal{OPS}$ is provided in Appendix A of this thesis. Its main features include the following:

1. $oo\mathcal{OPS}$ allows its client codes to construct and, if necessary, subsequently modify large-scale NLPs involving large sets of variables and linear and nonlinear constraints.
2. The construction of complex problems is facilitated by recognizing that, in most practical applications, variables and constraints can be categorised into relatively small numbers of sets which can be defined generically. Thus, $oo\mathcal{OPS}$ allows variables and constraints to be defined in a "structured" fashion as multi-dimensional arrays with an arbitrary number of dimensions.
3. The construction of an NLP in $oo\mathcal{OPS}$ is done in a symbolic manner. Thus, each nonlinear expression occurring in the objective function and/or the constraints is built by the client issuing a sequence of calls (i.e. on a term-by-term, factor-by-factor basis etc.). Consequently, $oo\mathcal{OPS}$ is fully aware at all times of the symbolic form of any NLP within it and can supply this information to its clients.
4. In order to facilitate the operation of numerical optimization solvers within the $oo\mathcal{OPS}$ framework, $oo\mathcal{OPS}$ automatically derives and makes available to its clients a "flat" form

[2]In fact, $oo\mathcal{OPS}$ has been designed and implemented to deal with mixed integer linear and nonlinear optimization problems. However, for the purposes of this chapter, we will refer only to its NLP capabilities.

of any NLP constructed in terms of structured variables and constraints. In this flat form, all variables and constraints are collected in two one-dimensional vectors (arrays).

5. The standard form (5.3) of an NLP is required by some sBB algorithms, such as the one by Smith which forms the basis of our work. It may also be useful in other contexts as it allows any NLP involving arbitrarily complex objective functions and/or constraints to be represented in terms of simple data structures, namely (cf. Section 5.2.2.1):

   - the index obj of the objective function in the vector of the problem variables;
   - the linear constraint matrix A, and the lower and upper bounds of the linear constraints, l and u respectively;
   - the triplets $\mathcal{M}$, $\mathcal{D}$, $\mathcal{P}$ and $\mathcal{U}$;
   - the values of the problem variables x and their lower and upper bounds x^L and x^U.

   For these reasons, and as has already been mentioned (cf. Section 5.4.1), *oo*$\mathcal{OPS}$ automatically derives the standard form for any NLP represented within it and makes it available to its clients.

6. *oo*$\mathcal{OPS}$ allows a client to manipulate any number of NLPs simultaneously; this is important for supporting applications which require iterating between two or more optimization problems, with information derived from the solution of one of these NLPS being used to define or modify one or more of the others.

7. *oo*$\mathcal{OPS}$ allows its clients to evaluate the objective function and constraints of any NLP held within it. Moreover, it automatically derives, and makes available to its clients, structural (e.g. sparsity pattern) and symbolic (e.g. exact partial derivatives[3]) information on these NLPs. An algorithm for fast evaluation of symbolic expressions has been devised and implemented [71].

8. *oo*$\mathcal{OPS}$ provides its clients a uniform interface to diverse LP/NLP/MILP/MINLP solvers from a variety of sources and coded in a variety of programming languages. These solvers operate on NLPs defined within *oo*$\mathcal{OPS}$.

[3]*oo*$\mathcal{OPS}$ derives partial derivatives in closed analytic form using symbolic differentiation.

5.4.2 Object classes in $oo\mathcal{OPS}$

In order to deliver the above functionality, $oo\mathcal{OPS}$ recognizes four major classes of objects, each with its own client interface.

1. The `ops` object.
   An `ops` object is a software representation of an NLP problem. The corresponding interface:

   - allows NLP objects to be constructed and modified in a structured manner (cf. Section 5.4.1;
   - provides access to all numerical and symbolic information pertaining to the NLP in structured, flat (unstructured) and standard forms.

2. The `opssystem` object.
   This represents the combination of an NLP `ops` object with a specific solver code (e.g. SNOPT [43]). The corresponding client interface:

   - allows the behaviour of the solver to be configured via the specification of any algorithmic parameters that the solver may support.
   - permits the solution of the NLP.

3. The `opssolvermanager` object.
   This corresponds to a particular NLP solver (e.g. SNOPT [43]). The primary function of the corresponding client interface is to receive an `ops` object representing a particular NLP and to return an `opssystem` (see above) that represents the combination of this NLP with the solver code.

4. The `convexifiermanager` object.
   This embeds a code which produces a convex relaxation of a given non-convex NLP. More specifically, given an `ops` object representing a particular NLP, it creates another `ops` object describing a convex relaxation of the NLP. The corresponding client interface also allows the "on-the-fly" updating of the convex relaxation whenever the ranges of the variables change.

5.4.3 Typical usage scenario for $oo\mathcal{OPS}$

A typical client application of $oo\mathcal{OPS}$ makes use of the object classes described above to formulate and solve an optimization problem. This involves a number of steps:

1. Construct an `ops` object describing the NLP to be solved. This is done by creating one or more structured variables and then introducing an objective function and constraints defined in terms of these variables.
2. Create an `opssolvermanager` for the particular NLP solver code that is to be used.
3. Configure the solver's behaviour by setting appropriate values for its algorithmic parameters (e.g. convergence tolerances, maximum number of iterations etc.); this is done by calling appropriate methods provided by the `opssolvermanager`'s interface.
4. Create an `opssystem` by passing the `ops` object created at step 1 to the `opssolvermanager` created at step 2.
5. Solve the problem by invoking the `opssystem`'s `Solve()` method. This will involve the solver code interacting directly with the `ops` object in a (usually large) number of steps. For example, a local NLP solver may typically issue the following requests to an `ops` object:
   (a) Obtain the numbers of variables and constraints in the NLP to be solved, as well as other structural information such as the nature of each constraint (e.g. linear/nonlinear, constraint bounds etc.), the sparsity pattern and so on.
   (b) Obtain the initial values of all the NLP problem variables.
   (c) In an iterative loop,
      i. evaluate the objective function and the constraints at the current values of the problem variables;
      ii. (possibly) evaluate the partial derivatives of the objective function and the constraints at the current values of the problem variables;
      iii. modify the values of the problem variables;

      until convergence is obtained or some other termination criterion is satisfied (e.g. the maximum number of allowable iterations is reached).
6. Obtain the NLP's solution by invoking the `ops`'s method for querying variable values.

5.4.4 Solver components for use within $oo\mathcal{OPS}$

The interactions at steps 2-5 of the typical usage scenario presented in Section 5.4.3 above impose certain requirements on every numerical solver designed for use in the context of $oo\mathcal{OPS}$. More specifically:

- each such solver must be implemented as a software component that exposes an `opssolvermanager` interface (cf. step 2 of the interaction);
- the solver's `opssolvermanager` interface must provide a set of methods for accessing and modifying the algorithmic parameters associated with this solver (cf. step 3 of the interaction);
- the solver must obtain all information regarding the NLP to be solved by calling appropriate methods in the NLP's `ops` object (cf. step 5 of the interaction);
- at the end of the numerical solution, the solver must place the final (converged) values of the solution variables back in the `ops` object describing the original NLP, from where they can be recovered at any later stage by the client (cf. step 6 of the interaction).

Albeit apparently burdensome, the above requirements are, in fact, relatively easy to satisfy. Some solver software components can be designed and implemented specifically for $oo\mathcal{OPS}$ while others may be pre-existing pieces of software which are "wrapped" in an $oo\mathcal{OPS}$-compliant interface. In either case, all solver modules consist of the actual solver code, an `opssystem` interface and an `opssolvermanager` interface. Template files are provided with $oo\mathcal{OPS}$ in order to facilitate the construction of these interfaces.

In practice, solver components are implemented as dynamic link libraries which expose a single function called `NewMINLPSolverManager`. Application clients typically apply dynamic linking to these solver components, and then invoke `NewMINLPSolverManager` to create a new solver manager object and obtain an interface to it.

As an illustration of the implementation of solver components within the $oo\mathcal{OPS}$ framework, we have wrapped the SNOPT code [43] for local NLP optimization. However, of more interest to this thesis is the implementation of a global optimization code based on a variation of Smith's sBB algorithm; this is described in Section 5.5.

5.5 An sBB solver for the $oo\mathcal{OPS}$ framework

The sBB algorithm of Smith [114, 116] for global optimization (cf. Section 5.2), with the minor improvements described in Section 5.3, has been implemented for use within the $oo\mathcal{OPS}$ framework.

5.5.1 Overview of the sBB code

The sBB implementation makes use of two sub-solvers: a local NLP solver for obtaining upper bounds by solving the original NLP in each region; and an optimization solver for obtaining lower bounds by solving the convex relaxation of the original NLP in each region. Each of these sub-solvers is itself implemented as an $oo\mathcal{OPS}$-compliant optimization solver. In addition, the sBB code makes use of a convexifier, a software component which, given an `ops` object, constructs another `ops` object that represents a convex relaxation of the former. This is described in more detail in Section 5.5.2.

More specifically, given an `ops` object describing a non-convex NLP, our sBB solver executes the following sequence of operations in the context of step 5 of the generic algorithm outlined in Section 5.4.3:

1. Create an `opssolvermanager` for the local solver which will be used to solve the upper bounding problem.
2. Pass the original `ops` object to the above local `opssolvermanager` to create an upper bounding `opssystem`.
3. Create a `convexifiermanager`.
4. Create a new `ops` object representing a convex relaxation of the original `ops` object by passing the latter to the above `convexifiermanager`.
5. Create an `opssolvermanager` for the optimization solver to be used for the solution of the lower bounding problem.
6. Create a lower bounding `opssystem` by passing the `ops` object constructed at step 4 to the `opssolvermanager` constructed at step 5.
7. During the branch-and-bound search:

(a) repeatedly call the `Solve` methods of the `opssystem`s constructed at steps 2 and 6 to solve the upper and lower bounding problems respectively.

(b) On changing the variable ranges during branching, update the lower bounding problem by invoking the `UpdateConvexVarBounds()` method in the `convexifiermanager`.

8. De-allocate all objects created by the global solver code.

As can be seen, the above procedure takes advantage of *oo*$\mathcal{OPS}$'s capabilities to simultaneously manipulate two `ops` objects respectively describing the original NLP problem and its convex relaxation.

Our current sBB implementation uses SNOPT [43] for solving the upper bounding problem, and the CPLEX [60] linear programming code as the solver for the convex relaxation[4]. It is worth noting, however, that, in principle, *any oo*$\mathcal{OPS}$-compliant local NLP solver can be used as a sub-solver within our sBB code.

5.5.2 The convexifier

In addition to generic numerical optimization solvers, the *oo*$\mathcal{OPS}$ framework recognizes another generic type of software component, that of convexifiers. These simply take one `ops` object representing a non-convex NLP, and return another `ops` object representing a convex relaxation of the original NLP. *oo*$\mathcal{OPS}$ does not prescribe how this relaxation is to be formed or indeed how tight it should be; all it does is to define a standard software interface for convexifiers or, more precisely, for `convexifiermanager` objects (cf. Section 5.4.2). This allows convexifiers based on different approaches and/or originating from different sources to be used within *oo*$\mathcal{OPS}$.

For our purposes, we have implemented a convexifier based on the approach described in Section 5.2.2. This makes use of the standard form of the NLP which is automatically constructed and made available by the `ops` object (cf. Section 5.4.1). In our current implementation, we use only linear relaxations (cf. Section 5.2.2.2), and consequently, the `ops` object constructed by the `convexifiermanager` is actually a cut-down version of the full `ops`

[4]The use of CPLEX in this context is possible because the convexifier implemented and used in the current code constructs a *linear* convex relaxation of the non-convex NLP (see Section 5.5.2).

class which only offers methods for manipulating the linear problem in flat (unstructured) form.

As has been seen in Section 5.2.2.2, the convex relaxation of an NLP involves the bounds x^L and x^U of the problem variables x within the current region. For the efficient operation of the sBB algorithm, it is important to be able to efficiently update the convex relaxation by taking account of changes in these bounds and not to have to regenerate it from the beginning. To achieve this, our convexifier implementation creates dependency links between the constraints in the relaxed problems and the variable ranges. This facilitates the "on-the-fly" updating of the convex relaxation whenever the ranges of any variable(s) change (i.e. as the sBB algorithm moves from considering one region to another).

5.5.3 Storing the list of regions

In abstract terms, a region to be considered by the sBB algorithm is a hypercube in the Euclidean space characterized by a list of n variable ranges where n is the total number of variables in the NLP. Thus, the memory requirements for storing each region would appear to be $2n$. Furthermore, during branching, we have to copy $2n$ memory units from the original region to each of the two new ones. However, this is wasteful since branching always reduces the range of just one variable while the ranges of all other variables remain unchanged.

In view of the above, we prefer to store regions in a tree structure, each node of which contains:

- the branch variable which led to the region's creation and its range;
- a pointer to the parent region;
- the objective function lower and upper bounds for this region;
- a flag that signals whether an upper bound is already available for the region prior to calculation (cf. Section 5.3.2).

The top region of the tree does not have any of the above information.

Starting from any node in the tree, we can derive the complete set of variable bounds for the corresponding region by ascending the tree via the node's parent. More specifically, given a particular node in the tree,

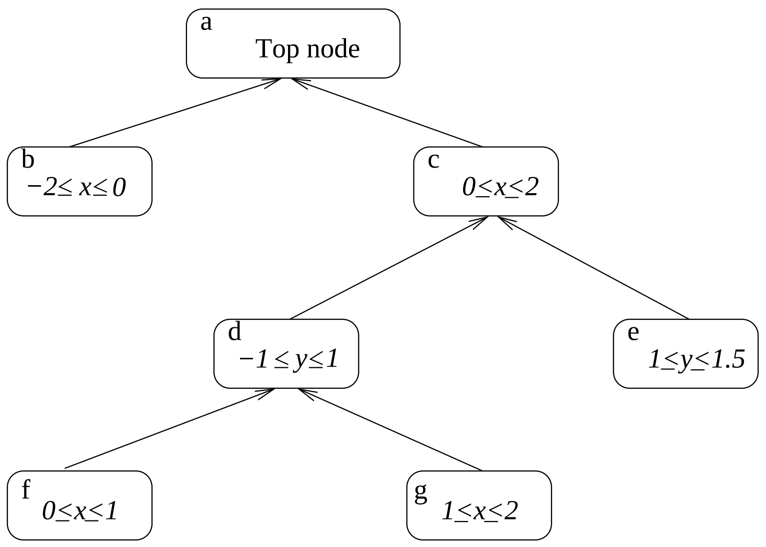


Figure 5.2: The region tree.

1. We label all problem variables as "unmarked" and initialize their ranges to be those in the original NLP.
2. If the branch variable in the current node is still unmarked, we:
   (a) modify its range to that indicated by the current node;
   (b) label this variable as "marked".
3. If the current node is not the top node, we apply the above procedure to the parent of the current node; otherwise we terminate.

As an example, consider the region tree shown in Fig. 5.2 constructed at some stage during the solution of an NLP involving three variables, namely $x \in [-2, 2]$, $y \in [-1, 1.5]$ and $z \in [-1, 1]$. For ease of reference, the nodes of the tree are labeled as $a \dots g$; the branch variable which led to the node's creation, and its range within this node, are also shown. Now suppose we wish to establish the ranges of all variables for node g. This can be achieved as follows:

1. We label all three variables x, y, z as unmarked and set their ranges to those in the original NLP, i.e. $x \in [-2, 2]$, $y \in [-1, 1.5]$ and $z \in [-1, 1]$.
2. Starting from node g, since the branch variable x is unmarked, we modify its range to $[1, 2]$ and label it as marked. The variable ranges are now $x \in [1, 2]$, $y \in [-1, 1.5]$ and $z \in [-1, 1]$.

3. We now move to d which is the parent node of g. Since the branch variable y is unmarked, we modify its range to $[-1, 1]$ and label it as marked. The variable ranges are now $x \in [1, 2]$, $y \in [-1, 1]$ and $z \in [-1, 1]$.

4. We now move to c which is the parent node of d. Since the branch variable x is already marked, we do not make any modifications to the variable ranges at this node.

5. We now move to a which is the parent node of c. Since this is the top node, we terminate.

Overall, the above sequence of steps has determined that the variable ranges for the region corresponding to node g are $x \in [1, 2]$, $y \in [-1, 1]$ and $z \in [-1, 1]$.

5.6 Concluding remarks

In this chapter, we presented an outline of the symbolic reformulation spatial Branch-and-Bound (sBB) algorithm by Smith [114, 116] and proposed some minor improvements to it. We also considered the software implementation of general sBB algorithms, and the requirements that this imposes in terms of numeric, structural and symbolic information on the NLP being solved. Our analysis led us to the development of *oo*$\mathcal{OPS}$, a general software framework for optimization that can support both local and global optimization solvers.

A key concept in the design of *oo*$\mathcal{OPS}$ is the separation of the NLP problem being solved from the optimization solver itself; the former is described by a software object `ops` while the latter is implemented as a software component exposing an `opssolvermanager` object interface. The combination of a solver manager with `ops` object leads to the creation of an `opssystem` object which can be solved by the invocation of an appropriate method. We borrowed all of these ideas from the design of standardized software components for the solution of sets of nonlinear algebraic and mixed differential-algebraic equations in the CAPE-OPEN project, an international initiative for the standardization of process engineering software[5]. Having extended these concepts to the solution of nonlinear optimization problems, we have contributed some of them back to the CAPE-OPEN initiative to form the basis of a new standard[6] for optimization solvers [88].

In order to test the generality and applicability of the *oo*$\mathcal{OPS}$ framework, we successfully

[5]See `http://www.colan.org`.

[6]Note, however, that this standard does not yet support *global* optimization solvers of the type considered here.

implemented an sBB algorithm as an $oo\mathcal{OPS}$-compliant solver code, together with all its associated sub-solvers as well as a convexifier. It is worth pointing out, however, that our current sBB implementation is rather basic without many of the sophisticated implementation details that accelerate the performance of such algorithms (e.g. good range-reduction techniques, improved branching procedures etc.). Nevertheless, $oo\mathcal{OPS}$ provides an open architecture within which more sophisticated implementations may be embedded in the future.

Chapter 6

Concluding remarks

This thesis has considered several topics related to the global solution of nonconvex nonlinear programming (NLP) problems. A major theme throughout our research has been the importance of the mathematical formulation of a given engineering optimization problem. The work presented in Chapter 3 was initially motivated by an observation by Smith [114], page 270 that the addition of a relatively small number of constraints to the material balances of a distillation column model could result in hugely improved computational performance of his spatial branch-and-bound algorithm. Smith derived these constraints by multiplying linear mole fraction normalization constraints by total flowrates; albeit redundant with respect to the original column equations, they were not redundant with respect to the convex relaxation of these equations, and in fact produced a tightening of the latter.

Based on the above observation, we sought to generalize the approach to produce such constraints for general NLPs involving bilinear terms. In the event, we discovered that these constraints were not only a means of tightening the convex relaxation, but in fact allowed the exact reformulation of the *original* NLP to replace a number of bilinear terms by linear constraints. Thus, the resulting formulation is less nonlinear than the original, as well as having a tighter convex relaxation.

Another important aspect of the work presented in Chapter 3 is the use of graph theoretical algorithms for limiting the number of redundant constraints generated to those that are guaranteed to produce an improved mathematical formulation. Smith's empirical observation (mentioned above) was that it was beneficial to multiply a linear constraint by a problem variable if this did not introduce any new bilinear term. In this thesis, we showed that even multiplications which do introduce some new bilinear terms may be beneficial provided the number of such terms is smaller than the number of constraints involved in these multiplications; and that graph theoretical algorithms for the identification of dilations in bipartite graphs are ideally suited for locating such situations even for very large problems. Overall, the proposed reformulation techniques are much more selective in generating new constraints and variables than earlier techniques such as RLT [111], which makes them more applicable to NLPs of non-trivial size.

