EFFICIENT AND SAFE GLOBAL CONSTRAINTS FOR HANDLING NUMERICAL CONSTRAINT SYSTEMS

YAHIA LEBBAH†, CLAUDE MICHEL‡¶, MICHEL RUEHER‡¶, DAVID DANEY§¶, AND JEAN-PIERRE MERLET§¶

Abstract. Numerical constraint systems are often handled by branch and prune algorithms that combine splitting techniques, local consistencies and interval methods. This paper first recalls the principles of Quad, a global constraint that works on a tight and safe linear relaxation of quadratic subsystems of constraints. Then, it introduces a generalisation of Quad to polynomial constraint systems. It also introduces a method to get safe linear relaxations and shows how to compute safe bounds of the variables of the linear constraint system. Different linearisation techniques are investigated to limit the number of generated constraints. QuadSolver, a new branch and prune algorithm that combines Quad, local consistencies and interval methods is introduced. QuadSolver has been evaluated on a variety of benchmarks from kinematics, mechanics and robotics. On these benchmarks, it outperforms classical interval methods as well as CSP solvers and it compares well with state-of-the-art optimisation solvers.

Key words. systems of equations and inequalities, constraint programming, reformulation linearisation technique (RLT), global constraint, interval arithmetic, safe rounding.

AMS subject classifications. 65H10, 65G40, 65H20, 93B18, 65G20.

1. Introduction. Many applications in engineering sciences require finding all isolated solutions to systems of constraints over real numbers. These systems may be non-polynomial and are difficult to solve: the inherent computational complexity is NP-hard and numerical issues are critical in practice (e.g., it is far from being obvious to guarantee correctness and completeness as well as to ensure termination). These systems, called numerical CSP in the rest of this paper, have been approached in the past by different interesting methods: interval methods [35, 24, 38, 20, 40], continuation methods [37, 2, 62] and constraint satisfaction methods [30, 6, 11, 61]. Of particular interest is the mathematical and programming simplicity of the latter approach: the general framework is a branch and prune algorithm that only requires specifying the constraints and the initial range of the variables.

The purpose of this paper is to introduce and to study a new branch and bound algorithm called QuadSolver. The essential feature of this algorithm is a global constraint –called Quad– that works on a tight and safe linear relaxation of the polynomial relations of the constraint systems. More precisely, QuadSolver is a branch and prune algorithm that combines Quad, local consistencies and interval methods. That is to say, QuadSolver is an attempt to merge the best interval and constraint programming techniques. QuadSolver has been evaluated on a variety of benchmarks from kinematics, mechanics and robotics. On these benchmarks, it outperforms classical interval methods as well as CSP solvers and it compares well with state-of-the-art optimisation solvers.

The Quad-filtering algorithm [27] has first been defined for quadratic constraints. The relaxation of quadratic terms is adapted from a classical linearisation method, the “Reformulation-Linearisation Technique (RLT)” [54, 53]. The simplex algorithm is used to narrow the domain of each variable with respect to the subset of the linear set of constraints generated by the relaxation.
process. The coefficients of these linear constraints are updated with the new values of the bounds of the domains and the process is restarted until no more significant reduction can be done. We have demonstrated [27] that the Quad algorithm yields a more effective pruning of the domains than local consistency filtering algorithms (e.g., 2b-consistency [30] or box-consistency [6]). Indeed, the drawback of classical local consistencies comes from the fact that the constraints are handled independently and in a blind way. That is to say, classical local consistencies do not exploit the semantic of quadratic terms; in other words, these approaches do not take advantage of the very specific semantic of quadratic constraints to reduce the domains of the variables. Conversely, linear programming techniques [1, 54, 3] do capture most of the semantics of quadratic terms e.g., convex and concave envelopes of these particular terms.

The extension of Quad for handling any polynomial constraint system requires to replace non-quadratic terms by new variables and to add the corresponding identities to the initial constraint system. However, a complete quadrification [58] would generate a huge number of linear constraints. Thus, we introduce here an heuristic based on a good tradeoff between a tight approximation of the nonlinear terms and the size of the generated constraint system. This heuristic works well on classical benchmarks (see § 8).

A safe rounding process is a key issue for the Quad framework. Let’s recall that the simplex algorithm is used to narrow the domain of each variable with respect to the subset of the linear set of constraints generated by the relaxation process. The point is that most implementations of the simplex algorithm are unsafe. Moreover, the coefficients of the generated linear constraints are computed with floating point numbers. So, two problems may occur in the Quad-filtering process:

1. The whole linearisation may become incorrect due to rounding errors when computing the coefficients of the generated linear constraints;
2. Some solutions may be lost when computing the bounds of the domains of the variables with the simplex algorithm.

We propose in this paper a safe procedure for computing the coefficients of the generated linear constraints. Neumaier and Shcherbina [42] have addressed the second problem. They have proposed a simple and cheap procedure to get a rigorous upper bound of the objective function. The incorporation of these procedures in the Quad-filtering process allows us to call the simplex algorithm without worrying about safety. So, with these two procedures, linear programming techniques can be used to tackle continuous CSPs without losing any solution.

The rest of this paper is organised as follows. Section 2 gives an overview of the approach whereas Section 3 contains the notation. Sections 4 and 5 recall the basic of interval programming and constraint programming. Section 6 details the principle of the Quad algorithm, the linearisation process and the extension to polynomial constraints. Section 7 introduces the rounding process we propose to ensure the safe relaxations. Section 8 describes the experimental results and discusses related work.

---

3 3b-consistency and kb-consistency are partial consistencies that can achieve a better pruning since they are “less local” [11]. However, they require numerous splitting steps to find the solutions of a system of quadratic constraints; so, they may become rather slow.

4 Sherali and Tuncbilek [55] have also proposed four different filtering techniques for solving quadratic problems. Roughly speaking, the first filtering strategy performs a feasibility check on inequality constraints to discard subintervals of the domains of the variables. This strategy is very close to Box-consistency filtering (see [60]). The three other techniques are based on specific properties of optimisation problems with a quadratic objective function: the eigenstructure of the quadratic objective function, fathoming node and Lagrangian dual problem. Thus, these techniques can be considered as local consistencies for optimisation problems (see also [59] and Neumaier’s survey [41]).

5 They also suggest a solution to the first problem though their solution is dedicated to MIP problems.
2. Overview of the approach. As mentioned, QuadSolver is a branch and prune algorithm that combines Quad and a box-consistency. Box-consistency is the most successful adaptation of arc-consistency [31] to constraints over the real numbers. The box-consistency implementation of P. Van Hentenryck et al. [60] is computed on three interval extensions of the initial constraints: the natural interval extension, the distributed interval extension, and the Taylor interval extension with a conditioning step. The leftmost and the rightmost zero are computed using a variation of the univariate interval Newton method.

The QuadSolver we propose here combines Quad-filtering and box-consistency filtering to prune the domain of the variables of numerical constraint systems. Operationally, QuadSolver performs the following filtering process:

1. box-consistency filtering,
2. Quad-filtering.

The box-consistency is first used to detect some inconsistencies before starting the Quad filtering algorithm which is more costly. These two steps are wrapped into a classical fixed point algorithm which stops when the domains of the variables can not be further reduced\(^6\).

To isolate the different solutions, Quad uses classical branching techniques.

Before going into the details, let us outline the advantages of our approach on a couple of small examples.

2.1. Quad-filtering.
Consider the constraint system \( C = \{2xy + y = 1, xy = 0.2\} \) which represents two intersecting curves (see figure 2.1). Suppose that \( x = [-10, +10] \) and \( y = [-10, +10] \) are the domains of the variables \( x \) and \( y \). An interval \( x = [\underline{x}, \overline{x}] \) denotes the set of reals \( \{r | \underline{x} \leq r \leq \overline{x}\} \).

![Fig. 2.1. Geometrical representation of \( \{2xy + y = 1, xy = 0.2\} \)](image)

The reformulation-linearisation technique (see § 6.2) yields the following constraint system:

\[
\begin{align*}
(a) \quad &y + 2w = 1, \ w = 0.2, \\
&\frac{y}{x} + \frac{y}{x} - w \leq \frac{x}{y}, \ \frac{x}{y} + \frac{x}{y} - w \geq \frac{x}{y}, \\
&\frac{x}{y} + \frac{x}{y} - w \geq \frac{x}{y}, \ \frac{x}{y} + \frac{x}{y} - w \leq \frac{x}{y},
\end{align*}
\]

where \( w \) is a new variable that stands for the product \( xy \). Note that constraint system \((a)\) implies that \( w \in [\underline{x}, \overline{x}] \times [\underline{y}, \overline{y}] \).