A second major focus of the thesis has been the derivation of convex relaxations for monomial terms of odd degree when the variable range includes zero. Whereas the idea of deriving the convex/concave envelopes of piecewise convex and concave functions is geometrically quite obvious and already present in the literature [80, 126], its mention was sporadic and its theoretical analysis superficial. In Chapter 4, we have given an in-depth analysis of these envelopes, derived tight linear relaxations, and compared these with other existing techniques. The results of the latter comparison have been quite favourable. Overall, monomial terms of odd degree do appear in many practical applications (albeit certainly not as frequently as other kinds of terms such as bilinear products); we believe that the work presented in this thesis fills an important gap in the handling of these terms by spatial branch-and-bound methods.

Finally, our work has considered the software implications and requirements of implementing spatial branch-and-bound algorithms. Although these are often thought of and referred to as "numerical" solution codes, in fact they pose many more demands in terms of the type and amount of information that they need than, say, conventional local optimization codes. In particular, automatic reformulation techniques of the type discussed in this thesis require substantial symbolic information, and so does the construction of convex relaxations. Some of these issues were addressed by earlier workers such as Smith [114] and Adjiman et al. [4] who designed stand-alone software systems for global optimization based on spatial branch-and-bound algorithms. The $oo\mathcal{OPS}$ system described in Chapter 5 takes a different approach where the aim is to develop a generic software libraries comprising callable functions that can be embedded within larger software systems in order to provide a global optimization capability.

The $oo\mathcal{OPS}$ system developed as part of the work leading to this thesis contains advanced mechanisms for formulation of complex models (e.g. in terms of multidimensional variables and constraints), as well as implementations of automatic reformulation and convexification procedures. However, it includes only a rudimentary implementation of the actual spatial branch-and-bound algorithm, without many of the sophisticated implementation details that accelerate the performance of such algorithms. In view of this, it is interesting to note that the numerical results reported in Section 3.5 indicate that, at least for the class of pooling and blending problems considered there, the algorithm’s performance was better than that of most of the algorithms reported in the literature; in fact, it was comparable to (albeit not quite as good as) that of the best code, namely BARON [102] which employs good range-reduction techniques, improved branching procedures and other advanced features. We believe that this is another illustration of the importance of good mathematical formulation. Of course, automatic reformulation techniques and advanced implementation features are largely complementary, and it is to be expected that even better performance could be obtained by combining them in a single code.

Bibliography

[1] N. Adhya, M. Tawarmalani, and N. Sahinidis. A Lagrangian approach to the pooling problem. *Industrial and Engineering Chemistry Research*, 38:1956–1972, 1999.

[2] C. Adjiman. *Global Optimization Techniques for Process Systems Engineering*. PhD thesis, Princeton University, June 1998.

[3] C. Adjiman, I. Androulakis, and C. Floudas. Global optimization of MINLP problems in process synthesis and design. *Computers & Chemical Engineering*, 21:S445–S450, 1997.

[4] C. Adjiman, S. Dallwig, C. Floudas, and A. Neumaier. A global optimization method, αBB, for general twice-differentiable constrained NLPs: I. Theoretical advances. *Computers & Chemical Engineering*, 22:1137–1158, 1998.

[5] C. Adjiman, I. Androulakis, and C. Floudas. A global optimization method, αBB, for general twice-differentiable constrained NLPs: II. Implementation and computational results. *Computers & Chemical Engineering*, 22:1159–1179, 1998.

[6] C. Adjiman, I. Androulakis, C. Maranas, and C. Floudas. A global optimization method, αBB, for process design. *Computers & Chemical Engineering*, 20:S419–S424, 1996.

[7] C. Adjiman and C. Floudas. Rigorous convex underestimators for general twice-differentiable problems. *Journal of Global Optimization*, 9:23–40, 1996.

[8] C. Adjiman, C. Schweiger, and C. Floudas. Mixed-integer nonlinear optimization in process synthesis. In D. Du and P. (Eds.) Pardalos, editors, *Handbook of Combinatorial Optimization, vol. I*, volume 1, pages 1–76, Dordrecht, 1998. Kluwer Academic Publishers.

[9] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.

[10] F. Al-Khayyal and J. Falk. Jointly constrained biconvex programming. *Mathematics of Operations Research*, 8:273–286, 1983.

[11] F. Al-Khayyal and H. Sherali. On finitely terminating branch-and-bound algorithms for some global optimization problems. *SIAM Journal of Optimization*, 10:1049–1057, 2000.

[12] R. Amarger, L. Biegler, and I. Grossmann. An automated modeling and reformulation system for design optimization. *Computers & Chemical Engineering*, 16:623–636, 1992.

[13] I. Androulakis, C. Maranas, and C. Floudas. αBB: A global optimization method for general constrained nonconvex problems. *Journal of Global Optimization*, 7:337–363, 1995.

[14] M. Bazaraa, H. Sherali, and C. Shetty. *Nonlinear Programming: Theory and Algorithms*. Wiley, Chichester, second edition, 1993.

[15] E. Beale and J. Tomlin. Special facilities in a general mathematical programming system for nonconvex problems using ordered sets of variables. In J. Laurence, editor, *Proceedings of the Fifth International Conference on Operational Research*, pages 447–454, London, 1970. Tavistock Publications.

[16] J. Beasley. Heuristic algorithms for the unconstrained binary quadratic programming problem. `http://mscmga.ms.ic.ac.uk/jeb/bqp.pdf`, 1998.

[17] A. Ben-Tal, G. Eiger, and V. Gershovitz. Global minimization by reducing the duality gap. *Mathematical Programming*, 63:193–212, 1994.

[18] H. Benson. Concave minimization: Theory, applications and algorithms. In Horst and Pardalos [57], pages 43–148.

[19] J. Bjorkqvist and T. Westerlund. Automated reformulation of disjunctive constraints in MINLP optimization. *Computers and Chemical Engineering*, 23:S11–S14, 1999.

[20] I. Bomze. On standard quadratic optimization problems. *Journal of Global Optimization*, 13:369–387, 1998.

[21] I. Bomze, M. Budinich, P. Pardalos, and M. Pelillo. The maximum clique problem. In Du and Pardalos [28], pages 1–74.

[22] A. Brook, D. Kendrick, and A. Meeraus. Gams, a user's guide. *ACM SIGNUM Newsletter*, 23:10–11, 1988.

[23] R. Chelouah and P. Siarry. Tabu search applied to global optimization. *European Journal of Operational Research*, 123:256–270, 2000.

[24] K. Ciesielski. *Set Theory for the Working Mathematician*. Cambridge University Press, Cambridge, 1997.

[25] A. Colorni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. In F. Varela and P. Bourgine, editors, *Proceedings of the European Conference on Artificial Life*, pages 134–142, Amsterdam, 1991. ECAL, Paris, France, Elsevier.

[26] D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties and Algorithms*. Springer-Verlag, Berlin, second edition, 1997.

[27] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ, 1963.

[28] D. Du and P. Pardalos, editors. *Handbook of Combinatorial Optimization*, volume supp. A. Kluwer Academic Publishers, Dordrecht, 1998.

[29] I. Duff. On algorithms for obtaining a maximum transversal. *ACM Trans. Math. Software*, 7:315–330, 1981.

[30] Process Systems Enterprise. *gPROMS v2.2 Introductory User Guide*. Process Systems Enterprise, Ltd., London, UK, 2003.

[31] T. Epperly. *Global Optimization of Nonconvex Nonlinear Programs using Parallel Branch and Bound*. PhD thesis, University of Winsconsin – Madison, 1995.

[32] T. Epperly and E. Pistikopoulos. A reduced space branch and bound algorithm for global optimization. *Journal of Global Optimization*, 11:287:311, 1997.

[33] E. Eskow and R. Schnabel. Mathematical modeling of a parallel global optimization algorithm. *Parallel Computing*, 12:315–325, 1989.

[34] J. Falk and R. Soland. An algorithm for separable nonconvex programming problems. *Management Science*, 15:550–569, 1969.

[35] A. Fiacco and G. McCormick. *Nonlinear Programming: Sequential Unconstrained Minimization Techniques*. Wiley, New York, 1968.

[36] A. Fischer. New constrained optimization reformulation of complementarity problems. *Journal of Optimization Theory and Applications*, 99:481–507, 1998.

[37] R. Fischer. A new algorithm for generating space-filling curves. *Software Practice and Experience*, 16:5–12, 1986.

[38] R. Fletcher. *Practical Methods of Optimization*. Wiley, Chichester, second edition, 1991.

[39] C. Floudas. *Deterministic Global Optimization*. Kluwer Academic Publishers, Dordrecht, 2000.

[40] L. Foulds, D. Haughland, and K. Jornsten. A bilinear approach to the pooling problem. *Optimization*, 24:165–180, 1992.

[41] A. Frangioni. On a new class of bilevel programming problems and its use for reformulating mixed-integer problems. *European Journal of Operations Research*, 82:615–646, 1995.

[42] R. Ge. A parallel global optimization algorithm for rational separable-factorable functions. *Applied Mathematics and Computation*, 32:61–72, 1989.

[43] P. Gill. *User's Guide for SNOPT 5.3*. Systems Optimization Laboratory, Department of EESOR, Stanford University, California, 1999.

[44] B. Goertzel. Global optimisation with space-filling curves. *Applied Mathematics Letters*, 12:133–135, 1999.

[45] E. Goodaire and M. Parmenter. *Discrete Mathematics with Graph Theory*. Prentice-Hall, London, 1998.

[46] I.E. Grossmann, editor. *Global Optimization in Engineering Design*. Kluwer Academic Publishers, Dordrecht, 1996.

[47] K. Hägglöf, P. Lindberg, and L. Svensson. Computing global minima to polynomial optimization problems using gröbner bases. *Journal of Global Optimization*, 7:115:125, 1995.

[48] E. Hansen. Global constrained optimization using interval analysis. In K. Nickel, editor, *Interval Mathematics 1980*, pages 25–47, New York, 1980. Proceedings of the International Symposium, Freiburg, Academic Press.

[49] E. Hansen. *Global Optimization Using Interval Analysis*. Marcel Dekker, Inc., New York, 1992.

[50] P. Hansen, B. Jaumard, and S. Lu. Analytical approach to global optimization. *Comptes Rendus de L Academie Des Sciences Serie I-mathematique*, 306:29–32, 1988.

[51] F. Harary. *Graph Theory*. Addison-Wesley, Reading, MA, second edition, 1971.

[52] M. Hirafuji and S. Hagan. A global optimization algorithm based on the process of evolution in complex biological systems. *Computers and Electronics in Agriculture*, 29:125–134, 2000.

[53] R. Horst. A general-class of branch-and-bound methods in global optimization with some new approaches for concave minimization. *Journal of Optimization Theory and Applications*, 51:271–291, 1986.

[54] R. Horst. Deterministic global optimization with partition sets whose feasibility is not known: Application to concave minimization, reverse convex constraints, d.c. programming and lipschitzian optimization. *Journal of Optimization Theory and Applications*, 58:11–37, 1988.

[55] R. Horst. On consistency of bounding operations in deterministic global optimization. *Journal of Optimization Theory and Applications*, 61:143–146, 1989.

[56] R. Horst, J. Devries, and N. Thoai. On finding new vertices and redundant constraints in cutting plane algorithms for global optimization. *Operations Research Letters*, 7:85–90, 1988.

[57] R. Horst and P. Pardalos, editors. *Handbook of Global Optimization*, volume 1. Kluwer Academic Publishers, Dordrecht, 1995.

[58] R. Horst and H. Tuy. *Global Optimization: Deterministic Approaches*. Springer-Verlag, Berlin, first edition, 1990.

[59] R. Horst and H. Tuy. *Global Optimization: Deterministic Approaches*. Springer-Verlag, Berlin, third edition, 1996.

[60] ILOG. *ILOG CPLEX 8.0 User's Manual*. ILOG S.A., Gentilly, France, 2002.

[61] P. Kesavan and P. Barton. Generalized branch-and-cut framework for mixed-integer nonlinear optimization problems. *Computers and Chemical Engineering*, 24:1361–1366, 2000.

[62] M. Kline. *Mathematical Thought from Ancient to Modern Times*. Oxford University Press, Oxford, 1972.

[63] G. Kocis and I. Grossmann. Global optimization of nonconvex mixed-integer nonlinear-programming (MINLP) problems in process synthesis. *Industrial & Engineering Chemistry Research*, 27:1407–1421, 1988.

[64] B. Korte and J. Vygen. *Combinatorial Optimization, Theory and Algorithms*. Springer-Verlag, Berlin, 2000.

[65] K. Kunen. *Set Theory*. North Holland, Amsterdam, 1980.

[66] J. Lagrange. *Théorie des fonctions analytiques*. Impr. de la République, Paris, 1797.

[67] B. Lamar. A method for converting a class of univariate functions into d.c. functions. *Journal of Global Optimization*, 15:55:71, 1999.

[68] Y. Lee and B. Berne. Global optimization: Quantum thermal annealing with path integral monte carlo. *Journal of Physical Chemistry A*, 104:86–95, 2000.

[69] D. Li and X. Sun. Local convexification of the lagrangian function in nonconvex optimization. *Journal of Optimization Theory and Applications*, 104:109–120, 2000.

[70] D. Li and X. Sun. Convexification and existence of a saddle point in a pth-power reformulation for nonconvex constrained optimization. *Nonlinear Analysis*, 47:5611–5622, 2001.

[71] L. Liberti. Performance comparison of function evaluation methods. *Progress in Computer Science Research*, accepted.

[72] L. Liberti and C. Pantelides. Convex envelopes of monomials of odd degree. *Journal of Global Optimization*, 25:157–168, 2003.

[73] L. Liberti, P. Tsiakis, B. Keeping, and C. Pantelides. *oo*$\mathcal{OPS}$. Centre for Process Systems Engineering, Chemical Engineering Department, Imperial College, London, UK, 1.24 edition, 2001.

[74] R. Luus and T. Jaakola. Optimization by direct search and systematic reduction of the size of the search region. *AIChE Journal*, 19:760–766, 1973.

[75] C. Maranas and C. Floudas. Finding all solutions to nonlinearly constrained systems of equations. *Journal of Global Optimization*, 7:143–182, 1995.

[76] H. Margenau and G. Murphy. *The Mathematics of Physics and Chemistry*. Van Nostrand, Princeton, NJ, second edition, 1956.

[77] S. Masri, G. Bekey, and F. Safford. A global optimization algorithm using adaptive random search. *Applied Mathematics and Computation*, 7:353–375, 1980.

[78] M. Mathur, S. Karale, S. Priye, V.K. Jayaraman, and B. Kulkarni. Ant colony approach to continuous function optimization. *Industrial and Engineering Chemistry Research*, 39:3814–3822, 2000.

[79] G. McCormick. Converting general nonlinear programming problems to separable nonlinear programming problems. *Technical Paper T-267*, Program in Logistics, The George Washington University, Washington D.C., 1972.

[80] G. McCormick. Computability of global solutions to factorable nonconvex programs: Part I — Convex underestimating problems. *Mathematical Programming*, 10:146–175, 1976.

[81] C. Meewella and D. Mayne. An algorithm for global optimization of lipschitz continuous functions. *Journal of Optimization Theory and Applications*, 57:307–322, 1988.

[82] C. Meewella and D. Mayne. Efficient domain partitioning algorithms for global optimization of rational and lipschitz continuous functions. *Journal of Optimization Theory and Applications*, 61:247–270, 1989.

[83] W. Murray and K. Ng. Algorithms for global optimization and discrete problems based on methods for local optimization. In Pardalos and Romeijn [91], pages 87–113.

[84] A. O'Grady, I. Bogle, and E. Fraga. Interval analysis in automated design for bounded solutions. *Chemicke Zvesti*, 55:376–381, 2001.

[85] J.M. Ortega and W.C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables*. Academic Press, New York, 1970.

[86] J. Pang. Complementarity problems. In Horst and Pardalos [57], pages 271–338.

[87] C. Pantelides. The consistent initialization of differential-algebraic systems. *SIAM J. Sci. Stat. Comput.*, 9:213–231, 1988.

[88] C. Pantelides, L. Liberti, P. Tsiakis, and T. Crombie. Mixed integer linear/nonlinear programming interface specification. *Global Cape-Open Deliverable WP2.3-04*, 2002.

[89] P. Pardalos. Enumerative techniques for solving some nonconvex global optimization problems. *Or Spektrum*, 10:29–35, 1988.

[90] P. Pardalos. Parallel search algorithms in global optimization. *Applied Mathematics and Computation*, 29:219–229, 1989.

[91] P. Pardalos and H. Romeijn, editors. *Handbook of Global Optimization*, volume 2. Kluwer Academic Publishers, Dordrecht, 2002.

[92] P. Pardalos and J. Rosen. *Constrained Global Optimization: Algorithms and Applications*. Springer-Verlag, Berlin, 1987.

[93] P. Pardalos and G. Schnitger. Checking local optimality in constrained quadratic programming is np-hard. *Operations Research Letters*, 7:33–35, 1988.

[94] J. Pinter. Global optimization on convex sets. *Or Spektrum*, 8:197–202, 1986.

[95] R. Pörn, I. Harjunkoski, and T. Westerlund. Convexification of different classes of non-convex MINLP problems. *Computers and Chemical Engineering*, 23:439–448, 1999.

[96] S. Rajasekaran. On simulated annealing and nested annealing. *Journal of Global Optimization*, 16:43–56, 2000.

[97] H. Ratschek and J. Rokne. Efficiency of a global optimization algorithm. *Siam Journal On Numerical Analysis*, 24:1191–1201, 1987.

[98] R. Rockafellar. *Convex Analysis*. Princeton University Press, Princeton, NJ, 1970.

[99] H. Ryoo and N. Sahinidis. Global optimization of nonconvex NLPs and MINLPs with applications in process design. *Computers & Chemical Engineering*, 19:551–566, 1995.

[100] H. Ryoo and N. Sahinidis. A branch-and-reduce approach to global optimization. *Journal of Global Optimization*, 8:107–138, 1996.

[101] H. Ryoo and N. Sahinidis. Global optimization of multiplicative programs. *Journal of Global Optimization*, 26:387–418, 2003.

[102] N. Sahinidis. Baron: Branch and reduce optimization navigator: User's manual, version 4.0. *http://archimedes.scs.uiuc.edu/baron/manuse.pdf*, 1999.

[103] P. Salus, editor. *Object-Oriented Programming Languages*. MacMillan, Indianapolis, 1998.

[104] F. Schoen. Two-phase methods for global optimization. In Pardalos and Romeijn [91], pages 151–177.

[105] 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.

[106] J. Shectman and N. Sahinidis. A finite algorithm for global minimization of separable concave programs. *Journal of Global Optimization*, 12:1–36, 1998.

[107] H. Sherali. Global optimization of nonconvex polynomial programming problems having rational exponents. *Journal of Global Optimization*, 12:267–283, 1998.

[108] H. Sherali. Tight relaxations for nonconvex optimization problems using the reformulation-linearization/convexification technique (RLT). In Pardalos and Romeijn [91], pages 1–63.

[109] H. Sherali and W. Adams. A tight linearization and an algorithm for 0-1 quadratic programming problems. *Management Science*, 32:1274–1290, 1986.

[110] H. Sherali and W. Adams. *A Reformulation-Linearization Technique for Solving Discrete and Continuous Nonconvex Problems*. Kluwer Academic Publishers, Dodrecht, 1999.

[111] H. Sherali and A. Alameddine. A new reformulation-linearization technique for bilinear programming problems. *Journal of Global Optimization*, 2:379–410, 1992.

[112] H. Sherali, J. Smith, and W. Adams. Reduced first-level representations via the reformulation-linearization technique: Results, counterexamples, and computations. *Discrete Applied Mathematics*, 101:247–267, 2000.

[113] H. Sherali and H. Wang. Global optimization of nonconvex factorable programming problems. *Mathematical Programming*, A89:459–478, 2001.

[114] E. Smith. *On the Optimal Design of Continuous Processes*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, October 1996.

[115] E. Smith and C. Pantelides. Global optimisation of nonconvex MINLPs. *Computers and Chemical Engineering*, 21:S791–S796, 1997.

[116] E. Smith and C. Pantelides. A symbolic reformulation/spatial branch-and-bound algorithm for the global optimisation of nonconvex MINLPs. *Computers and Chemical Engineering*, 23:457–478, 1999.

[117] R. Soland. An algorithm for separable nonconvex programming problems ii: Nonconvex constraints. *Management Science*, 17:759–773, 1971.

[118] I. Stewart. *Galois Theory*. Chapman & Hall, London, second edition, 1989.

[119] 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.

[120] A. Strekalovsky. On global optimality conditions for d.c. programming problems. *Technical Paper, Irkutsk State University*, 1997.

[121] A. Strekalovsky. Extremal problems with d.c. constraints. *Computational Mathematics and Mathematical Physics*, 41:1742–1751, 2001.

[122] R. Strongin. On the convergence of an algorithm for finding a global extremum. *Engineering Cybernetics*, 11:549–555, 1973.

[123] R. Strongin. Algorithms for multi-extremal mathematical programming porblems employing the set of joint space-filling curves. *Journal of Global Optimization*, 2:357–378, 1992.

[124] R. Strongin. *Global Optimization with Non-Convex Constraints*. Kluwer Academic Publishers, Dodrecht, 2000.

[125] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, third edition, 1999.

[126] M. Tawarmalani and N. Sahinidis. Global optimization of mixed integer nonlinear programs: A theoretical and computational study. *Mathematical Programming (to appear)*, `http://archimedes.scs.uiuc.edu/papers/comp.pdf`, 1999.

[127] M. Tawarmalani and N. Sahinidis. Semidefinite relaxations of fractional programming via novel techniques for constructing convex envelopes of nonlinear functions. *Journal of Global Optimization*, 20:137–158, 2001.

[128] M. Tawarmalani and N. Sahinidis. Convex extensions and envelopes of semi-continuous functions. *Mathematical Programming*, 93:247–263, 2002.

[129] M. Tawarmalani and N. Sahinidis. Convexification and global optimization of the pooling problem. *Mathematical Programming (submitted)*, 2002.

[130] M. Tawarmalani and N. Sahinidis. Exact algorithms for global optimization of mixed-integer nonlinear programs. In Pardalos and Romeijn [91], pages 1–63.

[131] A. Törn and A. Žilinskas. *Global Optimization*. Springer-Verlag, Berlin, 1989.

[132] H. Tuy. D.c. optimization: Theory, methods and algorithms. In Horst and Pardalos [57], pages 149–216.

[133] H. Tuy. *Convex Analysis and Global Optimization*. Kluwer Academic Publishers, Dodrecht, 1998.

[134] H. Tuy and R. Horst. Convergence and restart in branch-and-bound algorithms for global optimization: Application to concave minimization and d.c. optimization problems. *Mathematical Programming*, 41:161–183, 1988.

[135] A. Žilinskas. Axiomatic characterization of a global optimization algorithm and investigation of its search strategy. *Operations Research Letters*, 4:35–39, 1985.

[136] J. Žilinskas and I. Bogle. Evaluation ranges of functions using balanced random interval arithmetic. *Informatica*, 14:403–416, 2003.

[137] R. Vaidyanathan and M. El-Halwagi. Global optimization of nonconvex MINLPs by interval analysis. In Grossmann [46], pages 175–193.

[138] V. Visweswaran and C. Floudas. New properties and computational improvement of the gop algorithm for problems with quadratic objective functions and constraints. *Journal of Global Optimization*, 3:439–462, 1993.

[139] V. Visweswaran and C. Floudas. New formulations and branching strategies for the gop algorithm. In Grossmann [46].

[140] B. Wah and T. Wang. Efficient and adaptive lagrange-multiplier methods for nonlinear continuous global optimization. *Journal of Global Optimization*, 14:1:25, 1999.

[141] X. Wang and T. Change. A multivariate global optimization using linear bounding functions. *Journal of Global Optimization*, 12:383–404, 1998.

[142] T. Westerlund and R. Pörn. Solving pseudo-convex mixed integer optimization problems by cutting plane techniques. *Optimization and Engineering*, 3:235–280, 2002.

[143] T. Westerlund, H. Skrifvars, I. Harjunkoski, and R. Pörn. An extended cutting plane method for a class of non-convex MINLP problems. *Computers & Chemical Engineering*, 22:357–365, 1998.

[144] L. Wolsey. *Integer Programming*. Wiley, New York, 1998.

[145] Y. Yao. Dynamic tunnelling algorithm for global optimization. *IEEE Transactions On Systems Man and Cybernetics*, 19:1222–1230, 1989.

[146] J. Zamora and I. Grossmann. A branch and contract algorithm for problems with concave univariate, bilinear and linear fractional terms. *Journal of Global Optimization*, 14:217:249, 1999.

Appendix A

$\mathit{oo}\mathcal{OPS}$ Reference Manual

A.1 Introduction

$\mathit{oo}\mathcal{OPS}$ is a library of C++ callable procedures for the definition, manipulation and solution of large, sparse mixed integer linear and nonlinear programming (MINLP) problems. In particular, $\mathit{oo}\mathcal{OPS}$:

- facilitates the definition of complex sets of constraints, reducing the required programming effort to a minimum;
- allows its client programs to create and manipulate simultaneously more than one MINLP;
- provides a common interface to diverse MINLP solvers to be used without any changes to client programs.

MINLPs are optimization problems whose objective function and constraints can, in general, contain nonlinear terms. The variables appearing in the objective function and constraints are generally restricted to lie between specified lower and upper bounds. Furthermore, some of these variables may be restricted to integer values only. The aim of the optimization is to determine values of the variables that minimize or maximize the objective function while satisfying the constraints and all other restrictions imposed on them.

A simple mathematical description of an MINLP can be written as:

[Flat MINLP]:

$$\min_x \phi(x)+c^T x \tag{A.1}$$
$$a \le f(x) + Ax \le b \tag{A.2}$$
$$x^l \le x \le x^u \tag{A.3}$$
$$x_i \in \mathbb{Z} \quad \forall i \in I \subseteq \{1,..,n\} \tag{A.4}$$

where $x, x^l, x^u, c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $A : \mathbb{R}^n \to \mathbb{R}^m$, $\phi : \mathbb{R}^n \to \mathbb{R}$ and $f : \mathbb{R}^n \to \mathbb{R}^m$. Thus, the variables x are characterised by an index $i = 1,..,n$; all constraints are expressed as inequalities of the form ≤ 0 and are indexed over the discrete domain $1,..,m$.

The above general formulation also embeds three special cases:

- *Mixed Integer Linear Programming (MILP) Problems.*
  In this case, $\phi(x) = 0$ and $f(x) = 0$.
- *Nonlinear Programming (NLP) Problems.*
  In this case, $I = \emptyset$.
- *Linear Programming (LP) Problems.*
  In this case, $\phi(x) = 0$, $f(x) = 0$ and $I = \emptyset$.

The *oo$\mathcal{OPS}$* software design aims to support the definition and solution of all these special cases, with minimal overhead being incurred because of the generality of the overall software.

Albeit quite general, the above MINLP form is not necessarily easy to construct and/or manipulate. A major reason for this is that the variables and constraints are maintained as unstructured “flat” lists or sets which may contain thousands of elements. On the other hand, most mathematical formulations of practical problems in terms of MINLPs are expressed in terms of a relatively small number of distinct sets of variables and constraints.

For example, in a typical network flow problem, a commonly occurring set of variables would be the flow of material from one node in the network to another, while a typical set of constraints would be the conservation of material arriving at, and leaving any node i in the network.

Of course, each such set of variables (or constraints) may have multiple elements, each corresponding to an individual variable (or constraint). An indexed set representation is usually employed for notational purposes. For instance, F_{ij} could represent the flow from node i to node j, while $\mathcal{C}_k$ could represent the set of conservation constraints:

$$\sum_{i \neq k} F_{ik} = \sum_{j \neq k} F_{kj}, \quad \forall \text{ node } k$$

In view of the above discussion, *oo$\mathcal{OPS}$* allows types of variables and constraints to be defined in a structured fashion as sets of an arbitrary number of dimensions.

A.2 Fundamental concepts

A.2.1 Object classes

oo$\mathcal{OPS}$ is designed as object-orientated software recognising two major classes of objects, each with its own interface:

1. The `ops` object
   An `ops` object is a software representation of a MINLP problem. The corresponding interface, discussed in detail in section A.4, provides the following functionality:
   - It allows MINLP objects to be constructed and modified in a structured manner.
   - It allows access to all information pertaining to the MINLP.
   - It provides the equivalent simple **[Flat MINLP]** form of the structured MINLP (see section A.1).
2. The `opssystem` object
   This is formed by the combination of an MINLP object (see above) with a code (“solver”) for the numerical solution of MINLP problems. The corresponding interface, discussed in detail in section A.5, 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 MINLP.

A.2.2 Multidimensional sets in *oo$\mathcal{OPS}$*

As detailed later in the document, an MINLP is characterised by a number of distinct multidimensional sets of variables and constraints. We note that:

- A multidimensional set is an ordered set whose elements can be accessed through a list of indices.
- The *dimensionality* of a multidimensional set is given by the length of the index lists (i.e. the number of dimensions) and the range of each index in the list.

- The *dimensionality size* of a multidimensional set is the list of its dimension sizes. More precisely, a multidimensional set having n dimensions, with index i_j ranging from i_J^L to i_j^U for each j between 1 and n, has *dimensionality size*:
$$(i_1^U - i_1^L + 1, \dots, i_n^U - i_n^L + 1).$$
- A scalar has dimensionality size (1).

For instance, consider a 3-dimensional variable set $X(i, j, k)$, with the first dimension varying from $i = 0$ to $i = 10$, the second dimension from $j = 1$ to $j = 5$, and the third dimension from $k = -1$ to $k = +1$. We can view this as a multidimensional set, the dimensionality of which is characterised by the number of dimensions (3, in this case) and the ranges of each dimension $(0 : 10, 1 : 5, -1 : 1)$. The dimensionality size is the list $(11, 5, 3)$ $(= 10 - 0 + 1, 5 - 1 + 1, 1 - (-1) + 1)$. The total number of elements of this set is 165 $(= 11 \times 5 \times 3)$.

oo$\mathcal{OPS}$ makes extensive use of the concept of *slices* as a convenient way of referring to subsets of multidimensional sets. In general:

- A slice $[s^L : s^U]$ of an N-dimensional set is *defined* by a pair of N-dimensional integer vectors $s_i^L, i = 1, .., N$ and $s_i^U, i = 1, .., N$ where $s_i^L \leq s_i^U$.
- The element at position $(k_1, k_2, .., k_N)$ of the set belongs to the slice $[s^L : s^U]$ if and only if:

$$s_i^L \leq k_i \leq s_i^U \ , \quad \forall\ i = 1, .., N \tag{A.5}$$

Here are some examples:

- The slice $[(2) : (5)]$ of a 1-dimensional set X denotes the elements $X_i, i = 2, 3, 4, 5$.
- The slice $[(2, 3) : (5, 4)]$ of a 2-dimensional set X denotes the elements $X_{ij}, i = 2, 3, 4, 5, j = 3, 4$.
- The slice $[(2, 3, 3) : (5, 3, 4)]$ of a 3-dimensional set X denotes the elements $X_{i3k}, i = 2, 3, 4, 5, k = 3, 4$.
- The slice $[(2, 3, 3, 4) : (2, 3, 3, 4)]$ of a 4-dimensional set X denotes the single element X_{2334}.

A.2.3 Constants, variables, constraints and objective function

Most optimization problems involve arithmetic constants. In *oo*$\mathcal{OPS}$, these can be organized in one or more multidimensional sets. A *constant set* is characterised by the following information:

- its name;
- the dimensionality of the set;
- the current value of each element of the set;

The variables to be determined by the optimization are also organized in one or more multidimensional sets. A *variable set* is characterised by the following information:

- its name;
- the type of all variables in this set (continuous or integer);
- the dimensionality of the set ;
- the current value of each element of the set;
- the upper and lower bounds of the value of each element of the set.

Similarly, the constraints in the optimization problem are also organized in one or more multidimensional sets.

A *constraint set* is characterised by the following information:

- its name;
- the lower and upper bound of the constraint;
- the dimensionality of the set;
- the variables occurring in the linear parts of these constraints and the corresponding coefficients (see note 1 in section A.2.5 below);
- the expression (see below) defining the nonlinear part of these constraints.

Most of the information characterising a set of constants, variables or constraints is common to *all* elements of the set. The *only* exceptions to this rule are:

- the values of elements of constant sets may differ from element to element;
- the values and bounds of elements of variable sets may differ from element to element;
- the constraint bounds of the constraint sets may differ from element to element.

Each MINLP object has a unique *objective function*. This is characterised by:

- the name of the objective function
- the type of the problem (minimization or maximization)
- the variables occurring in the linear part of the objective function and the corresponding coefficients (see note 1 in section A.2.5 below);
- the expression defining the nonlinear part of the objective function.

A.2.4 Nonlinear expressions and constants

A nonlinear *expression* is, in general, built hierarchically from the algebraic combination of variables, constants (see below) and other expressions by using operators and functions. Expressions are characterised by the following information:

- their name;
- their dimensionality size;
- whether they represent variables, constants or operators.

An expression represents a valid algebraic expression. The operators and functions that can be used to combine variables, constants and other expressions are given in the following table:

Name	Meaning
Binary Arithmetic Operators	
`sum`	addition
`difference`	subtraction
`product`	multiplication
`ratio`	division
`power`	exponentiation
Unary Arithmetic Operators	
`minus`	unary minus
Unary Transcendental Functions	
`log`	natural logarithm
`exp`	exponential
`sin`	sine
`cos`	cosine
`tan`	tangent
`cot`	cotangent
`sinh`	hyperbolic sine
`cosh`	hyperbolic cosine
`tanh`	hyperbolic tangent
`coth`	hyperbolic cotangent
`sqrt`	square root

In general, expressions are multidimensional sets with the special property that the range of each dimension starts from 1. The dimensionality size of an expression can be determined from the dimensionality sizes of its variables, constants and subexpressions and the type of its constituent operators, according to the rules given in the following tables:

Binary Arithmetic Operators		
dimensionality size of 1st operand	dimensionality size of 2nd operand	dimensionality size of result
$(s_1,\dots,s_n)$	$(s_1,\dots,s_n)$	$(s_1,\dots,s_n)$
(1)	$(s_1,\dots,s_n)$	$(s_1,\dots,s_n)$
$(s_1,\dots,s_n)$	(1)	$(s_1,\dots,s_n)$
otherwise		illegal

Unary Operators and Functions	
dimensionality size of operand	dimensionality size of result
$(s_1,\dots,s_n)$	$(s_1,\dots,s_n)$

A.2.5 Notes

1. *Variable occurrences in linear parts of constraints*
   An important part of the characterisation of the linear part of a constraint is the information on which variables actually occur in it, and the coefficients that multiply each such variable.

   In *oo$\mathcal{OPS}$*, such occurrences are specified as a variable slice occurring in a constraint slice with a given coefficient. This simply means that:

   Every element of the variable slice occurs in *each* element of the constraint slice, always with the *same* coefficient.

   Of course, the client constructing an MINLP object using the facilities provided by *oo$\mathcal{OPS}$* can add any number of such occurrences. Moreover, if any such specification involves an element of a variable vector that already appears[1] in one or more of the specified constraints, the later specification overrides the earlier one.

[1]as a result of earlier specifications

2. *Variable occurrences in linear part of objective function*
   An important part of the characterisation of the linear part of the objective function is the information on which variables actually occur in it, and the coefficients that multiply each such variable.

   In *oo$\mathcal{OPS}$* , such occurrences are specified as a variable slice occurring in the objective function with a given coefficient. This simply means that:

   > *Every* element of the variable slice occurs in the objective function always with the *same* coefficient.

   Of course, the client constructing an MINLP object using the facilities provided by *oo$\mathcal{OPS}$* can add any number of such occurrences. Moreover, if any such specification involves an element of a variable vector that already appears[2] in the objective function, the later specification overrides all earlier ones.

3. *Nonlinear expression occurrences in constraints*
   An important part of the characterisation of a constraint is its nonlinear part. In *oo$\mathcal{OPS}$*, this can be specified by assigning one or more expressions to different slices of the constraint. Each expression must either be a scalar or its dimensionality size must match that of the constraint slice to which it is assigned, in which case each element of the slice is set the corresponding the corresponding element of the expression.

   Note that, unlike linear variable occurrences, expression elements do not accumulate. If a new expression is assigned to a constraint element which has already been assigned an expression, the later assignment overrides the earlier one.

4. *Nonlinear expression occurrences in objective function*
   An important part of the characterisation of the objective function is its nonlinear part. In *oo$\mathcal{OPS}$*, this is specified as a (scalar) expression occurring in the objective function.

   Note that, unlike linear variable occurrences, expression elements do not accumulate. if a new expression is assigned to an objective function which has already been assigned an expression, the later assignment overrides the earlier one.

A.3 General software issues

A.3.1 Software constants

oo$\mathcal{OPS}$ defines two constants which should primarily be used for the specification of lower and upper bounds of variables, as well as constraint right hand side constants. These are:

- `ooOPSPlusInfinity`
  Setting the upper bound of a variable to `PlusInfinity` implies that this variable is effectively unbounded from above.

  Setting the upper bound of an inequality constraint to `PlusInfinity` is the standard way to represent an inequality of the form $c(x) \geq LB$.
- `ooOPSMinusInfinity`
  Setting the lower bound of a variable to `MinusInfinity` implies that this variable is effectively unbounded from below.

  Setting the lower bound of an inequality constraint to `MinusInfinity` is the standard way to represent an inequality of the form $c(x) \leq UB$.

A.3.2 The `si`, `sd` and `li` Argument Types and the `IntSeq` Auxiliary Function

In order to describe the arguments to the various methods of its object classes, *oo$\mathcal{OPS}$* introduces the following C++ type definition:

[2]as a result of earlier specifications

- `si`: a sequence of integers
- `sd`: a sequence of doubles
- `li`: a list of integers

The precise implementation of this type is irrelevant as *ooOPS* also provides an auxiliary function for its construction. In particular, the `IntSeq` function takes a list of any number of integers and returns a vector of type `si`.

For example, the C++ code segment:

```
si* SetSize = IntSeq (3,7,4) ;
```

creates a sequence called `SetSize` of three integers (3, 7 and 4). This may then, for instance, be passed as an input argument to a method to create a new 3-dimensional variable set, with the lengths of the three dimensions being specified in `SetSize`.

A.4 The MINLP object class

A.4.1 MINLP instantiation: the `NewMINLP` function

Declaration: `ops* NewMINLP()`

Function: Creates a new empty MINLP.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Specified On Entry
return value	`ops*`	the MINLP object

Notes: None

Example of usage:

The following creates a new MINLP called `NetOpt` :

```
ops* NetOpt = NewMINLP() ;
```

A.4.2 MINLP construction methods

A.4.2.1 Method `NewContinuousVariable`

Declaration: `void NewContinuousVariable(string vname, si* dimLB, si* dimUB, double LB, double UB, double value)`

Function: Creates a new set of continuous variables with given name, domain, bounds and value.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`vname`	`string`	name of variable set to be created
`dimLB`	`si*`	lower bounds of the variable set dimensions
`dimUB`	`si*`	upper bounds of the variable set dimensions
`LB`	`double`	lower bound
`UB`	`double`	upper bound
`value`	`double`	value

Arguments returned to client: None

Notes:

- The specified name must be unique among all variables (continuous *or* integer) in the MINLP as it will be used to identify the variable in all future communications with the MINLP object.
- In general, a variable is represented by a multidimensional set (cf. section A.2.2). The arguments `dimLB` and `dimUB` are sequences of integers, representing respectively the lower and upper bounds of the domains of the individual dimensions. In the case of the example of set X mentioned in section A.2.2, these two sequences would be (0, 1, -1) and (10, 5, 1) respectively.
- The length of the above integer sequences must be equal to each other and implicitly determine the number of dimensions of the variable.
- Scalar variables should be specified as 1-dimensional sets of size 1.
- The index of the individual elements for each dimension ranges from the corresponding element of `dimLB` to the corresponding element of `dimUB`.
- The specified lower bound, upper bound and value must satisfy:

$$LB \leq value \leq UB$$

- All elements of the variable set being created are given the same lower and upper bounds as well as the same value. However, these quantities may subsequently be changed for individual elements or slices of the set (see sections A.4.3.1 and A.4.3.2).