Substituting \( \underline{x}, \overline{x}, \underline{y} \) and \( \overline{y} \) by their values and minimising (resp. maximising) \( x, y \) and \( w \) with the simplex algorithm yields the following new bounds:

\(^6\)In practice, the loop stops when the domain reduction is lower than a given \( \epsilon \).
\[ x = [-9.38, 9.42], \ y = [0.6, 0.6], \ w = [0.2, 0.2]. \]

By substituting the new bounds of \( x \), \( y \) and \( w \) in the constraint system \((a)\), we obtain a new linear constraint system. One more minimising (resp. maximising) step is required to obtain tight bounds of \( x \). Note that numerous splitting operations are required to find the unique solution of the problem with a 3b-consistency filtering algorithm. The proposed algorithm solves the problem by generating 6 linear constraints and with 8 calls to the simplex algorithm. It finds the same solution as a solver based on 3b-consistency but without splitting and in less time.

\[ \mathcal{C} = \left\{ \begin{array}{ll}
 w_1 + w_2 = 1, & w_1x_1 + w_2x_2 = 0, \\
 w_1x_1 + w_2x_2x_2 = 1, & w_1x_1x_1 + w_2x_2x_2 = 0
\end{array} \right\} \]

which represents a simple Gaussian quadrature formula to compute integrals [9]. Suppose that the domains of variables \( x_1 \), \( x_2 \), \( w_1 \) and \( w_2 \) are all equal to \([-1, 1]\). This system has two solutions:

- \( x_1 = -1, \ x_2 = 1, \ w_1 = 0.5, \ w_2 = 0.5 \).
- \( x_1 = 1, \ x_2 = -1, \ w_1 = 0.5, \ w_2 = 0.5 \).

A straightforward implementation of Quad would only find one unsafe solution with:

\[ x_2 \in [+0.999999944, +0.999998982] \]

Indeed, when we examine the Quad-filtering process, we can identify some LPs where the simplex algorithm steps to the wrong side of the objective.

With the corrections we propose in §7, we obtain a tight approximation of the two correct solutions (with \( x_2 \in [-1.000000000, -0.999999998] \) and \( x_2 \in [0.999999999, 1.000000000] \)).

### 3. Notation and basic definitions.

This paper focuses on CSPs where the domains are intervals and the constraints \( C_j(x_1, \ldots, x_\ell) \) are \( n \)-ary relations over the reals. \( \mathcal{C} \) stands for the set of constraints.

- \( x \) or \( D_x \) denotes the domain of variable \( x \), that is to say, the set of allowed values for \( x \). \( D \) stands for the set of domains of all the variables of the considered constraint system. \( \mathcal{F} \) denotes the set of real numbers whereas \( \mathcal{F} \) stands for the set of floating point numbers used in the implementation of nonlinear constraint solvers; if \( a \) is a constant in \( \mathcal{F} \), \( a^+ \) (resp. \( a^- \)) corresponds to the smallest (resp. largest) number of \( \mathcal{F} \) strictly greater (resp. lower) than \( a \).

- \( X = [x, \overline{x}] \) is defined as the set of real numbers \( x \) verifying \( x \leq x \leq \overline{x} \). \( x, y \) denote real variables, \( X, Y \) denote vectors whereas \( X, Y \) denote interval vectors. The width \( w(x) \) of an interval \( x \) is the quantity \( \overline{x} - \underline{x} \) while the midpoint \( m(x) \) of the interval \( x \) is \((\overline{x} + \underline{x})/2 \). A point interval \( x \) is obtained if \( \underline{x} = \overline{x} \). A box is a set of intervals: its width is defined as the largest width of its interval members, while its centre is defined as the point whose coordinates is the midpoint of the ranges.

\( \text{II} \mathcal{F}^\ell \) denotes the set of boxes and is ordered by set inclusion.

We use the “reformulation-linearisation technique” (RLT) notation introduced in [54, 3] with slight modifications. More precisely, we will use the following notations: \([c]_L \) is the set of linear constraints generated by replacing the nonlinear terms by new variables in constraint \( c \) and \([c]_R \) denotes the set of equations that keep the link between the new variables and the nonlinear terms while \([c]_R \) contains linear inequalities that approximate the semantics of nonlinear terms of constraint \( c \). These notations will be used indifferently whether \( c \) is a constraint or \( \mathcal{C} \) is a set of constraints.