Example of usage:

The following creates a 2-dimensional continuous variable called `MaterialFlow`, with individual elements `MaterialFlow`$(i,j), i = 0,..,4, j = 1,..,6$ within an existing `ops` object called `NetOpt`. All elements of the new variable are initialised with a lower bound of 0.0, an upper bound of 100.0 and a current value of 1.0.

```
NetOpt->NewContinuousVariable("MaterialFlow", IntSeq(0,1),
                                     IntSeq(4,6), 0.0, 100.0, 1.0);
```

A.4.2.2 Method `NewIntegerVariable`

Declaration: `void NewIntegerVariable(string vname, si* dimLB, si* dimUB, int LB, int UB, int value)`

Function: Creates a new set of integer variables with given name, size, bounds and value.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`vname`	`string`	name of variable set to be created
`dimLB`	`si*`	lower bounds of the variable set dimensions
`dimUB`	`si*`	upper bounds of the variable set dimensions
`LB`	`int`	lower bound
`UB`	`int`	upper bound
`value`	`int`	value

Arguments returned to client: None

Notes:

- The specified name must be unique among all variables (continuous *or* integer) in the MINLP as it will be used to identify the variable in all future communications with the MINLP object.
- In general, a variable is represented by a multidimensional set (cf. section A.2.2). The arguments `dimLB` and `dimUB` are sequences of integers, representing respectively the lower and upper bounds of the domains of the individual dimensions. In the case of the example of set X mentioned in section A.2.2, these two sequences would be (0, 1, -1) and (10, 5, 1) respectively.
- The length of the above integer sequences must be equal to each other and implicitly determine the number of dimensions of the variable.
- Scalar variables should be specified as 1-dimensional sets of size 1.
- The index of the individual elements for each dimension ranges from the corresponding element of `dimLB` to the corresponding element of `dimUB`.
- The specified lower and upper bounds ,and value must satisfy:

$$LB \leq value \leq UB$$

- All elements of the variable set being created are given the same lower and upper bounds as well as the same value. However, these quantities may subsequently be changed for individual elements or slices of the set (see sections A.4.3.1 and A.4.3.2).

Example of usage:

The following creates a 1-dimensional integer (binary) variable called `PlantExists` of length 3 with individual elements `PlantExists`$(i), i = 1, .., 3$ within an existing `ops` object called `NetOpt`. All elements of the new variable are initialised with a lower bound of 0, an upper bound of 1 and a current value of 0.

```
NetOpt->NewIntegerVariable("PlantExists", IntSeq(1), IntSeq(3),
                                 0, 1, 0);
```

A.4.2.3 Method `NewConstraint`

Declaration: `void NewConstraint(string cname, si* dimLB, si* dimUB, double LB, double UB)`

Function: Creates a new set of constraints with given name, size, type and bounds.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`cname`	`string`	name of constraint set to be created
`dimLB`	`si*`	lower bounds of the constraint set dimensions
`dimUB`	`si*`	upper bounds of the constraint set dimensions
`LB`	`double`	lower bound
`UB`	`double`	upper bound

Arguments returned to client: None

Notes:

- The specified name must be unique among all constraints in the MINLP as it will be used to identify the constraint in all future communications with the MINLP object.

- In general, a constraint is represented by a multidimensional set (cf. note A.2.2 in section A.2.5). The arguments `dimLB` and `dimUB` are sequences of integers, representing respectively the lower and upper bounds of the domains of the individual dimensions. Their length must be equal and determine the number of dimensions of the constraint set.
- Scalar constraints should be specified as 1-dimensional sets of size 1.
- The index of the individual elements for each dimension ranges from the corresponding element of `dimLB` to the corresponding element of `dimUB`.
- All elements of the constraint set being created are assigned lower and upper bounds. However, these may subsequently be changed (see section A.4.3.3).
- Initially, the constraint does not contain any variables occurring linearly in it. It is also assigned a null nonlinear part.

Example of usage:

Creates a new constraint `UniqueAllocation` with bounds fixed at 1.

```
NetOpt->NewConstraint("UniqueAllocation", IntSeq(1), IntSeq(1),
                      1.0, 1.0);
```

A.4.2.4 `AddLinearVariableSliceToConstraintSlice`

Declaration: `void AddLinearVariableSliceToConstraintSlice(string vname, si* vdimLB. si* vdimUB, string cname, si* cdimLB, si* cdimUB, double coefficient)`

Function: Adds a linear occurrence of every element of a specified variable slice to each element of a specified constraint slice, always with the specified coefficient.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`vname`	`string`	variable set, elements of which are to be added
`vdimLB`	`si*`	lower bounds of the variable set slice to be added to constraints
`vdimUB`	`si*`	upper bounds of the variable set slice to be added to constraints
`cname`	`string`	name of constraint set, elements of which are to receive the new variable occurrences
`cdimLB`	`si*`	lower bounds of the constraint set slice to receive variable occurrences
`cdimUB`	`si*`	upper bounds of the constraint set slice to receive variable occurrences
`coefficient`	`double`	coefficient of variables in the constraints

Arguments returned to client: None

Notes:

- The specified variable set must have already been created using the `NewContinuousVariable` or `NewIntegerVariable` methods (see sections A.4.2.1 and A.4.2.2 respectively).
- The specified constraint set must have already been created using the `NewConstraint` method (see section A.4.2.3).
- If an element of the variable slice has already been declared to occur in an element of constraint slice via an earlier invocation of this method, then the current specification supersedes the earlier one.

- A value of 0.0 for the specified `coefficient` has the effect of removing an existing occurrence of a variable in a constraint.

Examples of usage:

The following creates a new inequality constraint called `SourceFlowLimit` within an existing `ops` object called `NetOpt` for each of the 5 source nodes in a network with a right hand side of 450. It then adds to the constraint for each node all the elements of the corresponding row of the variable `MaterialFlow` (cf. example in section A.4.2.1) with a coefficient of 1.

```
NetOpt->NewConstraint ("SourceFlowLimit",
                       IntSeq(0), IntSeq(4), '<', 450.0);
for (int i=0; i<=4; i++)
  NetOpt->AddLinearVariableSliceToConstraintSlice(
    "MaterialFlow",     IntSeq(i,1), IntSeq(i,6),
    "SourceFlowLimit",  IntSeq(i),   IntSeq(i),   1.0);
```

The following adds an occurrence of all elements of the variable `PlantExists` (cf. example in section A.4.2.2) to the scalar constraint `UniqueAllocation` (cf. example in section A.4.2.3) within an existing `ops` object called `NetOpt`.

```
NetOpt->AddLinearVariableSliceToConstraintSlice
        ("PlantExists",      IntSeq(1), IntSeq(3),
         "UniqueAllocation", IntSeq(1), IntSeq(3), 1.0);
```

A.4.2.5 Method `NewConstant`

Declaration: `void NewConstant(string kname, si* kdimLB, si* kdimUB, double value)`

Function: Creates a new constant set of specified dimensionality, each element of which is initialized to the specified value.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`kname`	`string`	name of the constant set/slice being created
`kdimLB`	`si*`	lower bounds of constant set/slice
`kdimUB`	`si*`	upper bounds of constant set/slice
`value`	`value`	initial value

Arguments returned to client: None

Notes:

- None

Examples of usage:

The following code creates a constant called `Const1` within an existing `ops` object called `NetOpt`, consisting of a three-dimensional vector whose components are all 1.

```
NetOpt->NewConstant("Const1", IntSeq(1), IntSeq(3), 1);
```

A.4.2.6 Method `NewConstantExpression`

Declaration: `void NewConstantExpression(string ename, string kname, si* kdimLB, si* kdimUB)`

Function: Creates a new expression consisting of a constant set or constant slice.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`ename`	`string`	name of the expression being created
`kname`	`string`	name of the constant set/slice
`kdimLB`	`si*`	lower bounds of constant set/slice
`kdimUB`	`si*`	upper bounds of constant set/slice

Arguments returned to client: None

Notes:

- If an expression called `ename` already exists within the *ops* object, the new expression will overwrite it.

Examples of usage:

The following code creates an expression called `Expr1` (within an existing `ops` object called `NetOpt`) from the constant set `Const1`.

```
NetOpt->NewConstant("Const1", IntSeq(1), IntSeq(3), 1);
NetOpt->NewConstantExpression("Expr1", "Const1",
                              IntSeq(1), IntSeq(3));
```

A.4.2.7 Method `NewVariableExpression`

Declaration: `void NewVariableExpression(string ename, string vname, si* vdimLB, si* vdimUB*)`

Function: Creates a new expression consisting of a variable set or a variable slice.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`ename`	`string`	name of the expression
`vname`	`string`	name of the variable set/slice
`vdimLB`	`si*`	lower bounds of the variable set/slice
`vdimUB`	`si*`	upper bounds of the variable set/slice

Arguments returned to client: None

Notes:

- If an expression called `ename` already exists within the *ops* object, the new expression will overwrite it.

Examples of usage:

The following code creates a variable expression called `Expr2` (within an existing `ops` object called `NetOpt`) consisting of the vector (x_1, x_2, x_3).

```
NetOpt->NewContinuousVariable("x", IntSeq(1), IntSeq(3), -1, 1, 0);
NetOpt->NewVariableExpression("Expr2", "x", IntSeq(1), IntSeq(3));
```

A.4.2.8 Method `BinaryExpression`

Declaration: `void BinaryExpression(string ename, string ename1, si* edimLB1, si* edimUB1, string ename2, si* edimLB2, si* edimUB2, string binaryoperator)`

Function: Creates a new expression representing a binary operation between the expressions specified in the argument.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`ename`	`string`	name of the new expression
`ename1`	`string`	name of expression representing the first operand
`edimLB1`	`si*`	lower bound of slice to be used from first expression
`edimUB1`	`si*`	upper bound of slice to be used from first expression
`ename2`	`string`	name of expression representing the second operand
`edimLB2`	`si*`	lower bound of slice to be used from second expression
`edimUB2`	`si*`	upper bound of slice to be used from second expression
`binaryoperator`	`string`	label of the binary operator to be used.

Arguments returned to client: None

Notes:

- The dimensionality size of the new expression depends on those of its operands (see table in section A.2.4).
- The argument `binaryoperator` describes the type of binary operator in the expression. It can be one of the following strings: `"sum"`, `"difference"`, `"product"`, `"ratio"`, `"power"` (cf. section A.2.4).
- If an expression called `ename` already exists within the *ops* object, the new expression will overwrite it.

Examples of usage: The following code creates a nonlinear term $x_1 x_2$ within an existing `ops` object called `NetOpt`.

```
NetOpt->NewContinuousVariable("x", IntSeq(1), IntSeq(2), -1, 1, 0);
NetOpt->BinaryExpression("NLT", "x", IntSeq(1), IntSeq(1),
                              "x", IntSeq(2), IntSeq(2), "product");
```

A.4.2.9 Method `UnaryExpression`

Declaration: `void UnaryExpression(string ename, string ename1, si* edimLB1, si* edimUB1, string unaryoperator)`

Function: Creates a new expression representing a unary operation on the expression specified in the argument.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`ename`	`string`	name of the new expression
`ename1`	`string`	name of expression representing the operand
`edimLB1`	`si*`	lower bound of slice to be used from the operand expression
`edimUB1`	`si*`	upper bound of slice to be used from the operand expression
`unaryoperator`	`string`	label of the unary arithmetic operator or unary transcendental function to be used

Arguments returned to client: None

Notes:

- The dimensionality and dimension sizes of the new expression is the same as that of its operands.
- The argument `unaryoperator` describes the type of unary operator in the expression. It can be one of the following strings (the meaning is self-explanatory): `"minus"`, `"log"`, `"exp"`, `"sin"`, `"cos"`, `"tan"`, `"cot"`, `"sinh"`, `"cosh"`, `"tanh"`, `"coth"` (cf. section A.2.4).
- If an expression called `ename` already exists within the *ops* object, the new expression will overwrite it.

Examples of usage: The following code creates a nonlinear term $\log(x)$ within an existing `ops` object called `NetOpt`.

```
NetOpt->NewContinuousVariable("x", IntSeq(1), IntSeq(3), -1, 1, 0);
NetOpt->UnaryExpression("NLT", "x", IntSeq(1), IntSeq(3), "log");
```

A.4.2.10 `AssignExpressionSliceToConstraintSlice`

Declaration: `void AssignExpressionSliceToConstraintSlice (string ename, si* edimLB, si* edimUB, string cname, si* cdimLB, si* cdimUB)`

Function: Assigns the specified expression slice to the specified constraint slice.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`ename`	`string`	the name of the expression to be set
`edimLB`	`si*`	lower bound of expression slice
`edimUB`	`si*`	upper bound of expression slice
`cname`	`string`	name of constraint set to which the expression must be set
`cdimLB`	`si*`	lower bounds of the constraint set slice to be set with the expression elements
`cdimUB`	`si*`	upper bounds of the constraint set slice to be set with the expression elements

Arguments returned to client: None

Notes:

- The specified constraint set must have already been created using the `NewConstraint` method (see section A.4.2.3).

- The specified expression either must be a scalar or its dimensionality size must match exactly that of the constraint slice. In the former case, the same scalar expression will be assigned to each of the constraints in the slice; in the latter, each element in the expression slice will be assigned to the corresponding element in the constraint slice.

Examples of usage:

The following code creates the term $\frac{x}{\log(x)}$ within an existing `ops` object called `NetOpt` and assigns it to the constraint `UniqueAllocation`.

```
NetOpt->NewContinuousVariable("x", IntSeq(1), IntSeq(3), -1, 1, 0);
NetOpt->UnaryExpression("NLT", "x", IntSeq(1), IntSeq(3), "log");
NetOpt->BinaryExpression("NLT", "x",
                         IntSeq(1), IntSeq(3), "NLT",
                         IntSeq(1), IntSeq(3), "ratio");
NetOpt->AssignExpressionSliceToConstraintSlice
        ("NLT",             IntSeq(1), IntSeq(3),
         "UniqueAllocation", IntSeq(1), IntSeq(3));
```

A.4.2.11 Method `NewObjectiveFunction`

Declaration: `void NewObjectiveFunction(string oname, string otype)`

Function: Specifies the name and the type for the objective function for the MINLP.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`oname`	`string`	objective function name
`otype`	`string`	objective function type

Arguments returned to client: None

Notes:

- The objective function created by this method does not contain any variables either linearly or nonlinearly.
- The type of the objective function must be a string of at least 3 characters; any characters beyond the third one are ignored. Valid type specifications are `"min"` and `"max"` denoting minimization and maximization respectively. The case of the characters in the type specification is irrelevant.
- If the method is invoked more than once for a given MINLP, then each invocation supersedes all earlier ones and any information associated with the previous objective function (e.g. on the variables occurring in it) is lost.

Examples of usage:

The following creates an objective function called `TotalProfit` within an existing `ops` object called `NetOpt`, that is to be maximized by the solution of the MINLP:

```
NetOpt->NewObjectiveFunction{"TotalProfit", "max") ;
```

A.4.2.12 Method `AddLinearVariableSliceToObjectiveFunction`

Declaration: `void AddLinearVariableSliceToObjectiveFunction (string vname, si* dimLB, si* dimUB, double coefficient, string oname)`

Function: Adds a linear occurrence of every element of a specified variable slice to the specified objective function using the specified coefficient.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`vname`	`string`	name of variable set, elements of which are to be added
`dimLB`	`si*`	lower bounds of the variable set slice to be added to objective function
`dimUB`	`si*`	upper bounds of the variable set slice to be added to objective function
`coefficient`	`double`	coefficient of variables in the objective function
`oname`	`string`	name of current objective function

Arguments returned to client: None

Notes:

- The specified variable set must have already been created using the `NewContinuousVariable` or `NewIntegerVariable` methods (see sections A.4.2.1 and A.4.2.2 respectively).
- If an element of the variable slice has already been declared to occur in the objective function via an earlier invocation of this method, then the current specification supersedes the earlier one.
- A value of 0.0 for the specified `coefficient` has the effect of removing an existing occurrence of a variable in a constraint.
- The specified objective function must be the current objective function which must have already been created using the `NewObjectiveFunction` method (see section A.4.2.11).

Examples of usage:

The following adds the elements of variables `MaterialFlow` (cf. example in section A.4.2.1) relating to destination node 1 to the objective function `TotalProfit` (cf. example in section A.4.2.11) within the existing `ops` object called `NetOpt`. A coefficient of 100 is used. It then subtracts from the same objective function the sum of the integer variables `PlantExists` (cf. example in section A.4.2.2) multiplied by a coefficient of -0.1:

```
NetOpt->AddLinearVariableSliceToObjectiveFunction
        ("MaterialFlow", IntSeq(0, 1), IntSeq(4,1),
         100.0, "TotalProfit");
NetOpt->AddLinearVariableSliceToObjectiveFunction
        ("PlantExists",  IntSeq(1), IntSeq(3),
         -0.1,  "TotalProfit");
```

A.4.2.13 Method `AssignExpressionToObjectiveFunction`

Declaration: `void AssignExpressionToObjectiveFunction (string ename, string oname)`

Function: Sets the specified expression to the specified objective function.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`ename`	`string`	expression to be set
`oname`	`string`	name of current objective function

Arguments returned to client: None

Notes:

- The specified objective function must be the current objective function which must have already been created using the `NewObjectiveFunction` method (see section A.4.2.11).
- The expression must be scalar.

Examples of usage:

The following code creates the scalar nonlinear term x_1x_2 and assigns it to the objective function `TotalProfit` within an existing `ops` object called `NetOpt`.

```
NewContinuousVariable("x", IntSeq(1), IntSeq(2), -1, 1, 0);
BinaryExpression("NLT", "x", IntSeq(1), IntSeq(1),
                 "x", IntSeq(2), IntSeq(2), "product");
AssignExpressionToObjectiveFunction("NLT", "TotalProfit");
```

A.4.3 MINLP modification methods

A.4.3.1 Method `SetVariableValue`

Declaration: `void SetVariableValue(string vname, si* dimLB, si* dimUB, double value)`

Function: Sets the current value of the elements of a specified slice of a specified variable set, overriding their previous values.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`vname`	`string`	name of variable set, elements of which are to be modified
`dimLB`	`si*`	lower bounds of the variable set slice to be modified
`dimUB`	`si*`	upper bounds of the variable set slice to be modified
`value`	`double`	new value for elements to be modified

Arguments returned to client: None

Notes:

- The specified variable set must have already been created using the `NewContinuousVariable` or `NewIntegerVariable` methods (see sections A.4.2.1 and A.4.2.2 respectively).
- The specified value must lie between the lower and upper bounds for *every* element of the specified variable slice.

Examples of usage:

The following modifies all elements of the variable set `MaterialFlow` (cf. example in section A.4.2.1) pertaining to source 3 to a new value of 150:

```
SetVariableValue ("MaterialFlow", IntSeq(3, 1), IntSeq(3,6),
                  150.0) ;
```

A.4.3.2 Method `SetVariableBounds`

Declaration: `void SetVariableBounds(string vame, si* dimLB, si* dimUB, double LB, double UB)`

Function: Sets the current lower and upper bounds of the elements of a specified slice of a specified variable set, overriding their previous values.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`vname`	`string`	name of variable set, elements of which are to be modified
`dimLB`	`si*`	lower bounds of the variable set slice to be modified
`dimUB`	`si*`	upper bounds of the variable set slice to be modified
`LB`	`double`	new value of lower bound for elements to be modified
`UB`	`double`	new value of upper bound for elements to be modified

Arguments returned to client: None

Notes:

- The specified variable set must have already been created using the `NewContinuousVariable` or `NewIntegerVariable` methods (see sections A.4.2.1 and A.4.2.2 respectively).
- The specified bounds must satisfy $LB \leq UB$.

Examples of usage:

The following modifies the bounds of all elements of the variable set `MaterialFlow` (cf. example in section A.4.2.1) pertaining to source 3 to new values of 10 and 20 respectively:

```
SetVariableBounds("MaterialFlow", IntSeq(3, 1), IntSeq(3,6),
                  10.0, 20.0) ;
```

The following sets both bounds of an element of the variable set `PlantExists` (cf. example in section A.4.2.2) to 1, thereby effectively fixing the value of this variable in any MINLP solution also to 1:

```
SetVariableBounds("PlantExists",  IntSeq(2), IntSeq(2),
                  1.0, 1.0) ;
```

A.4.3.3 Method `SetConstraintBounds`

Declaration: `void SetConstraintBounds(string cname, si* dimLB, si* dimUB, double LB, double UB)`

Function: Sets the lower and upper bounds of a specified slice of a specified constraint set, overriding any previous bounds.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`cname`	`string`	name of constraint set, elements of which are to be modified
`dimLB`	`si*`	lower bounds of the constraint set slice to be modified
`dimUB`	`si*`	upper bounds of the constraint set slice to be modified
`LB`	`double`	new lower bound for elements to be modified
`UB`	`double`	new upper bound for elements to be modified

Arguments returned to client: None

Notes:

- The specified constraint set must have already been created using the `NewConstraint` method (see section A.4.2.3).
- The specified bounds must satisfy $LB \leq UB$.

Examples of usage:

The following modifies the lower and upper bounds of constraint `UniqueAllocation` (cf. example in section A.4.2.3) to infinity:

```
SetConstraintBounds("UniqueAllocation",  IntSeq(0), IntSeq(0),
                    MinusInfinity, PlusInfinity);
```

This modification effectively de-activates the `UniqueAllocation` constraint.

A.4.3.4 Method `SetConstantValue`

Declaration: `void SetConstantValue(string kname, si* indexlist, double value)`

Function: Sets the value of the constant element pointed to by the specified index list to the specified value.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`kname`	`string`	name of constant set containing the element to be modified
`indexlist`	`si*`	list of indices which point to the element to be modified
`value`	`double`	new value to be assigned to element

Arguments returned to client: None

Notes:

- The indices in the index list must lie between the respective index bounds defining the dimensionality of the constant set.

A.4.3.5 **Method** `SetConstantSliceValue`

Declaration: `void SetConstantSliceValue(string kname, si* kdimLB, si* kdimUB, double value)`

Function: Sets the values of the constant elements in the specified slice to the specified value.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`kname`	`string`	name of constant set containing the elements to be modified
`kdimLB`	`si*`	lower bounds of the constant set slice to be modified
`kdimUB`	`si*`	upper bounds of the constant set slice to be modified
`value`	`double`	new value to be assigned to elements

Arguments returned to client: None

Notes:

- None.

A.4.3.6 **Method** `SetKeyVariable`

Declaration: `void SetKeyVariable(string vame, si* dimLB,`
`si* dimUB)`

Function: Sets the elements of a specified slice of a specified variable set to be used as key variables for the decomposition algorithm.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`vname`	`string`	name of variable set, elements of which are to be modified
`dimLB`	`si*`	lower bounds of the variable set slice to be modified
`dimUB`	`si*`	upper bounds of the variable set slice to be modified

Arguments returned to client: None

Notes:

- The specified variable set must have already been created using the `NewContinuousVariable` or `NewIntegerVariable` methods (see sections A.4.2.1 and A.4.2.2 respectively).

Examples of usage:

The following sets all elements of the variable set `MaterialFlow` (cf. example in section A.4.2.1) to be key variables:

```
SetKeyVariable ("MaterialFlow", IntSeq(3, 1), IntSeq(3,6)) ;
```

The following sets an element of the variable set `PlantExists` (cf. example in section A.4.2.2) to be a key variable:

```
SetKeyVariable ("PlantExists",  IntSeq(2), IntSeq(2)) ;
```

A.4.4 Structured MINLP Information Access Methods

A.4.4.1 Method `GetVariableInfo`

Declaration: `void GetVariableInfo(string vname, si* dimLB, si* dimUB, double* value, double* LB, double* UB)`

Function: Returns the current values and lower and upper bounds of all elements of a specified slice of a specified set of variables.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`vname`	`string`	name of variable set on which information is required
`dimLB`	`si*`	lower bounds of the variable set slice on which information is required
`dimUB`	`si*`	upper bounds of the variable set slice on which information is required

Arguments returned to client:

Argument	Type	Value on Exit
`value`	`double*`	pointer to set of real numbers containing current values of elements in specified slice
`LB`	`double*`	pointer to set of real numbers containing lower bounds of elements in specified slice
`UB`	`double*`	pointer to set of real numbers containing upper bounds of elements in specified slice

Notes:

- The specified variable set must have already been created using the `NewContinuousVariable` or `NewIntegerVariable` methods (see sections A.4.2.1 and A.4.2.2 respectively).
- In the case of multidimensional variable sets, the information in the sets `value`, `LB` and `UB` is ordered so that the last index varies fastest, followed by the penultimate index, and so on until the first index which varies the most slowly. For example, a 2-dimensional set is ordered by rows.

Examples of usage:

The following returns the information on a slice of the variable `MaterialFlow` (cf. example in section A.4.2.1):

```
double* CurrentFlows ;
double* MinimumFlows ;
```

```
double* MaximumFlows ;

GetVariableInfo ("MaterialFlow",  IntSeq(1,1), IntSeq(4,4),
                     CurrentFlows, MinimumFlows, MaximumFlows) ;
```

A.4.4.2 Method `GetConstraintInfo`

Declaration: `bool GetConstraintInfo(string cname, si* dimLB, si* dimUB, double* LB, double* UB, double* LagMult)`

Function: Returns the current values, lower and upper bounds and Lagrange multipliers of all elements of a specified slice of a specified constraint set; the return value is `true` if every element of the constraint slice is linear, or `false` otherwise.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`cname`	`string`	name of constraint set on which information is required
`dimLB`	`si*`	lower bounds of the constraint set slice on which information is required
`dimUB`	`si*`	upper bounds of the constraint set slice on which information is required

Arguments returned to client:

Argument	Type	Value on Exit
`LB`	`double*`	pointer to set of real numbers containing the upper bounds of elements in specified slice
`UB`	`double*`	pointer to set of real numbers containing the upper bounds of elements in specified slice
`LagMult`	`double*`	pointer to set of real numbers containing the Lagrange multipliers of elements in specified slice
return value	`bool`	true if constraint is linear

Notes:

- The return value is `true` if every element of the constraint is linear and `false` otherwise.
- The specified constraint set must have already been created using the `NewConstraint` method (see section A.4.2.3).
- In the case of multidimensional constraint sets, the information in the sets `LB`, `UB` and `LagMult` is ordered so that the last index varies fastest, followed by the penultimate index, and so on until the first index which varies the most slowly. For example, a 2-dimensional set is ordered by rows.
- If the Lagrange multiplier for a element of the specified slice is not available (e.g. because the MINLP has not yet been solved or because the MINLP solver does not make this information available), the a value of `PlusInfinity` (cf. section A.3.1) will be returned for the corresponding element of `LagMult`.

Examples of usage:

The following returns information on all elements of the constraint `SourceFlowLimit` (cf. example in section A.4.2.3):

```
char*   CurrentType ;
```

```
double* CurrentUB  ;
double* CurrentLM   ;
bool isLinear    ;

isLinear = GetConstraintInfo("SourceFlowLimit",
                             IntSeq(0), IntSeq(4),
                             CurrentType, CurrentUB, CurrentLM);
```

A.4.4.3 Method `GetObjectiveFunctionInfo`

Declaration: `bool GetObjectiveFunctionInfo(string oname, char* otype, double& ovalue)`

Function: Returns the current value and type of the objective function specified by name; the "return value" is `true` if the constraint slice has an expression set to it, `false` otherwise.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`oname`	`string`	name of objective function on which information is required

Arguments returned to client:

Argument	Type	Value on Exit
`otype`	`char*`	pointer to set of characters containing the type of the objective function
`ovalue`	`double&`	value of the objective function
return value	`bool`	true if objective function is linear

Notes:

- The return value is `false` if the objective function has a nonlinear expression assigned to it and `true` otherwise.
- The specified objective function must have already been created using the `NewObjectiveFunction` method (see section A.4.2.11).
- The returned type of the objective function could be either `"min"` or `"max"`, denoting minimization and maximization respectively.
- The value of the objective function returned is based on the current values of the variables.

Examples of usage:

The following returns information on the objective function named `TotalProfit` (cf. example in section A.4.2.11):

```
char*   CurrentObjType ;
double  CurrentObjValue  ;
bool    isLinear;

isLinear = GetObjectiveFunctionInfo ("TotalProfit", CurrentObjType,
                                      CurrentObjValue) ;
```

A.4.4.4 Method `GetProblemInfo`

Declaration: `GetProblemInfo(string pname, bool& islinear, double& ovalue, bool& isfeasible)`

Function: Returns the problem name, whether the problem is a MILP or a MINLP, the current recorded value of the objective function (as set by the last solver module that tried to solve the problem), and whether the problem is feasible or not (with respect to the current values of problem variables).

Arguments to be specified by the client: None.

Arguments returned to client:

Argument	Type	Value on Exit
`pname`	`string`	name of the problem
`islinear`	`bool&`	`true` if problem is a MILP, `false` if it is a MINLP
`ovalue`	`double&`	value of the objective function
`isfeasible`	`bool&`	`true` if problem is feasible, `false` if it is not

Notes:

- The variable `ovalue` is *not* the value of the objective function calculated at the current variable values, but rather the value set by the solver module that last tried to solve this problem.

Examples of usage:

The following returns information on the problem.

```
string ProblemName;
bool MILP;
double ObjFunValue;
bool Feasible;

NetOpt->GetProblemInfo(ProblemName, MILP, ObjFunValue, Feasible);
```

A.4.5 Flat MINLP Information Access Methods

Although it is convenient for client programs to construct `ops` objects in a structured manner using the methods of section A.4.2, most existing numerical solvers are designed to operate on the much simpler "flat" form **[Flat MINLP]** described in section A.1.

In view of the above, the `ops` interface provides a set of methods that allows access to the information characterising this flat representation. The latter is constructed automatically and efficiently by *ooOPS* in a manner that is transparent to the client.

A.4.5.1 General Properties of the Flat MINLP

The flat MINLP generated by *ooOPS* has the following characteristics:

- The constraints in the flat MINLP comprise those elements of the constraint sets in the `ops` object that fulfill both of the following criteria:

  - they have at least one variable occurring linearly with a non-zero coefficient, or a nonlinear part occurrence;
  - at least one of the bounds is different from the respective infinity constant.

- The variables in the flat MINLP comprise those elements of the variable sets in the `ops` object that appear with a non-zero coefficient in at least one of the constraints and/or in the objective function in the flat MINLP (see above), or in the nonlinear parts of at least one constraint and/or the objective function.

A.4.5.2 Method `GetFlatMINLPSize`

Declaration: `void GetFlatMINLPSize(int& nv, int& niv, int& nlv, int& nliv, int& nc, int& nlc, int& nlz, int& nnz, int& nlzof, int& nnzof)`

Function: Returns information on the size of the flat MINLP.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value on Exit
`nv`	`int`	total number of variables in flat MINLP
`niv`	`int`	number of integer variables in flat MINLP
`nlv`	`int`	number of variables in flat MINLP which only appear linearly
`nliv`	`int`	number of integer variables in flat MINLP which only appear linearly
`nc`	`int`	total number of constraints in flat MINLP
`nlc`	`int`	number of linear constraints in flat MINLP
`nlz`	`int`	number of non-zero elements in the matrix of flat MINLP (A in eqn. (A.2))
`nnz`	`int`	number of non-zero elements in the Jacobian matrix of the nonlinear constraints (f in eqn. (A.2))
`nlzof`	`int`	number of variables having non-zero coefficients in the linear part of the objective function of flat MINLP
`nnzof`	`int`	number of non-zero first order derivatives in the objective function of flat MINLP

Notes:

- The numbers of variables and constraints in the flat MINLP are determined using the rules detailed in section A.4.5.1.
- The number of variables `nv` includes both continuous and integer variables.

Examples of usage:

The following returns information on the size of a flat MINLP described by an existing `ops` object called `NetOpt`:

```
int NumberOfVariables                  ;
int NumberOfIntegerVariables           ;
int NumberOfLinearVariables            ;
int NumberOfLinearIntegerVars          ;
int NumberOfConstraints                ;
int NumberOfLinearConstraints          ;
int NumberOfNZLinVarsInConstraints    ;
int NumberOfNZNonLinJacInConstraints  ;
int NumberOfNZLinVarsInObjFun         ;
int NumberOfNZNonLinJacInObjFun       ;

NetOpt->GetFlatMINLPSize(&NumberOfVariables,
                         &NumberOfIntegerVariables,
                         &NumberOfLinearVariables,
                         &NumberOfLinearIntegerVariables,
                         &NumberOfConstraints,
                         &NumberOfLinearConstraints,
                         &NumberOfNZLinVarsInConstraints,
                         &NumberOfNZNonLinJacInConstraints,
                         &NumberOfNZLinVarsInObjFun,
                         &NumberOfNZNonLinJacInObjFun);
```

A.4.5.3 Method `GetFlatMINLPStructure`

Declaration: `void GetFlatMINLPStructure(int* rowindex, int* columnindex, int* objindex, string structuretype )`

Function: Returns information on the sparsity structure of the objective function and the constraints. This corresponds to one of the following, depending on the request issued by the client:

1. the linear variable occurrences;
2. the jacobian elements occurrences;
3. the union of the preceding structures.

Arguments to be specified by the client:

Argument	Type	Specified on Entry
`structuretype`	`string`	specifies whether returned structure should be of type (1), (2) or (3) (see above)

Arguments returned to client:

Argument	Type	Value on Exit
`rowindex`	`int*`	pointer to set of integers containing the numbers of the constraints in the flat MINLP from which the nonzero elements originate
`columnindex`	`int*`	pointer to set of integers containing the numbers of the variables in the flat MINLP from which the nonzero elements in constraints originate
`objindex`	`int*`	pointer to set of integers containing the numbers of the variables in the flat MINLP from which the nonzero elements in the objective function originate

Notes:

- The input parameter `structuretype` must be one of the following strings: `"LINEAR", "NONLINEAR", "BOTH"` depending on whether the client needs the linear structure, the nonlinear structure or a union of both.

- The integer sets pointed at by `rowindex` and `columnindex` are both of length `nlz, nnz` or `nlz + nnz` (see section A.4.5.2) depending on whether `structuretype` is `"LINEAR", "NONLINEAR"` or `"BOTH"`.
- The integer set pointed at by `objindex` is of length `nlzof, nnzof` or `nlzof + nnzof` (see section A.4.5.2) according as to whether `structuretype` is `"LINEAR", "NONLINEAR"` or `"BOTH"`.
- Constraints and variables in the flat MINLP are numbered starting from 1.
- When calling with `"BOTH"` the linear and nonlinear vectors may be overlapping (i.e. there may be indices $i < j$ such that `rowindex[i]` = `rowindex[j]` and `columnindex[i]` = `columnindex[j]`, but the actual derivatives refer respectively to linear and nonlinear entries). For example, if the first flat problem constraint is $0 \leq x_1^2 + 3x_1 \leq 0$ then the linear derivatives matrix entry $(1,1)$ is 3, and the nonlinear derivatives matrix entry $(1,1)$ is $\frac{\partial x_1^2}{\partial x_1}$ evaluated at the current value of x_1. Thus the union of linear and nonlinear problem structure contains two entries for position $(1,1)$, but the first refers to the linear derivative and the second refers to the nonlinear derivative.
- The same is true for the objective function structure.

Examples of usage:

The following returns the linear structure of a flat MINLP described by an existing `ops` object called `NetOpt`:

```
int* Rows          ;
int* Columns       ;
int* ObjFunColumns ;

NetOpt->GetFlatMINLPStructure(Rows, Columns,
                              ObjFunColumns, "LINEAR");
```

A.4.5.4 Method `GetFlatMINLPVariableInfo`

Declaration: `void GetFlatMINLPVariableInfo(int vid, string& vname, si* &index, bool& isinteger, bool& islinear, double& value, double& LB, double& UB)`

Function: Returns information pertaining to a variable in the flat MINLP.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`vid`	`int`	the number of the variable in the flat MINLP structure

Arguments returned to client:

Argument	Type	Value on Exit
`&vname`	`string`	the name of the variable set from which this variable originates
`&index`	`si*`	the index of the variable in the variable set from which it originates
`&isinteger`	`bool`	true if the variable is of type integer false otherwise
`&islinear`	`bool`	true if the variable only appears linearly in the problem
`&value`	`double`	the current value of the variable
`&LB`	`double`	the lower bound of the variable
`&UB`	`double`	the upper bound of the variable

Notes:

- The variable number `vid` specified must be in the range 1,.., `nv` (see section A.4.5.2).
- Index holds a pointer to a sequence which is part of the MINLP's internal data and must not be altered in any way.

Examples of usage:

The following returns information on variable 375 in a flat MINLP described by an existing `ops` object called `NetOpt`:

```
string vname      ;
si*    index      ;
bool   isinteger  ;
bool   islinear   ;
double value      ;
double LB         ;
double UB         ;

NetOpt->GetFlatMINLPVariableInfo(375, vname, index, isinteger,
                                 islinear, value, LB, UB);
```

On return from `GetFlatMINLPVariableInfo`, variable `vname` could have the value `MaterialFlow` (cf. example in section A.4.2.1), `index` could be `(2, 4)`, *isinteger* could be `false`, and `value`, `LB` and `UB` could be 1, 0 and 100 respectively. Thus, we can deduce that variable 375 in the flat MINLP originated from the continuous variable `MaterialFlow(2,4)` in the original (structured) MINLP.

A.4.5.5 Method `SetFlatMINLPVariableBounds`

Declaration: `void SetFlatMINLPVariableBounds(double* LB, double* UB)`

Function: Changes the lower and upper problem variable bounds.

Arguments to be specified by the client:

Argument	Type	Specified on Entry
`LB`	`double*`	set of double precision numbers which will hold the new lower bounds
`UB`	`double*`	set of double precision numbers which will hold the new upper bounds

Arguments returned to client: None

Notes:

- The sets `LB` and `UB` have size `nv` (see section A.4.5.2).

Examples of usage:

The following reads the values of the problem variables described by an existing `ops` object called `NetOpt`:

```
double* vl = new double [nv];
double* vu = new double [nv];
for(int i = 0; i < nv; i++) {
  vl[i] = 0;
  vl[i] = 1;
}
NetOpt->SetFlatMINLPVariableBounds(vl, vu);
```

A.4.5.6 Method `GetFlatMINLPVariableValues`

Declaration: `void GetFlatMINLPVariableValues(double* values)`

Function: Fills the set of double precision numbers passed to the function with the current values of the flat MINLP problem variables.

Arguments to be specified by the client:

Argument	Type	Specified on Entry
`values`	`double*`	set of double precision numbers which will hold the variable values

Arguments returned to client: None

Notes:

- The set `values` has size `nv` (see section A.4.5.2).

Examples of usage:

The following sets the bounds of the problem variables described by an existing `ops` object called `NetOpt`:

```
double* v = new double [nv];
NetOpt->GetFlatMINLPVariableValues(v);
```

A.4.5.7 Method `SetFlatMINLPVariableValues`

Declaration: `void SetFlatMINLPVariableValues(double* values)`

Function: Sets the values of the flat MINLP variables.

Arguments to be specified by the client:

Argument	Type	Specified on Entry
`values`	`double*`	set of double precision numbers holding the variable values to be set

Arguments returned to client: None

Notes:

- The set `values` has size `nv` (see section A.4.5.2).