Rounding is necessary to close the operations over \( \mathcal{F} \) [18]. A rounding function maps the result of the evaluation of an expression to available floating-point numbers. Rounding \( x \) towards \(+\infty\) maps \( x \) to the least floating point number \( x_f \) such that \( x \leq x_f \). \( \nabla(x) \) (resp. \( \nabla(x) \)) denotes a rounding mode of \( x \) towards \( -\infty \) (resp. \(+\infty\)).
4. Interval Programming. This section recalls the basic concepts of interval arithmetic that are required to understand the rest of the paper. Readers familiar with interval arithmetic may skip this section.

4.1. Interval Arithmetic. Interval arithmetic has been introduced by Moore [35]. It is based on the representation of variables as intervals.

Let \( f \) be a real-valued function of \( n \) unknowns \( X = (x_1, \ldots, x_n) \). An interval evaluation of \( f \) for given ranges \( X = (x_1, \ldots, x_n) \) for the unknowns is an interval \( y \) such that

\[
\underline{y} \leq f(X) \leq \overline{y}, \quad \text{for all} \quad X = (x_1, \ldots, x_n) \in X = (x_1, \ldots, x_n)
\]  

(4.1)

In other words, \( y, \overline{y} \) are lower and upper bounds for the values of \( f \) when the values of the unknowns are restricted to the box \( X \).

There are numerous ways to calculate an interval evaluation of a function [20, 46]. The simplest is the natural evaluation in which all the mathematical operators in \( f \) are substituted by their interval equivalents. Interval equivalents exist for all classical mathematical operators. Hence interval arithmetic allows to calculate an interval evaluation for all nonlinear expressions, whether algebraic or not. For example, if \( f(x) = x + \sin(x) \), then the interval evaluation of \( f \) for \( x \in [1.1, 2] \) can be calculated as follows:

\[
f([1.1, 2]) = [1.1, 2] + \sin([1.1, 2]) = [1.1, 2] + [0.8912, 1] = [1.9912, 3]
\]

Interval arithmetic can be implemented with directed rounding to take into account round-off errors. There are numerous interval arithmetic packages implementing this property: one of the most famous library is BIAS/Profi\(^7\) but a promising new package — based on the multi-precision software MPFR\(^8\) — is MPFI [47].

The main limitation of interval arithmetic is the over-estimation of interval functions. This is due to two well-known problems:

- the so-called wrapping effect [35, 39], which overestimates by a unique vector the image of an interval vector (which is in general not a vector). That is to say, \( \{ f(X) \mid X \in X \} \) is contained in \( f(X) \) but is usually not equal to \( f(X) \).
- the so-called dependency problem [20], which is due to the independence of the different occurrences of some variables during the interval evaluation of an expression. In other words, during the interval evaluation process there is no correlation between the different occurrences of a same variable in an equation. For instance, consider \( x = [0, 10] \), \( x - x = [x, \overline{x} - \underline{x}] = [-10, 10] \) instead of \([0, 0]\) as one could expect.

In general, it is not possible to compute the exact enclosure of the range for an arbitrary function over the real numbers [25]. Thus, Moore introduced the concept of interval extension: the interval extension of a function is an interval function that computes outer approximations on the range of the function over a domain [20, 36]. Two main extensions have been introduced: the natural extension and the Taylor extension\(^9\) [46, 20, 38]. Due to the properties of interval arithmetic, the evaluation of a function may yield different results according to the literal form of the equations. Thus, many literal forms may be used as, for example, factorised form (Horner for polynomial system) or distributed form [60].

Nevertheless, in general, neither the natural form nor the Taylor expansion allow to compute the exact range of a function \( f \). For instance, consider \( f(x) = 1 - x + x^2 \), and \( x = [0, 2] \), we have:

\[
\begin{align*}
\text{f}_{\text{asy}}([0, 2]) &= f(x) + (2x - 1)(x - x) = f(1) + (2[0, 2] - 1)([0, 2] - 1) = [-2, 4], \\
f([0, 2]) &= 1 - x + x^2 = 1 - [0, 2] + [0, 2]^2 = [-1.5], \\
f_{\text{factor}}([0, 2]) &= 1 + x(x - 1) = 1 + [0, 2]([0, 2] - 1) = [-1, 3]
\end{align*}
\]

\(^7\)http://www.ti3.tu-harburg.de/Software/PROFIELieghlsch.html
\(^8\)http://www.mpfr.org
\(^9\)\( f_{\text{asy}}(X) = f(X) + A(X - X) \) where \( A \) is the Jacobian or the interval slope matrix.
whereas the range of \( f \) over \( X = [0, 2] \) is \([3/4, 3]\). In this case, this result could directly be obtained by a second form of factorisation: \( \text{factor}_2([0, 2]) = (x - 1/2)^2 + 3/4 = ([0, 2] - 1/2)^2 + 3/4 = [3/4, 3] \).