Examples of usage:

The following sets the values of the problem variables described by an existing `ops` object called `NetOpt`:

```
double* vv = new double [nv];
for(int i = 0; i < nv; i++)
    vv[i] = i / 2;
NetOpt->SetFlatMINLPVariableValues(vv);
```

A.4.5.8 Method `GetFlatMINLPConstraintInfo`

Declaration: `void GetFlatMINLPConstraintInfo(int cid, string& cname, si* &index, double& LB, double& UB, si* &vlist, sd* &cflist, FlatExpression* &fe)`

Function: Returns information pertaining to a constraint in the flat MINLP.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`cid`	`int`	the number of the constraint in the flat MINLP structure

Arguments returned to client:

Argument	Type	Value on Exit
`&cname`	`string`	the name of the constraint set from which this constraint originates
`&index`	`si*`	the index of the constraint in the constraint set from which it originates
`&LB`	`double`	the current constraint lower bound
`&UB`	`double`	the current constraint upper bound
`&vlist`	`si*`	the vector of integers which contains the variable indices occuring in this constraint
`&cflist`	`sd*`	the vector of doubles which contains the coefficients of the variables occuring in this constraint
`&fe`	`Flat-Expres-sion*`	an object which contains the symbolic information which defines the nonlinear part of the constraint (see section A.6.2)

Notes:

- The variable number `cid` specified must be in the range 1,..,`nc` (see section A.4.5.2).
- Index holds a pointer to a sequence which is part of the MINLP's internal data and must not be altered in any way.
- The argument `FlatExpression* &fe` contains the symbolic information which defines the nonlinear part of the constraint. For its description and usage see section A.6.2.

Examples of usage: The following returns information on constraint 532 in a flat MINLP described by an existing `ops` object called `NetOpt`:

```
string          cname   ;
si*             index   ;
double          LB     ;
double          UB     ;
si*             varList ;
sd*             coefList;
FlatExpression* fe;

NetOpt->GetFlatMINLPConstraintInfo(532, cname, index, LB, UB,
                                   varList, coefList, fe)
```

On return from `GetFlatMINLPConstraintInfo`, variable `cname` could have the value `"SourceFlowLimit"` (cf. example in section A.4.2.4), `index` could be `(3)`, and `UB` could be 450. Thus, we can deduce that constraint 532 in the flat MINLP originated from the constraint `SourceFlowLimit(3)` in the original (structured) MINLP.

A.4.5.9 Method `EvalFlatMINLPNonlinearObjectiveFunction`

Declaration: `double EvalFlatMINLPNonlinearObjectiveFunction(void)`

Function: Returns the value of the nonlinear part of the objective function.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Specified On Entry
return value	`double`	the value of the objective function at the current variable values

Notes: None

Examples of usage:

The following sets the variable values and evaluates the objective function of the problem described by an existing `ops` object called `NetOpt`:

```
double* vv = new double [nv];
for(int i = 0; i < nv; i++)
    vv[i] = i / 2;
NetOpt->SetFlagMINLPVariableValues(vv);
double vof = NetOpt->EvalFlatMINLPNonlinearObjectiveFunction();
```

A.4.5.10 Method `EvalFlatMINLPNonlinearConstraint`

Declaration: `void EvalFlatMINLPNonlinearConstraint(int lowercid, int uppercid, double* values)`

Function: Returns the values of a range of nonlinear parts of constraints, starting with constraint `lowercid` up to and including constraint `uppercid` (also see (A.4.5.8)).

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`lowercid`	`int`	the number of the first constraint of the range to be evaluated
`uppercid`	`int`	the number of the last constraint of the range to be evaluated

Arguments returned to client:

Argument	Type	Value on Exit
`values`	`double*`	vector containing the constraint values

Notes:

- The vector `values` containing the values of the evaluated constraints, has size `uppercid - lowercid + 1`.

Examples of usage:

The following evaluates constraints 2-5 in the flat MINLP described by an existing `ops` object called `NetOpt`:

```
double* cv = new double[4];
NetOpt->EvalFlatMINLPNonlinearConstraint(cv, 2, 5);
```

A.4.5.11 Method `GetFlatMINLPObjectiveFunctionDerivatives`

Declaration: `void GetFlatMINLPObjectiveFunctionDerivatives (double* A, string structuretype)`

Function: Returns a vector containing one of the following:

1. the values of the nonzero coefficients in the vector c defining the linear part of the objective function (cf. equation (A.1));
2. the values of the nonzero first order partial derivatives of the nonlinear part of the objective function evaluated at the current variable values;
3. the union of the preceding vectors.

Arguments to be specified by the client:

Argument	Type	Specified on Entry
`structuretype`	`string`	specifies whether returned structure should be of type (1), (2) or (3) (see above)

Arguments returned to client:

Argument	Type	Value on Exit
`values`	`double*`	pointer to set of doubles containing the nonzero elements of c, of the derivatives of the obj. fun., or both

Notes:

- The input parameter `structuretype` must be one of the following strings: `"LINEAR", "NONLINEAR", "BOTH"` depending on whether the client needs information about the linear part of the objective function, or the nonlinear part, or both.
- The length of this vector follows the rules given in note 3 to section A.4.5.3.
- The indices of the variables to which the elements of this vector correspond can be obtained from method `GetFlatMINLPStructure` as the integer vector `objindex` (see section A.4.5.3).
- When calling with `"BOTH"`, see notes on page 168.

Examples of usage:

The following returns the nonzero partial derivatives of the nonlinear part of the objective function of a flat MINLP described by an existing `ops` object called `NetOpt`, evaluated at the current variable values:

```
int d0, d1, d2, d3, d4, d5, d6, d7, d8;
int nnzof;
NetOpt->GetFlatMINLPSize(&d0, &d1, &2, &d3, &d4,
                              &d5, &d6, &d7, &8, &d9,  &nnzof);
double* A = new double [nnzof];
NetOpt->GetFlatMINLPObjectiveFunctionDerivatives(A, "NONLINEAR");
```

In this case, $c(k)$, $k = 1$,..,`nlzof` is the coefficient of variable `objindex(k)` in the objective function.

A.4.5.12 Method `GetFlatMINLPConstraintDerivatives`

Declaration: `void GetFlatMINLPConstraintDerivatives(string structuretype, int lowercid, int uppercid, sd& values)`

Function: Returns a sequence of doubles (see section A.3.2) containing one of the following:

1. the values of the nonzero coefficient in the linear part of the specified constraints;
2. the values of the nonzero partial derivatives, evaluated at the current variable values, of the nonlinear parts of the specified constraints;
3. the union of the preceding vectors.

Arguments to be specified by the client:

Argument	Type	Specified on Entry
`structuretype`	`string`	specifies whether returned structure should be of type (1), (2) or (3) (see above)
`lowercid`	`int`	specifies the lower end of the constraint range
`uppercid`	`int`	specifies the upper end of the constraint range

Arguments returned to client:

Argument	Type	Value on Exit
`values`	`sd&`	sequence of doubles containing the nonzero elements of A, of the Jacobian of f, or both

Notes:

- The input parameter `structuretype` must be one of the following strings: `"LINEAR", "NONLINEAR", "BOTH"` according as to whether the client needs the linear part of the constraints, the derivatives, or a union of both.
- The length of this vector is given by `values.size()`.
- If `lowercid` and `uppercid` are both set to zero, then this function returns the nonzero coefficients of the entire matrix A (see equation (A.2)), or the nonzero coefficients of the entire Jacobian of f (see equation (A.2)) evaluated at the current variable values, or both, depending on `structuretype`. In this case this method effectively acts as though the whole range of problem constraints had been specified.
- The row and column indices of the elements of the vector can be obtained from method `GetFlatMINLP-Structure` as integer vectors `rowindex` and `columnindex` respectively (see section A.4.5.3).
- Nonlinear derivatives which are identically zero are not recorded. Thus, for example, if you have a linear constraint and you request its nonlinear derivatives, the vector `values` might be empty. Referring to elements of an empty vector results in runtime segmentation fault errors, so it is advisable to check `values.size()` before using the vector.
- When calling with `"BOTH"`, see notes on page 168.

Examples of usage:

The following returns the matrix A of a flat MINLP described by an existing `ops` object called `NetOpt`:

```
sd A;
NetOpt->GetFlatMINLPConstraintDerivatives(A, 0, 0, "LINEAR");
```

In this case, $A(k)$, k =1,..,`nlz` is the coefficient of variable `columnindex(k)` in constraint `rowindex(k)` in the left hand side matrix A (cf. equation (A.2)).

A.4.5.13 Method `GetFlatMINLPNoPartitions`

Declaration: `void GetFlatMINLPNoPartitions(int& np)`

Function: Returns the number of partitions of the flat MINLP.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value on Exit
`np`	`int`	total number of partitions of flat MINLP

Notes:

- The number of partitions are deternimed based on the properties of key variables.

Examples of usage:

The following returns the number of partitions occuring in a flat MINLP described by an existing `ops` object called `NetOpt`:

```
int  np;

NetOpt->GetFlatMINLPNoPartitions(np) ;
```

A.4.5.14 Method `GetFlatMINLPPartition`

Declaration: `void GetFlatMINLPPartition(int& np, li* &varlist, li* &conlist)`

Function: Returns information on the partitions of the flat MINLP specified by its number.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`np`	`int`	the number of the partition of the flat MINLP

Arguments returned to client:

Argument	Type	Value on Exit
`varlist`	`li`	list of variable indices occuring in the partition
`conlist`	`li`	list of constraint indeces occuring in the partition

Notes:

- The number np must be less or equal to the maximum number of partitions.

Examples of usage:

The following returns the variable and constraint lists of the partition 2 occuring in a flat MINLP described by an existing `ops` object called `NetOpt`:

```
int  np;
```

```
li*  variableList;
li*  constraintList;

NetOpt->GetFlatMINLPPartition(np, variableList, constraintList) ;
```

A.4.6 Standard Form MINLP Information Access Methods

A MINLP is in *standard form* when its nonlinear parts are reduced to their basic building blocks and all its linear parts are gathered together in a matrix. This is explained in more details in [116]. Suffice it here to recall the basics with an example. The constraint

$$-1 \leq 4x_1 + 3x_2 - x_3 + \frac{x_1 x_2}{\log(x_3)} \leq 10$$

in standard form becomes

$$\begin{array}{rll} -1 \leq & w_1 & \leq 10 \\ & w_1 & = 4x_1 + 3x_2 - x_3 + w_2 \\ & w_2 & = \dfrac{w_3}{w_4} \\ & w_3 & = x_1 x_2 \\ & w_4 & = \log(x_3) \end{array}$$

The standard form of a MINLP is as follows:

$$\begin{array}{rcl} \min_x & x_i & \\ l \leq & Ax & \leq u \\ y & = & w \otimes z \\ x^L \leq & x & \leq x^U \end{array} \tag{A.6}$$

where y, w, z are subsets of $x \cup \mathbb{R}$ and $\otimes$ is any unary or binary operator.

In view of the above, the `ops` interface provides a set of methods that allows access to the information characterising this standard form representation. The latter is constructed automatically and efficiently by *ooOPS* in a manner that is transparent to the client.

A.4.6.1 Method `GetSFNumberOfVariables`

Declaration: `void GetSFNumberOfVariables(int& nv, int& nzlv)`

Function: It returns the total number of variables of the problem in standard form and the number of linearly appearing variables with nonzero coefficients in the linear part of the problem.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value On Exit
`nv`	`int&`	the number of total problem variables
`nzlv`	`int&`	number of nonzero coeff. linear variables

Notes:

- The total number of problem variables includes the original problem variables and the variables which have been added by the standard form process.

Examples of usage:

```
int  nv;
int  nzlv;
NetOpt->GetSFNumberOfVariables(nv, nzlv) ;
```

A.4.6.2 Method `GetSFVariableInfo`

Declaration: `void GetSFVariableInfo(int vid, double value, double& LB, double& UB)`

Function: It returns the current variable value and the lower and upper bounds of variable `vid` in the problem in standard form.

Arguments to be specified by the client:

Argument	Type	Specified on Entry
`vid`	`int`	variable id in the standard form problem

Arguments returned to client:

Argument	Type	Value On Exit
`value`	`double&`	current variable value
`LB`	`double&`	variable lower bound
`UB`	`double&`	variable upper bound

Notes: None

Examples of usage:

```
double value, LB, UB;
NetOpt->GetSFVariableInfo(1, value, LB, UB) ;
```

A.4.6.3 Method `GetSFObjFunVarIndex`

Declaration: `void GetSFObjFunVarIndex(int& vid)`

Function: It returns the variable index (variable id) corresponding to the objective function.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value on Exit
`vid`	`int&`	variable index corresponding to the objective function

Notes:

- As well as the constraints, the objective function of the MINLP is transformed by the standard form reduction, so that, for example, min x_1x_2 would become min w_1 s.t. $w_1 = x_1x_2$. In this case, the variable index of w_1 would be returned.

Examples of usage:

```
int  vid;
NetOpt->GetSFObjFunVarIndex(vid);
```

A.4.6.4 Method `GetSFNumberOfLinearConstraints`

Declaration: `void GetSFNumberOfLinearConstraints(int& nlc)`

Function: This returns the number of linear constraints in the standard form of the problem.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value on Exit
`nlc`	`int&`	number of linear constraints

Notes: None

- The number of linear constraints is equal to the number of rows in the matrix A in the general formulation of the standard form (see eqn. A.6 above).

Examples of usage:

```
int  nlc;
NetOpt->GetSFNumberOfLinearConstraints(nlc);
```

A.4.6.5 Method `GetSFLinearBounds`

Declaration: `void GetSFLinearBounds(double* lb, double* ub)`

Function: It returns the vector of lower and upper bounds in the linear constraints of the standard form problem (l and u in the formulation A.6).

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value on Exit
`lb`	`double*`	vector of lower bounds
`ub`	`double*`	vector of upper bounds

Notes:

- The arrays `lb` and `ub` must be created by the client with the correct length `nlc` (use method `GetSF-NumberOfLinearConstraints` above).

Examples of usage:

```
int     nlc;
NetOpt->GetSFNumberOfLinearConstraints(nlc);
double* lb = new double[nlc];
double* ub = new double[nlc];
NetOpt->GetSFLinearBounds(lb, ub);
```

A.4.6.6 Method `GetSFLinearStructure`

Declaration: `void GetSFLinearStructure(int* rowindex, int* columnindex)`

Function: Returns the sparsity structure of the linear part of the problem in standard form.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value on Exit
`rowindex`	`int*`	vector of row indices
`columnindex`	`int*`	vector of column indices

Notes:

- The arrays `rowindex` and `columnindex` must be created by the client with the correct length `nzlv` (use method `GetSFNumberOfVariables` above).

Examples of usage:

```
int  nv;
int  nzlv;
NetOpt->GetSFNumberOfVariables(nv, nzlv) ;
int* rowindex = new int [nzlv];
int* columnindex = new int [nzlv];
NetOpt->GetSFLinearStructure(rowindex, columnindex);
```

A.4.6.7 Method `GetSFMatrix`

Declaration: `void GetSFMatrix(double* A)`

Function: Returns the linear part of the standard form problem in sparse format.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value on Exit
`A`	`double*`	matrix A in sparse form

Notes:

- The array `A` must be created by the client with the correct length `nlzv` (use method `GetSFNumberOf-Variables` above).
- The sparsity structure can be found with the method `GetSFLinearStructure` (A.4.6.6).

Examples of usage:

```
int  nv;
int  nzlv;
NetOpt->GetSFNumberOfVariables(nv, nzlv) ;
double* A = new double [nzlv];
NetOpt->GetSFMatrix(A);
```

A.4.6.8 Method `GetSFNumberOfNonlinearConstraints`

Declaration: `void GetSFNumberOfNonlinearConstraints(int& nnlc)`

Function: This returns the number of nonlinear constraints in the standard form of the problem.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value on Exit
`nnlc`	`int&`	number of nonlinear constraints

Notes: None

- The number of nonlinear constraints is equal to the number of "constraint definitions" of the form $y = w \otimes z$ in the general formulation of the standard form (see eqn. A.6 above).

Examples of usage:

```
int  nnlc;
NetOpt->GetSFNumberOfNonlinearConstraints(nnlc);
```

A.4.6.9 Method `GetSFNonlinearConstraint`

Declaration: `void GetSFNonlinearConstraint(int cid, int& vid, int& vid1, int& vid2, string& operator, double& constant1, double& constant2`

Function: This function returns the elements of nonlinear standardized constraint number `cid` in the standard form of the problem.

Arguments to be specified by the client:

Argument	Type	Specified on Entry
`cid`	`int`	standardized nonlinear constraint id

Arguments returned to client:

Argument	Type	Value on Exit
`vid`	`int&`	left hand side variable id
`vid1`	`int&`	first right hand side variable id
`vid2`	`int&`	second right hand side variable id
`operator`	`string&`	operator type
`constant1`	`double&`	first left hand side constant
`constant2`	`double&`	second left hand side constant

Notes:

- The standard nonlinear constraint ID (`cid`) always starts from 1.
- In this discussion, we assume that each standard form nonlinear constraint has the form

$$\textit{variable} = \textit{operand1} \otimes \textit{operand2}$$

where *variable* is the "added variable" that is defined by the right hand side and the operator $\otimes$ is either unary or binary (if it is unary, then *operand2* is a dummy placeholder).

- `vid` is the variable id of *variable*.
- `vid1` is the variable id of *operand1* if the latter is a variable (e.g. $w_1 = x_1 x_2$). If *operand1* is a constant then `vid1` is set to -1 (e.g. $w_2 = \frac{2}{x_3}$).
- `vid2` is the the variable id of *operand2* if the latter is a variable and if $\otimes$ is a binary operator. If *operand2* is a constant (e.g. $w_3 = x_4^2$) or if $\otimes$ is unary then `vid2` is set to -1.
- `operator` represents the type of operator. It can be one of the following strings (the meaning is self-explanatory): `"sum"`, `"difference"`, `"product"`, `"ratio"`, `"power"`, `"minus"`, `"log"`, `"exp"`, `"sin"`, `"cos"`, `"tan"`, `"cot"`, `"sinh"`, `"cosh"`, `"tanh"`, `"coth"` (cf. section A.2.4).
- `constant1` is meaningful only when `vid1` is set to -1.
- `constant2` is meaningful only when `vid2` is set to -1 and $\otimes$ is a binary operator.
- Note that in a problem in standard form the constraint id `cid` only applies to nonlinear standardized constraints and does not take into account linear constraints.

Examples of usage:

```
int     cid = 1;
int     vid;
int     vid1;
int     vid2;
string  operat;
double  constant1;
double  constant2;
NetOpt->GetSFNonlinearConstraint(cid, vid, vid1, vid2,
                                 operat, constant1, constant2);
```

A.4.6.10 Method `UpdateSolution`

Declaration: `void UpdateSolution(double* sol)`

Function: This method is specifically designed to make it easy to insert the solution of the MINLP into both the structured and the flat form. In short, this method updates the variable values in both the structured and flat MINLP forms.

Arguments to be specified by the client:

Argument	Type	Specified on Entry
`sol`	`double`	array of (flat) variable values

Arguments returned to client: None

Notes:

- The length of the array `sol` has to be at least `nv`, the number of variables in flat form (see section A.4.5.2).
- This method has been specifically designed to make it easy to update the solution in the MINLP at the end of a MINLP solver module, and is therefore targeted towards those programmers who wish to write their own solver module.

Examples of usage:

```
double  sol[nv];
NetOpt->UpdateSolution(&(sol[0]));
```

A.5 MINLP solvers and systems

A.5.1 Introduction

Section A.4 of this document described in detail how `ops` objects can be constructed and modified, and how information in them can be accessed in both a structured and a flat form. This section is concerned with the solution of the mathematical problem described by an already existing `ops` object.

A.5.1.1 MINLP solver managers and MINLP systems

The solution of an MINLP is normally effected by a numerical solver. There are commercial solvers (e.g. SNOPT) as well as non-commercial ones (e.g. DONLP2). A major objective for the *ooOPS* software is to provide application programs with a *uniform* interface to all such solvers. This is achieved by embedding each solver within a `opssolvermanager` object.

The main function of the Manager for a given MINLP Solver is the creation of `opssystem` objects (cf. section A.2.1) by combining an existing `ops` object with the numerical solver embedded within the Manager. It is this combination that ultimately permits the solution of the MINLP to take place.

A.5.1.2 Algorithmic parameters for MINLP solvers

MINLP solvers of the kind of interest to *ooOPS* are sophisticated pieces of software. Although the basic algorithms implemented by different solvers are often very similar, the specific implementations may be significantly different. Moreover, the users of these solvers are normally provided with substantial flexibility in configuring the details of the behaviour of the implementation. This is typically achieved by setting the values of one or more *algorithmic parameters*. These are usually quantities of logical, integer, real or string type. Different MINLP solvers may recognise different sets of parameters; some typical examples include:

- The branching strategy to be used by branch-and-bound algorithms (e.g. depth-first, breadth-first, etc.).
- The maximum number of nodes to be examined during the branch-and-bound search.
- The maximum CPU time to be spent by the solution.
- The infeasibility tolerance within which constraints need to be satisfied.