### 4.2. Interval analysis methods.

This section provides a short introduction on interval analysis methods (see [35, 20, 38, 40] for a more detailed introduction). We limit this overview to interval Newton-like methods for solving a multivariate system of nonlinear equations. Their use is complementary to methods provided by the Constraint Programming community.

The aim is to determine the zeros of a system of \( n \) equations \( f_i(x_1, \ldots, x_n) \) in \( n \) unknowns \( x_i \) inside the interval vector \( X = (x_1, \ldots, x_n) \) with \( x_i \in X_i \) for \( i = 1, \ldots, n \).

First, consider solving an interval linear system of equations defined as follows:

\[
AX = b, \ A \in \mathbf{A}, \ b \in \mathbf{b}
\]

(4.3)

where \( \mathbf{A} \) is an interval matrix and \( \mathbf{b} \) is an interval vector. Solving this linear interval system requires to determine an interval vector \( \mathbf{X} \) containing all solutions of all scalar linear systems noted \( AX = b \) such that \( A \in \mathbf{A} \) and \( b \in \mathbf{b} \). Finding the exact value of \( \mathbf{X} \) is a difficult problem but three basic interval methods exist: Gaussian elimination, Gauss-Seidel iterative method or Krawczyk method (see [24, 38, 20, 40]). They may provide an overestimated interval vector \( \mathbf{X}_1 \) including \( \mathbf{X} \). However, in general the computed interval are too wide and a preconditioning is required. That is to say, a multiplication of both side of equation (4.3) by the inverse of a midpoint of \( \mathbf{A} \). The matrix \( m(\mathbf{A})^{-1} \mathbf{A} \) is then "closer" to the identity matrix and the width of \( \mathbf{X}_1 \) is smaller [20].

To solve nonlinear systems, an interval Newton algorithm is often used – see [20] or [38]. The basic idea is to solve iteratively a linear approximation of the nonlinear system obtained by a Taylor expansion. Many improvements [24, 19], based on variations of the resolution of the linear subsystem or the preconditioning have been proposed. Note that many interesting properties are provided by Newton-like methods: existence and/or uniqueness of a root, convergence area/rate, ...

### 5. Constraint programming.

This section recalls the basics of constraint programming techniques which are required to understand the rest of this paper. A detailed discussion of these concepts and techniques can be found in [6, 26].

#### 5.1. The general framework.

The constraint programming framework is based on a branch and prune scheme which was inspired by the traditional branch and bound approach used in optimisation problems. That is to say, it is best viewed as an iteration of two steps [60]:

1. Pruning the search space;
2. Making a choice to generate two (or more) sub-problems.

The pruning step ensures that some local consistency holds. In other words, the pruning step reduces an interval when it can prove that the upper bound or the lower bound does not satisfy some constraint. Informally speaking, a constraint system \( \mathcal{C} \) satisfies a partial consistency property if a relaxation of \( \mathcal{C} \) is consistent. For instance consider \( \mathbf{x} = [\mathbf{x}_1, \mathbf{\bar{x}}] \) and \( c(x, x_1, \ldots, x_n) \in \mathcal{C} \). Whenever \( c(x, x_1, \ldots, x_n) \) does not hold for any values \( a \in \mathbf{x} = [\mathbf{x}_1, \mathbf{\bar{x}}] \), then \( \mathbf{x} \) may be shrunk to \( \mathbf{x} = [\mathbf{\underline{x}}, \mathbf{\bar{x}}] \). Local consistencies are detailed in the next subsection. Roughly speaking, they are relaxations of arc-consistency, a notion well known in artificial intelligence [31, 34].

The branching step usually splits the interval associated to some variable in two intervals with the same width. However, the splitting process may generate more than two sub-problems and one may split an interval at a point different from its midpoint. The choice of the variable to split is a critical issue in difficult problems. Sophisticated splitting strategies have been developed for finite domains but few results [23] are available for continuous domains.

#### 5.2. Local consistencies [11, 26].

Local consistencies are conditions that filtering algorithms must satisfy. A filtering algorithm can be seen as a fixed point algorithm defined by the
sequence \( \{D_k\} \) of domains generated by the iterative application of an operator \( Op : \mathbb{R}^n \rightarrow \mathbb{R}^n \) (see Figure 5.1).

\[
D_k = \begin{cases} 
D & \text{if } k = 0 \\
Op(D_{k-1}) & \text{if } k > 0
\end{cases}
\]

Fig. 5.1. Filtering algorithms as fixed point algorithms

The operator \( Op \) of a filtering algorithm generally satisfies the following three properties:

- \( Op(D) \subseteq D \) (contractance)
- \( Op \) is conservative; that is, it cannot remove any solution.
- \( D' \subseteq D \Rightarrow Op(D') \subseteq Op(D) \) (monotonicity)

Under those conditions, the limit of the sequence \( \{D_k\} \), which corresponds to the greatest fixed point of the operator \( Op \), exists and is called a closure. A fixed point for \( Op \) may be characterised by a \( lc \)-consistency property, called a local consistency. The algorithm achieving filtering by \( lc \)-consistency is denoted \( lc \)-filtering. A CSP is said to be \( lc \)-satisfiable if \( lc \)-filtering of this CSP does not produce an empty domain.

Consistencies used in numerical CSPs solvers can be categorised in two main classes: arc-consistency-like consistencies and strong consistencies. Strong consistencies will not be discussed in this paper (see [30, 26] for a detailed introduction).

Most of the numerical CSPs systems (e.g., BNR-prolog [43], Interlog [8], CLP(BNR) [7], PrologIV [12], UniCalc [4], Ilog Solver [22], Numerica [61] and RealPaver [5]) compute an approximation of arc-consistency [31] which will be named ac-like-consistency in this paper. An ac-like-consistency states a local property on a constraint and on the bounds of the domains of its variables. Roughly speaking, a constraint \( c_j \) is ac-like-consistent if for any variable \( x_i \) in \( var(c_j) \), the bounds \( x_{i\text{low}} \) and \( x_{i\text{up}} \) have a support in the domains of all other variables of \( c_j \).

The most famous ac-like consistencies are 2b–consistency and box–consistency.

2b–consistency (also known as hull consistency) [10, 7, 28, 30] only requires to check the arc-consistency property for each bound of the intervals. The key point is that this relaxation is more easily verifiable than arc-consistency itself. Informally speaking, variable \( x \) is 2b–consistent for constraint \( \forall f(x, x_1, \ldots, x_n) = 0 \) if the lower (resp. upper) bound of the domain of \( x \) is the smallest (resp. largest) solution of \( f(x, x_1, \ldots, x_n) \). The box–consistency [6, 21] is a coarser relaxation (i.e., it allows less stringent pruning) of arc–consistency than 2b–consistency. Variable \( x \) is box–consistent for constraint \( \forall f(x, x_1, \ldots, x_n) = 0 \) if the bounds of the domain of \( x \) correspond to the leftmost and the rightmost zero of the optimal interval extension of \( f(x, x_1, \ldots, x_n) \).

2b–consistency algorithms actually achieve a weaker filtering (i.e., a filtering that yields bigger intervals) than box–consistency, more precisely when a variable occurs more than once in some constraint (see proposition 6 in [11]). This is due to the fact that 2b–consistency algorithms require a decomposition of the constraints with multiple occurrences of the same variable.

2b–consistency [30] states a local property on the bounds of the domains of a variable at a single constraint level. A constraint \( c \) is 2b–consistent if, for any variable \( x \), there exist values in the domains of all other variables which satisfy \( c \) when \( x \) is fixed to \( x_{\text{low}} \) and \( x_{\text{up}} \).

Filtering by 2b–consistency of \( P = (D, C) \) is the CSP \( P' = (D', C) \) such that:

- \( P \) and \( P' \) have the same solutions;
- \( P' \) is 2b–consistent;
- \( D' \subseteq D \) and the domains in \( D' \) are the largest ones for which \( P' \) is 2b–consistent.