Usually, MINLP solvers also incorporate a default value for each parameter. Although these values may lead to reasonably good performance for a wide range of applications, sophisticated users may wish to change them to suit the specific characteristics of particular applications.

ooOPS provides general mechanisms for handling algorithmic parameters that allows the client program (a) to determine the parameters a particular MINLP solver recognises, and their current values, and (b) to specify new values for one or more of these parameters. Parameter specification can operate at two levels:

- *At the Solver Manager level*:
  Specifying the value of a parameter in a `MINLPSolverManager` object ensures that any `opssystem` objects *subsequently* created from this `MINLPSolverManager` will, at least initially, have this value of the parameter.
- *At the MINLP System level*:
  Specifying the value of a parameter in a `opssystem` object affects this particular object only.

A.5.2 The `ssolpar` and `sstat` argument types

In order to handle the passing of these different kinds of parameters, *ooOPS* introduces the following C++ type definitions:

- `variant`: a union containing a value which may be one of several different types
- `ssolpar`: a sequence of solver parameters
- `sstat`: a sequence of solution statistics

The following C++ type definitions, which are included in the `ops.h` header file supplied to the user describe these types fully:

```
enum vtype {logical, integer, real, expression};

struct variant {
        vtype thetype;
        union val {
          int    ival;
          double  dval;
          string* sval;
          bool    bval;
        };
};


struct solparameter {
  variant theval;
  string  name;
  string  description;
  double  lowerbound;
  string  upperbound;
};

typedef vector<solparameter> ssolpar;

struct statistic {
  variant theval;
  string  description;
  string  unit;
};

typedef vector<statistic> sstat;
```

A.5.3 MINLP Solver Manager Instantiation: The `NewMINLPSolverManager` Function

Declaration: `opssolvermanager* (*NewMINLPSolverManager)(void)`

Function: Creates a new `MINLPSolverManager` object incorporating a specified numerical code.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Specified On Entry
return value	`opssolvermanager*`	the ops solver manager incorporating the MINLP

Notes:

- This function is not available at compile time (hence it is only a pointer to a function). It is loaded from a shared object library at run time using the `dlopen()/dlsym()` mechanism. See example below for details.

- When using run time linking, keep in mind that search paths for shared object files vary from operating system to operating system and do not usually include the current working directory.
- Please be warned that shared object files produced from C++ source code have ”mangled” symbol named which usually `dlopen` and `dlsym` cannot read properly. There are two solutions: first, write wrapper functions to `dlopen` and `dlsym` which take care of this problem; and second, use non-demangled symbol names (as in the example below).
- When writing a new solver manager for a particular solver code, the new solver manager has to expose the following function in the global namespace:

  ```
  opssolvermanager* NewMINLPSolverManager(void) {
    // ... code
    return new opssolvermanager_i();
  }
  ```

  Please use the provided template source files for the creation of new solver managers.

Examples of usage:

The following creates a new MINLP solver manager called `MySNOPTSolverManager` incorporating the SNOPT numerical MINLP solver:

```
#include <dcfcn.h>
opssolvermanager* (*NewMINLPSolverManager)(void);
opssolvermanager* MySNOPTSolverManager;
void* handle = dlopen("libopssnopt.so", RTLD_LAZY);
if (!handle) {
  cerr << "MAIN: shared object error: \n\t" << dlerror() << endl;
  exit(-1);
} else {
  // have to use "non-demangled" C++ symbol names
  void *tmp = dlsym(handle, "NewMINLPSolverManager__Fv");
  char* error;
  if ((error = dlerror()) != NULL) {
    cerr << "MAIN: shared object error: \n\t" << error << endl;
    exit(-1);
  }
  NewMINLPSolverManager = (opssolvermanager* (*)()) (tmp);
}
opssolvermanager* MySNOPTSolverManager = (*NewMINLPSolverManager)();
// ... code
dlclose(handle);
```

This manager can now be used to create one or more MINLP systems, each incorporating a separate `ops` object (see section A.5.4.3 below).

A.5.4 MINLP solver managers

A.5.4.1 Method `GetParameterList`

Declaration: `ssolpar* GetParameterList()`

Function: Gets the list of parameters with which a MINLPSystem can be configured. It returns a sequence of structures holding the current values of the parameters, their (single word) names and short descriptions, and valid upper and lower bounds where applicable (the values `MinusInfinity` and `PlusInfinity` will be used to indicate unconstrained parameters).

The section A.5.2 for the detailed description of this type.

Arguments to be specified by the client: None

Arguments returned to client: None

Examples of usage:

The following retrieves the list of parameters for a solver:

```
ssolpar* params = SnoptManager->GetParameterList();
```

A.5.4.2 Method `SetParameter`

Declaration: `void SetParameter(string ParamName, variant ParamValue)`

Function: Sets a specific parameter to configure a Solver Manager. Subsequent calls to `GetParameterList` will return the value supplied, and all `MINLPSystem`s subsequently created will use the value supplied.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
ParamName	string	parameter name
ParamValue	variant	assigns a value to the parameter

Arguments returned to client: None

Examples of usage:

The following sets a parameter named “`MaxRelaxations`” to 100 in the Solver Manager `SnoptManager`:

```
variant var100;
var100.vtype=integer;
var100.val.ival=100;
SnoptManager->SetParameter("MaxRelaxations",var100) ;
```

A.5.4.3 Method `NewMINLPSystem`

Declaration: `opssystem* NewMINLPSystem(const ops* theops)`

Function: Creates a new `opssystem` object from a given `ops` object.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`theops`	`const ops*`	the `ops` object to be incorporated in the new `opssystem`

Arguments returned to client:

Argument	Type	Specified On Entry
return value	`opssystem*`	the ops system incorporating the MINLP and the numerical code

Notes:

- The specified `theops` object must already exist, having been created using the `NewMINLP` function (cf. section A.4.1 and the methods described in sections A.4.2 and A.4.3.

Examples of usage:

The following uses the MINLP solver manager object `SnoptManager` created in the example of section A.5.3 to create a new MINLP system, called `NetOptSnopt` incorporating the `ops` object `NetOpt` created in the example of section A.4.1:

```
opssystem* NetOptSnopt = SnoptManager->NewMINLPSystem(NetOpt) ;
```

A.5.5 MINLP systems

A.5.5.1 Method `GetParameterList`

Declaration: `ssolpar* GetParameterList()`

Function: Gets the list of parameters with which a MINLPSystem can be reconfigured after creation. It returns a sequence of structures holding the current values of the parameters, their (single word) names and short descriptions, and valid upper and lower bounds where applicable (the values `MinusInfinity` and `PlusInfinity` will be used to indicate unconstrained parameters).

See section A.5.2 for the detailed description of this type.

Arguments to be specified by the client: None

Arguments returned to client: None

Examples of usage:

The following retrieves the list of parameters for a system:

```
 ssolpar* params=NetOptSnopt->GetParameterList();
```

A.5.5.2 Method `SetParameter`

Declaration: `void SetParameter(string ParamName,`
`variant ParamValue)`

Function: Sets a specific parameter to configure an MINLPSystem.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`ParamName`	`string`	parameter name
`ParamValue`	`variant`	assigns a value to the parameter

Arguments returned to client: None

Examples of usage:

The following sets a parameter named "`DiagnosticsOutputFile`" to "`mydiag.out`" in the MINLPSystem `NetOptSnopt`:

```
variant filename;
filename.vtype=string;
filename.val.sval="mydiag.out";
NetOptSnopt->SetParameter("DiagnosticsOutputFile",filename) ;
```

A.5.5.3 Method `GetStatistics`

Declaration: `sstat* GetStatistics()`

Function: Gets the list of solution statistics which accumulate during the lifetime of the MINLPSystem. It returns a sequence of structures holding the current values of the statistics, their short descriptions, and units.

See section A.5.2 for the detailed description of this type.

Arguments to be specified by the client: None

Arguments returned to client: None

Examples of usage:

The following retrieves the statistics for a system and writes them to standard output, assuming the C++ << operator has been suitably configured for the `variant` type.

```
sstat* stats = NetOptSnopt->GetStatistics();
cout << "Solution statistics:" << endl;
for(sstat::const_iterator it = stats->begin();
    it != stats->end();
    it++)
  cout << "   " << it->description << ";"
       << it->theval << it->unit << endl;
```

The output produced might be:

```
Solution statistics:
   CPU time: 233.23 seconds
   Number of relaxations: 23
```

A.5.5.4 Method `Solve`

Declaration: `void Solve()`

Function: Attempts to solve the MINLP problem incorporated in the `opssystem` object using the numerical MINLP solver embedded within the `opssystem` object.

Arguments to be specified by the client: None

Arguments returned to client: None

Notes:

- The numerical solution algorithm is applied to the "flat" form of the MINLP. The solver obtains the latter from the `ops` object incorporated within the `opssystem` object using the methods of section A.4.5. This operation is performed automatically and is completely transparent to the client.
- The solution procedure will leave the final values of variables and other information within the `ops` object. The client may subsequently retrieve them from there using the methods of section A.4.4.

Examples of usage:

The following method invocation applied to the MINLP system `NetOptSnopt` created in the example of section A.5.4.3 will trigger the solution of the MINLP described by the `ops` object `NetOpt` using the SNOPT solver:

```
NetOptSnopt->Solve() ;
```

A.5.5.5 Method `GetSolutionStatus`

Declaration: `void GetSolutionStatus(int& status)`

Function: Returns the exit status of the solver which attempted to solve the MINLP.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Specified On Entry
`status`	`int&`	exit status

Notes:

- This method only returns meaningful information after the method `Solve()` has been called.

Examples of usage:

The following returns the exit status of the numerical solver (SNOPT) which just tried to solve the `NetOpt` problem

```
int status;
NetOptSnopt->GetSolutionStatus(status);
if (status == 0)
  cout << "Solution OK!" << endl;
```

A.6 Auxiliary interfaces

A.6.1 The convexification module

This interface provides functionality for producing a convex relaxation of the MINLP. The *ooOPS* software provides a `convexifiermanager` object which embeds an existing `ops` object and provides only one public method, `Convexify()`, which returns an `ops` containing a convex flat MILP.

The "convexification" algorithm gets its input data from the MINLP in standard form (see section A.4.6) and produces a convex (linear) MILP. The output convex problem is embedded in a slightly modified version of the `ops` class (see section A.4) whose interface only offers flat form data access (see section A.6.1.4).

A.6.1.1 Convexifier manager instantiation: the function `NewConvexifierManager`

Declaration: `convexifiermanager* (*NewConvexifierManager) (ops* theops)`

Function: Creates a new `convexifiermanager` object incorporating the MINLP `theops`.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
`theops`	`ops*`	the MINLP to be convexified

Arguments returned to client:

Argument	Type	Specified On Entry
return value	`convexifiermanager*`	the convexifiermanager incorporating the MINLP

Notes:

- This function is not available at compile time (hence it is only a pointer to a function). It is loaded from a shared object library at run time using the `dlopen()/dlsym()` mechanism. See example below for details.
- When using run time linking, keep in mind that search paths for shared object files vary from operating system to operating system and do not usually include the current working directory.
- Please be warned that shared object files produced from C++ source code have ”mangled” symbol named which usually `dlopen` and `dlsym` cannot read properly. There are two solutions: first, write wrapper functions to `dlopen` and `dlsym` which take care of this problem; and second, use non-demangled symbol names (as in the example below).

Examples of usage: The following creates a new `convexifiermanager` incorporating the MINLP `NetOpt`.

```
#include <dcfcn.h>
convexifiermanager* (*NewConvexifierManager)(ops*);
convexifiermanager* NetOptCM;
void* handle = dlopen("libopssnopt.so", RTLD_LAZY);
if (!handle) {
  cerr << "MAIN: shared object error: \n\t" << dlerror() << endl;
  exit(-1);
} else {
  // have to use "non-demangled" C++ symbol names
  void *tmp = dlsym(handle, "NewMINLPSolverManager__FP3ops");
  char* error;
  if ((error = dlerror()) != NULL) {
    cerr << "MAIN: shared object error: \n\t" << error << endl;
    exit(-1);
  }
  NetOptCM = (convexifiermanager*) (*)(ops*)) (tmp);
}
// ... code
dlclose(handle);
```

A.6.1.2 Method `GetConvexMINLP`

Declaration: `ops* GetConvexMINLP(void)`

Function: Returns a convex linear relaxation of the MINLP embedded in the `convexifiermanager`.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Specified On Entry
return value	`ops*`	the convex linear relaxation of the MINLP

Notes:

- The memory allocated by the returned convex linear problem should not be deallocated before the convexifiermanager object is deleted.

Examples of usage: The following returns a new `ops` object containing a convex linear relaxation of the MINLP.

```
ops* myconvexops = NetOptCM->GetConvexMINLP() ;
```

A.6.1.3 Method `UpdateConvexVarBounds`

Declaration: `void UpdateConvexVarBounds(double* lb, double* ub)`

Function: Updates the convex problem created with `GetConvexMINLP` (see A.6.1.2) with new variable bounds. Because of the way the convexification is done, this has the effect of changing some of the linear structure of the convex problem.

Arguments to be specified by the client:

Argument	Type	Specified On Entry
lb	`double*`	new lower bounds of variables in the convex problem
ub	`double*`	new upper bounds of variables in the convex problem

Arguments returned to client: None

Notes:

- The convex problem created with `GetConvexMINLP` must not be deallocated prior to the call to this function.

Examples of usage: The following updates bounds to the first convex problem variable.

```
// get convex problem
ops* myconvexops = NetOptCM->GetConvexMINLP() ;
// get convex problem size
int NumberOfVariables                  ;
int NumberOfIntegerVariables           ;
int NumberOfLinearVariables            ;
int NumberOfLinearIntegerVars          ;
int NumberOfConstraints                ;
int NumberOfLinearConstraints          ;
int NumberOfNZLinVarsInConstraints     ;
int NumberOfNZNonLinJacInConstraints   ;
int NumberOfNZLinVarsInObjFun        ;
int NumberOfNZNonLinJacInObjFun      ;
myconvexops->GetFlatMINLPSize(&NumberOfVariables,
                              &NumberOfIntegerVariables,
                              &NumberOfLinearVariables,
                              &NumberOfLinearIntegerVariables,
                              &NumberOfConstraints,
                              &NumberOfLinearConstraints,
                              &NumberOfNZLinVarsInConstraints,
                              &NumberOfNZNonLinJacInConstraints,
                              &NumberOfNZLinVarsInObjFun,
                              &NumberOfNZNonLinJacInObjFun);
// get convex problem variable bounds
double vlb = new double [NumberOfVariables];
double vub = new double [NumberOfVariables];
string strdummy;
si* sidummy;
bool bdummy1, bdummy2;
double ddummy;
for(int i = 1; i <= NumberOfVariables; i++) {
  myconvexops->GetFlatMINLPVariableInfo(i, strdummy, sidummy,
                                        bdummy1, bdummy2, ddummy,
                                        vlb[i - 1], vub[i - 1]);
}
// change first variable bounds
vlb[0] = vlb[0] - 1;
vub[0] = vub[0] + 1;
// update the convex problem
NetOptCM->UpdateConvexVarBounds(vlb, vub);
```

A.6.1.4 Methods of the convex MILP

This is a cut-down version of the MINLP object interface (the `ops` class, see section A.4) which only offers functionality for reading/writing (linear) data in the flat form problem. This modified `ops` class offers no methods for dealing with (nonlinear) information, no multidimensional construction methods and no standard form methods.

The methods provided by this interface are:

- `GetFlatMINLPSize` (see A.4.5.2);

  Notes: Call is the same as in section A.4.5.2.

- `GetFlatMINLPStructure` (see A.4.5.3);

  Notes: Call is the same as in section A.4.5.3.

- `GetFlatMINLPVariableInfo` (see A.4.5.4);

  Notes: The arguments `string vname`, `si* index`, `bool isinteger`, `bool islinear` are meaningless in this context and are only kept for compatibility[3].

- `GetFlatMINLPConstraintInfo` (see A.4.5.8);

  Notes: The arguments `string cname`, `si* index`, `si* cilist`, `sd* cflist`, `FlatExpression* fe` are meaningless in this context and are only kept for compatibility[4].

[3]They can simply be skipped in the call.
[4]They can simply be skipped in the call.

- `GetFlatMINLPVariableValues` (see A.4.5.6);

  Notes: Call is the same as in section A.4.5.6.

- `SetFlatMINLPVariableValues` (see A.4.5.7);

  Notes: Call is the same as in section A.4.5.7.

- `SetFlatMINLPVariableBounds` (see A.4.5.5);

  Notes: Call is the same as in section A.4.5.5.

- `EvalFlatMINLPNonlinearObjectiveFunction` (see A.4.5.9);

  Notes: This is for compatibility only; it returns 0.

- `EvalFlatMINLPNonlinearConstraint` (see A.4.5.10);

  Notes: This is for compatibility only; it does not do anything.

- `GetFlatMINLPObjectiveFunctionDerivatives` — linear objective function only (see A.4.5.11);

  Notes: Call is the same as in section A.4.5.3.

- `GetFlatMINLPConstraintDerivatives` — linear constraints only (see A.4.5.12).

  Notes: Call is the same as in section A.4.5.3.

A.6.2 The `FlatExpression` interface

This object class has the purpose of conveying symbolic flat expression information to the client. A `Flat-Expression` object is returned by the `GetFlatMINLPConstraintInfo` method (see section A.4.5.8) and is then transformed, according to its type, to any of the derived objects `FlatVariableExpression`, `Flat-ConstantExpression`, `FlatOperatorExpression`. Each of these objects provides a special interface used to access the encapsulated data.

The `FlatExpression` interface has only one method.

A.6.2.0.1 Method `GetKind`

Declaration: `int GetKind()`

Function: Returns the kind of this `FlatExpression`.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value On Exit
return value	`int`	kind of `FlatExpression`

Notes:

- The value returned by this method can be one of `FlatConstantType`, `FlatVariableType`, `Flat-OperatorType` according as to whether the current `FlatExpression` is respectively one of `Flat-ConstantExpression`, `FlatVariableExpression`, `FlatOperatorExpression`.

Examples of usage:

See below for a comprehensive example of all the methods relative to `FlatExpression`.

A.6.2.1 The `FlatConstantExpression` interface

This object class is derived from the `FlatExpression` class and provides flat symbolic information about constant expression objects created with the `NewConstantExpression` method (see section A.4.2.6).

This interface has only one method.

A.6.2.1.1 Method `GetValue`

Declaration: `double GetValue()`

Function: Returns the value of the flat symbolic constant expression.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value On Exit
return value	`double`	value of flat symbolic constant

Notes: None

Examples of usage:

See below for a comprehensive example of all the methods relative to `FlatExpression`.

A.6.2.2 The `FlatVariableExpression` interface

This object class is derived from the `FlatExpression` class and provides flat symbolic information about variable expression objects created with the `NewVariableExpression` method (see section A.4.2.7).

This interface has only one method.

A.6.2.2.1 Method `GetVarIndex`

Declaration: `double GetVarIndex()`

Function: Returns the variable index (or variable id, a.k.a. `vid`) of the flat symbolic variable expression.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value On Exit
return value	`double`	value of flat symbolic constant

Notes: None

Examples of usage:

See below for a comprehensive example of all the methods relative to `FlatExpression`.

A.6.2.3 The `FlatOperatorExpression` interface

This object class is derived from the `FlatExpression` class and provides flat symbolic information about expression objects created with the `UnaryExpression` and `BinaryExpression` methods (see sections A.4.2.9 and A.4.2.8).

This interface has two methods.

A.6.2.3.1 Method `GetOperator`

Declaration: `int GetOperator()`

Function: Returns the operator type of the flat symbolic unary or binary expression.

Arguments to be specified by the client: None

Arguments returned to client:

Argument	Type	Value On Exit
return value	`int`	operator type

Notes:

- The return function value describes the type of operator in the expression. It can be one of the following strings (the meaning is self-explanatory): `"sum"`, `"difference"`, `"product"`, `"ratio"`, `"power"`, `"minus"`, `"log"`, `"exp"`, `"sin"`, `"cos"`, `"tan"`, `"cot"`, `"sinh"`, `"cosh"`, `"tanh"`, `"coth"` (cf. section A.2.4).

Examples of usage:

See below for a comprehensive example of all the methods relative to `FlatExpression`.

A.6.2.3.2 Method `GetOperand`

Declaration: `FlatExpression* GetOperand(int eid)`

Function: Returns one of the operands of the flat symbolic unary or binary expression.

Arguments to be specified by the client:

Argument	Type	Specified on Entry
eid	`int`	the operand expression id

Arguments returned to client:

Argument	Type	Value On Exit
return value	`FlatExpression*`	flat symbolic operand expression

Notes:

- The operand expression id, `eid`, can only be 1 for unary expressions; it can be 1 or 2 for binary expressions.
- The return value is placed into an object of type `FlatExpression`. This can be analysed with the methods described in section A.6.2.

Examples of usage:

See below for a comprehensive example of all the methods relative to `FlatExpression`.

A.6.2.4 Usage of `FlatExpression` interface

The procedure to gather symbolic expression information from a `FlatExpression` is as follows.

1. Find out the `FlatExpression` type; there are three possible types: `Constant, Variable` and `Operator`.

   ```
   int fetype = fe->GetKind();
   ```

2. Depending on the type, cast the `FlatExpression` object dynamically to one of the following objects: `FlatConstantExpression, FlatVariableExpression` and `FlatOperatorExpression`.

   ```
   FlatConstantExpression* fke;
   FlatVariableExpression* fve;
   FlatOperatorExpression* foe;
   switch(fetype) {
   case Constant:
     fke = dynamic_cast<FlatConstantExpression*>(fe);
     break;
   case Variable:
     fve = dynamic_cast<FlatVariableExpression*>(fe);
     break;
   case Operator:
     foe = dynamic_cast<FlatOperatorExpression*>(fe);
     break;
   }
   ```

3. Now find out the actual information.
   - The `FlatConstantExpression` interface has only one method:

     ```
     double GetValue(void);
     ```

     which returns the actual value of the constant.
   - The `FlatVariableExpression` interface has only one method:

     ```
     long int GetVarIndex(void);
     ```

     which returns the flat variable index of the variable.
   - The `FlatOperatorExpression` interface has two methods: the first,

     ```
     int GetOpType(void);
     ```

     returns the type of operator of the expression (possible values are as on page 146); the second method,

     ```
     FlatExpression* GetFlatExpression(int index);
     ```

     returns an operand of the operator given by `GetOpType()`. In the case of binary operators `index` can be 0 (for the left operand) or 1 (for the right operand). In the case of unary operators `index` can only be zero.

4. The symbolic analysis of the expression can go on in a recursive fashion until no more `FlatOperator-Expression`s are found.
5. It is important to notice that the deallocation of all the `FlatExpression` objects is left to the client.

The following is the actual coded example:

```
string          cname   ;
si*             index   ;
double          LB     ;
double          UB     ;
si*             varList ;
sd*             coefList;
FlatExpression* fe;

NetOpt->GetFlatMINLPConstraintInfo(532, cname, index, LB, UB,
                                   varList, coefList, fe)

FlatConstantExpression* fke;
FlatVariableExpression* fve;
FlatOperatorExpression* foe;
switch(fe->GetKind()) {
case Constant:
  fke = dynamic_cast<FlatConstantExpression*>(fe);
  cout << "Constraint Nonlinear Part is a Constant" << endl;
  cout << "Value = " << fke->GetValue() << endl;
  break;
case Variable:
  fve = dynamic_cast<FlatVariableExpression*>(fe);

  cout << "Constraint Nonlinear Part is a Variable" << endl;
  cout << "VarIndex = " << fve->GetVarIndex() << endl;
  break;
case Operator:
  foe = dynamic_cast<FlatOperatorExpression*>(fe);
  cout << "Constraint Nonlinear Part is an Operator" << endl;
  cout << "Operator Type = " << foe->GetOpType() << endl;
  break;
}
delete fe;
```

A.7 Implementation restrictions

The following restrictions of the current implementation in comparison to the functionality described in this document are known:

1. *Limited number of arguments to* `IntSeq` *function*
   The `IntSeq` auxiliary function can have a maximum of 8 integer arguments.
2. *Currently available MINLP Managers*
   The following MINLP solvers have been interfaced to date to *ooOPS* and can be used in conjunction with the function `NewMINLPSolverManager` (see section A.5.3):
   - SNOPT v. 5.3 (Systems Optimization Laboratory, Stanford University.)
     Accessed by specifying `sname` = `"snopt"`

A.8 An Example of the use of *ooOPS*

The example is based on a slightly simplified form of the Resource Task Network (RTN) formulation proposed by C. Pantelides for process scheduling, plus a few spurious nonlinear terms which mess up the model hopelessly and completely, but help show how to use the methods which deal with nonlinearity. The formulation seeks to optimise a process involving NR resources $r = 1, .., NR$ and NK tasks $k = 1, .., NK$ over a time horizon discretised into NT time intervals $t = 1, .., NT$. It involves the objective function:

$$\max \sum_r (C_r^F (R_{r,NT} - R_{r0}) + \sum_t R_{rt}^2) \tag{A.7}$$

subject to the constraints:

$$R_{rt} = R_{r,t-1} + \sum_k \sum_{\theta=0}^{\tau_k} (\mu_{kr\theta} N_{k,t-\theta} + \nu_{kr\theta} \xi_{k,t-\theta}) + \Pi_{rt} + \frac{R_{rt} R_{r,t-1}}{\Pi_{rt}} \quad \forall r, t \tag{A.8}$$

$$0 \leq R_{rt} \leq R_{rt}^{\max} \quad \forall r, t \tag{A.9}$$

$$V_k^{\min} N_{kt} \leq \xi_{kt} \leq V_k^{\max} N_{kt} \quad \forall k, t \tag{A.10}$$

The variables in the above formulation are the following:

Variable	Type	Range	Description
R_{rt}	Continuous	$r = 1, .., NR$	Amount of resource r at time t
		$t = 0, .., NT$	
N_{kt}	Integer	$k = 1, .., NK$	Number of units used for task k at time t
		$t = 1, .., NR$	
ξ_{kt}	Continuous	$k = 1, .., NK$	Size of task k at time t
		$t = 1, .., NT$	

Table A.1: Variable sets

A number of parameters and coefficients also appear in the formulation. These are listed in Table A.2. The initial values of the resource levels, R_{r0}, are also fixed at given values R_r^*.

Parameter	Type	Description
C_r^f	Constant	Unit cost of resource r
μ_{rt}	Constant	Production or consumption coefficient referring to integer variable N_{kt}
ν_{kt}	Constant	Production or consumption coefficient referring to continuous variable ξ_{kt}
Π_{rt}	Constant	Amount of resource r made available from/to external sources at time t
τ_k	Constant	Duration of task k
$R_{rt}^{\max}$	Constant	Maximum amount of resource r that can be stored at time t
$V_k^{\min}$	Constant	Minimum useful capacity of unit suitable for task k
$V_k^{\max}$	Constant	Maximum useful capacity of unit suitable for task k

Table A.2: Parameters appearing in the RTN formulation

A.8.1 Creating the MINLP

The first step is to create (an empty) `ops` object:

```
ops* RTNops=NewMINLP();
```

Now we have to add to this object (`RTNops`) the information that defines it. This is done below using the methods described in section A.4.

A.8.1.1 Creating variables

Variables are added to the `ops` object using the variable construction methods described in section A.4.2.

Continuous variables are created using the method `NewContinuousVariable` in the way specified in section A.4.2.1:

```
RTNops->NewContinuousVariable("Xi", IntSeq(1,1), IntSeq(NK,NT),
                              0.0, ooOPSPlusInfinity, 0.0);
RTNops->NewContinuousVariable("R", IntSeq(1,0), IntSeq(NR,NT),
                              0.0, ooOPSPlusInfinity, 0.0);
```

while integer variables are created using the method `NewIntegerVariable` described in section A.4.2.2:

```
RTNops->NewIntegerVariable("N", IntSeq(1,1), IntSeq(NK,NT),
                            0, ooOPSPlusInfinity, 0);
```

All variables are two-dimensional, their lower bounds are initialised to zero and so are their default values. No upper bounds are imposed at this stage.

A.8.1.2 Creating constraints

We now use the construction method presented in section A.4.2.3 to create constraints (A.8) and (A.10):

```
RTNops->NewConstraint("ResourceBalance",IntSeq(1,1),
                      IntSeq(NR,NT), 0.0, 0.0);
RTNops->NewConstraint("EquipmentCapacityLB",IntSeq(1,1),
                      IntSeq(NK,NT), 0.0, ooOPSPlusInfinity);
RTNops->NewConstraint("EquipmentCapacityUB",IntSeq(1,1),
                      IntSeq(NK,NT), ooOPSMinusInfinity, 0.0);
```

We note that constraint A.8 has been rearranged to the form:

$$\begin{aligned} -R_{rt} + R_{r,t-1} + \sum_k \sum_{\theta=0}^{\tau_k} (\mu_{kr\theta} N_{k,t-\theta} + \nu_{kr\theta} \xi_{k,t-\theta}) + \\ +\Pi_{rt} + \frac{R_{rt} R_{r,t-1}}{\Pi_{rt}} = 0 \quad \forall r, t \end{aligned} \tag{A.8$'$}$$

while A.10 has been split into two separate constraints of the form:

$$\xi_{kt} - V_k^{\min} N_{kt} \geq 0 \quad \forall k, t \tag{A.10$'$}$$

and

$$\xi_{kt} - V_k^{\max} N_{kt} \leq 0 \quad \forall k, t \tag{A.10$''$}$$

On the other hand, no constraint corresponding to (A.9) is introduced since this can be dealt with via upper bounds imposed on the R_{rt} variables (see section A.8.1.6).

A.8.1.3 Adding variables to constraints

The constraints created in section A.8.1.2 do not yet contain any variables. We now have to create appropriate variable occurrences in them by applying the method `AddVariableSliceToConstraintSlice` as described in section A.4.2.4 [5]:

- Constraint (A.8$'$)

```
for(int r=1 ; r<=NR ; r++){
  for(int t=1 ; t<=NT ; t++){
    RTNops->AddVariableSliceToConstraintSlice("R",
             IntSeq(r,t), IntSeq(r,t), "ResourceBalance",
             IntSeq(r,t),IntSeq(r,t), -1.0);
    RTNops->AddVariableSliceToConstraintSlice("R",
             IntSeq(r,t-1),IntSeq(r,t-1),"ResourceBalance",
             IntSeq(r,t),IntSeq(r,t), 1.0);
    for(int k=1 ; k<=NK ; k++){
      for(int theta = 0;
          theta <= min(tau(k), (double) t - 1);
          theta++){
        RTNops->AddVariableSliceToConstraintSlice("N",
                 IntSeq(k,t-theta), IntSeq(k,t-theta),
                 "ResourceBalance",IntSeq(r,t),IntSeq(r,t),
                 mu(k,r,theta));
        RTNops->AddVariableSliceToConstraintSlice("Xi",
                 IntSeq(r,t-theta), IntSeq(r,t-theta),
                 "ResourceBalance",IntSeq(r,t),IntSeq(r,t),
                 nu(k,r,theta));
      }
    }
  }
}
```

- Constraint (A.10$'$)

```
  for(int k=1 ; k<=NK ; k++){
    for(int t=1 ; t<=NT ; t++){
RTNops->AddVariableSliceToConstraintSlice("Xi",
                 IntSeq(k,t), IntSeq(k,t),
                 "EquipmentCapacityLB",
```

[5]Here we assume that appropriate sets holding data components to the parameters of table A.2 are already available.

```
                    IntSeq(k,t),IntSeq(k,t), 1.0);
    RTNops->AddVariableSliceToConstraintSlice("N",
                    IntSeq(k,t), IntSeq(k,t),
                    "EquipmentCapacityLB",
                    IntSeq(k,t),IntSeq(k,t),
                    -Vmin(k,t));
        }
      }
```

- Constraint (A.10′)

```
      for(int k=1 ; k<=NK ; k++){
        for(int t=1 ; t<=NT ; t++){
    RTNops->AddVariableSliceToConstraintSlice("Xi",
                    IntSeq(k,t), IntSeq(k,t),
                    "EquipmentCapacityUB",
                    IntSeq(k,t),IntSeq(k,t), 1.0);
    RTNops->AddVariableSliceToConstraintSlice("N",
                    IntSeq(k,t), IntSeq(k,t),
                    "EquipmentCapacityUB",
                    IntSeq(k,t),IntSeq(k,t),
                    -Vmax(k,t));
        }
      }
```

A.8.1.4 Objective function

The creation of the objective function is effected using the method `NewObjectiveFunction` (cf. section A.4.2.11):

```
RTNops->NewObjectiveFunction("TotalProfit","max");
```

For this example, the name of the objective function to be maximized is `"TotalProfit"` .

A.8.1.5 Objective function coefficients

The creation of the objective function is followed by the declaration of the coefficients of the variable instances appearing in it. The appropriate method is `AddVariableSliceToObjectiveFunction` described in section A.4.2.12:

```
  for(int r=1 ; r<=NR ; r++){
    RTNops->AddVariableSliceToObjectiveFunction("R",IntSeq(r,NT),
             IntSeq(r,NT), CFR(r),"TotalResources");
    RTNops->AddVariableSliceToObjectiveFunction("R",IntSeq(r,0),
             IntSeq(r,0), -CFR(r),"TotalResources");
  }
```

Again, we assume the coefficients C_r^F that appear in the objective function are stored in set `CFR`.

A.8.1.6 Modifying the variable bounds

Variable R_{rt} is bounded as shown in eqn. (A.9). Method `SetVariableBounds` can be used to impose these bounds which may be different for different elements of the R_{rt} set:

```
for(int r=1 ; r<=NR ; r++){
  for(int t=1 ; t<=NT ; t++){
    RTNops->SetVariableBounds("R",IntSeq(r,t),IntSeq(r,t),
                                 0.0, R_max[r][t]);
  }
}
```

Moreover, the initial resource levels R_{r0} are fixed at given values R^*_r. This is achieved by setting both lower and upper bounds to R^*_r:

```
for(int r=1 ; r<=NR ; r++){
  RTNops->SetVariableBounds("R",IntSeq(r,0),IntSeq(r,0),
                               R_star[r], R_star[r]);
}
```

A.8.1.7 Modifying the constraint bounds

Constraint (A.8) has a right hand coefficient which is not constant so we use the function `SetConstraintBounds` to obligate this restriction.

```
for(int r=1 ; r<=NR ; r++){
  for(int t=1 ; t<=NT ; t++){
    RTNops->SetConstraintBounds("ResourceBalance",
                                  IntSeq(r,t), IntSeq(r,t),
                                  -Pi[r][t], -Pi[r][t]);
  }
}
```

A.8.1.8 Creating the nonlinear parts

We create nonlinear parts of constraints and objective function by employing the methods described in section A.4.2.6.

```
// make nonlinear part of constraints
RTNops->NewVariableExpression("Leaf1", "R",
                              IntSeq(1,1), IntSeq(NR, NT));
RTNops->NewVariableExpression("Leaf2", "R",
                              IntSeq(1,1), IntSeq(NR, NT));
RTNops->BinaryExpression("Expr1",
                         "Leaf1", IntSeq(1,1), IntSeq(NR, NT),
                         "Leaf2", IntSeq(1,1), IntSeq(NR, NT),
                         "product");
RTNops->NewConstant("Const1", IntSeq(1,1), IntSeq(NR, NT), 1.0);
for(int r=1 ; r<=NR ; r++){
  for(int t=1 ; t<=NT ; t++){
    RTNops->SetConstantValue("Const1", IntSeq(r,t), 1/Pi[r][t]);
  }
}
RTNops->NewConstantExpression("Leaf3", "Const1",
                              IntSeq(1,1), IntSeq(NR, NT));
RTNops->BinaryExpression("Expr1",
                         "Expr1", IntSeq(1,1), IntSeq(NR,NT),
                         "Leaf3", IntSeq(1,1), IntSeq(NR,NT),
                         "ratio");

// make nonlinear part of objective function
RTNops->NewConstant("Const2", IntSeq(1,1), IntSeq(1,1), 2.0);
RTNops->NewConstantExpression("Power2", "Const2",
                              IntSeq(1,1), IntSeq(1,1));
RTNops->BinaryExpression("Expr2",
                         "Leaf1", IntSeq(1,1), IntSeq(NR,NT),
                         "Const2", IntSeq(1,1), IntSeq(1,1),
                         "power");
RTNops->NewConstant("Zero", IntSeq(1), IntSeq(1), 0);
RTNops->NewConstantExpression("Expr3", "Zero",
                              IntSeq(1), IntSeq(1));
for(int r=1; r<=NR; r++) {
  for(int t=1; t<= NT; t++) {
     RTNops->BinaryExpression("Expr3",
                              "Expr3", IntSeq(1), IntSeq(1),
                              "Expr2", IntSeq(r,t), IntSeq(r,t),
                              "sum");
  }
}
```

A.8.1.9 Assigning expressions to constraints and objective function

After having created the expressions representing the nonlinear parts, we assign them to the existing constraints and objective function by using the methods described in section A.4.2.7.

```
  // assign expressions to constraints
  RTNops->AssignExpressionSliceToConstraintSlice
          ("Expr1", IntSeq(1,1), IntSeq(NR, NT),
           "ResourceBalance", IntSeq(1,1), IntSeq(NR, NT));

  // assign expression to objective function
  RTNops->AssignExpressionToObjectiveFunction
          ("Expr3", "TotalProfit");
```

A.8.2 MINLP solution

Having created the `ops` object `RTNops`, we now have to combine it with an appropriate MINLP solver to create an `opssystem` that can be solved.

A.8.2.1 Creating an MINLP solver manager object

We start by creating an appropriate `opssolvermanager` object. The usage of this method is described in section A.5.3. Here, we create a MINLP solver manager based on the SNOPT solver:

```
opssolvermanager* SnoptManager = NewMINLPSolverManager("snopt");
```

A.8.2.2 Creating an MINLP system

Using the MINLP solver manager object created above, a `opssystem` is created from the `ops` object:

```
opssystem* RTNsystem = SnoptManager->NewMINLPSystem(RTNops);
```

A.8.2.3 Solving the MINLP

At last, the above `MINLPsystem` can be solved by invoking its `Solve` method as described in section A.5.5.4:

```
RTNsystem->Solve();
```

A.8.3 Accessing the Solution of the MINLP

Various aspects of the MINLP solution can be accessed using methods described in section A.4.4.

A.8.3.1 Obtaining information on the variables

The optimal values of the variables can be obtained using method A.4.4.1. In this example we have three types of variables:

- R_{rt}

  ```
  double* R_value;
  double* R_LB;
  double* R_UB;

  RTNops->GetVariableInfo("R",IntSeq(1,1),IntSeq(NR,NT),
                              R_value, R_LB, R_UB);
  ```

- N_{kt}

  ```
  int* N_value;
  int* N_LB;
  int* N_UB;

  RTNops->GetVariableInfo("N",IntSeq(1,1),IntSeq(NK,NT),
                              N_value, N_LB, N_UB);
  ```

- ξ_{kt}

```
      double* Xi_value;
      double* Xi_LB;
      double* Xi_UB;

      RTNops->GetVariableInfo("Xi",IntSeq(1,1),IntSeq(NK,NT),
                              Xi_value, Xi_LB, Xi_UB);
```

A.8.3.2 Obtaining information on the objective function

The value of the objective function can be accessed using the method of section A.4.4.3:

```
  char*   obj_type;
  double* obj_value;

  RTNops->GetObjectiveFunctionInfo("TotalProfit",
                                   objt_type, obj_value);
```

The value of the objective function returned is based on the current (hopefully optimal) values of the variables.

Index