Filtering by 2b–consistency of \( P \) always exists and is unique [30], that is to say it is a closure.
The box-consistency \([6, 21]\) is a coarser relaxation of arc-consistency than \(2b\)-consistency. It mainly consists of replacing every existentially quantified variable but one with its interval in the definition of \(2b\)-consistency. Thus, box-consistency generates a system of univariate interval functions which can be tackled by numerical methods such as interval Newton. In contrast to \(2b\)-consistency, box-consistency does not require any constraint decomposition and thus does not amplify the locality problem. Moreover, box-consistency can tackle some dependency problems when each constraint of a CSP contains only one variable which has multiple occurrences. More formally:

**Definition 5.1 (box-consistency).**

Let \((\mathcal{D}, \mathcal{C})\) be a CSP and \(c \in \mathcal{C}\) a \(k\)-ary constraint over the variables \((x_1, \ldots, x_k)\). \(c\) is box-consistent if, for all \(x_i\) the following relations hold:

1. \(c(x_1, \ldots, x_{i-1}, [x_i^-, x_i^+], x_{i+1}, \ldots, x_k)\).
2. \(c(x_1, \ldots, x_{i-1}, [\overline{x_i^+}], x_{i+1}, \ldots, x_k)\).

Closure by box-consistency of \(P\) is defined similarly to closure by \(2b\)-consistency of \(P\).

Benhamou et al. have introduced hc4 \([5]\), an ac-like-consistency that merges \(2b\)-consistency and box-consistency and which optimises the computation process.

6. **Quad basics and extensions.** This section first introduces **Quad**, a global constraint that works on a tight and safe linear relaxation of quadratic subsystems of constraints. Then, it generalises **Quad** to the polynomial part of numerical constraint systems. Different linearisation techniques are investigated to limit the number of generated constraints.

6.1. **The Quad algorithm.** The **Quad**-filtering algorithm (see Algorithm 1) consists of three main steps: reformulation, linearisation and pruning.

The reformulation step generates \([C]_R\), the set of implied linear constraints. More precisely, \([C]_R\) contains linear inequalities that approximate the semantics of nonlinear terms of \(C\).

The linearisation process first decomposes each nonlinear term in sums and products of univariate terms, then, it replaces nonlinear terms with their associated new variables. For example, consider constraint \(c : x_2x_3x_4^2(x_6 + x_7) + \sin(x_1)(x_2x_6 - x_3) = 0\), a simple linearisation transformation may yield the following sets:

- \([c]_L = \{y_1 + y_3 = 0, \ y_2 = x_6 + x_7, \ y_4 = y_5 - x_3\}\)
- \([c]_{LI} = \{y_1 = x_2x_3x_4^2y_2, \ y_3 = \sin(x_1)y_4, \ y_5 = x_2x_6\}\).

\([c]_L\) is the set of linear constraints generated by replacing the nonlinear terms by new variables and \([c]_{LI}\) denotes the set of equations that keep the link between the new variables and the nonlinear terms. Note that the nonlinear terms which are not directly handled by the **Quad** are taken into account by the box-filtering process.

Finally, the linearisation step computes the set of final linear inequalities and equations \(LR = [C]_L \cup [C]_R\), the linear relaxation of the original constraints \(C\).

The pruning step is just a fixed point algorithm that calls iteratively a linear programming solver to reduce the upper and the lower bound of every original variable. The algorithm converges and terminates if \(\epsilon\) is greater than zero.

Now we are in position to introduce the reformulation of nonlinear terms. Subsection 6.2 first introduces the handling of quadratic constraints while subsection 6.3 extends the previous results to polynomial constraints.

6.2. **Handling quadratic constraints.** Quadratic constraints are approximated by linear constraints in the following way. **Quad** creates a new variable for each quadratic term: \(y\) for \(x^2\), \(y_{i,j}\) for \(x_{i}x_{j}\). The produced system is denoted:

\[
\left[ \sum_{(i,j) \in M} a_{k,i,j}x_{i}x_{j} + \sum_{i \in N} b_{k,i}x_i^2 + \sum_{i \in N} d_{k,i}x_i = b_k \right]_L
\]
A tight linear (convex) relaxation, or outer-approximation to the convex and concave envelope of the quadratic terms over the constrained region, is built by generating new linear inequalities.

Quad uses two tight linear relaxation classes that preserve equations \( y = x^2 \) and \( y_{i,j} = x_i x_j \) and that provide a better approximation than interval arithmetic [27].

6.2.1. Linearisation of \( x^2 \). The term \( x^2 \) with \( \underline{x} \leq x \leq \overline{x} \) is approximated by the following relations:

\[
[x^2]_R = \left\{ \begin{array}{ll}
L1(\alpha) & \equiv [(x - \alpha)^2 \geq 0]_L \text{ where } \alpha \in [\underline{x}, \overline{x}]
\end{array} \right. \quad (6.1)
\]

Note that \([(x - \alpha_i)^2 = 0]_L \) generates the tangent line to the curve \( y = x^2 \) at the point \( x = \alpha_i \).

Actually, Quad computes only \( L1(\overline{x}) \) and \( L1(\underline{x}) \). Consider for instance the quadratic term \( x^2 \) with \( x \in [-4, 5] \). Figure 6.1 displays the initial curve (i.e., \( D_1 \)), and the lines corresponding to the equations generated by the relaxations: \( D_2 \) for \( L1(-4) \equiv y + 8x + 16 \geq 0 \), \( D_3 \) for \( L1(5) \equiv y - 10x + 25 \geq 0 \), and \( D_4 \) for \( L2 \equiv -y + x + 20 \geq 0 \).

We may note that \( L1(\overline{x}) \) and \( L1(\underline{x}) \) are underestimations of \( x^2 \) whereas \( L2 \) is an overestimation. \( L2 \) is also the concave envelope, which means that it is the optimal concave overestimation.

6.2.2. Bilinear terms. In the case of bilinear terms \( xy \), McCormick [32] proposed the following relaxations of \( xy \) over the box \( [\underline{x}, \overline{x}] \times [\underline{y}, \overline{y}] \), stated in the equivalent RLT form [54]:

\[
[xy]_R = \left\{ \begin{array}{ll}
BIL1 & \equiv [(x - \underline{x})(y - \underline{y}) \geq 0]_L \\
BIL2 & \equiv [(x - \overline{x})(y - \underline{y}) \geq 0]_L \\
BIL3 & \equiv [(\overline{x} - x)(y - \overline{y}) \geq 0]_L \\
BIL4 & \equiv [(\overline{x} - x)(\overline{y} - y) \geq 0]_L 
\end{array} \right. \quad (6.2)
\]

BIL1 and BIL3 define a convex envelope of \( xy \) whereas BIL2 and BIL4 define a concave envelope of \( xy \) over the box \( [\underline{x}, \overline{x}] \times [\underline{y}, \overline{y}] \). Al-Khayyal and Falk [1] showed that these relaxations are the optimal convex/concave outer-estimations of \( xy \).

Consider for instance the quadratic term \( xy \) with \( x \in [-5,5] \) and \( y \in [-5,5] \). The work done by the linear relaxations of the 3D curve \( z = xy \) is well illustrated in 2D by fixing \( z \). Figure 6.2 displays the 2D shape, for the level \( z = 5 \), of the initial curve (i.e., \( Cu \)), and the lines corresponding
Linearisation process to polynomial constraints. We first discuss the quadrification process and constraint system, a process called *quadrification*. In this section, we show how to transform a polynomial constraint system into an equivalent quadratic one and compare it with RLT. Then, we present the linearisations of product and power terms.

6.3. Extension to polynomial constraints. In this section, we show how to extend the linearisation process to polynomial constraints. We first discuss the quadrification process and compare it with RLT. Then, we present the linearisations of product and power terms.

6.3.1. Transformation of nonlinear constraints into quadratic constraints. In this section, we show how to transform a polynomial constraint system into an equivalent quadratic constraint system, a process called *quadrification* [58].

For example, consider the constraint \( c : x_2 x_3 x_4^2 + 3 x_6 x_7 + \sin(x_1) = 0 \), the proposed transformation yields

\[
\{y_1 y_2 + 3 y_2 + s_1 = 0, \ y_1 = x_2 x_3, \ y_2 = x_4 x_4, \ y_3 = x_6 x_7\}
\]

and the set \( \{y_1 = x_2 x_3, \ y_2 = x_4^2, \ y_3 = x_6 x_7, \ s_1 = \sin(x_1)\} \) of equations that keep the link between the new variables and the nonlinear terms that cannot be further quadrified. Such a transformation is one of the possible quadrifications. It is called a *single* quadrification.

We could generate all possible single quadrifications, or all quadrifying identities, and perform a so called *complete* quadrification. For example, the complete quadrification of \( E = \{x_2 x_3 x_4^2 + 3 x_6 x_7 + \sin(x_1) = 0\} \) is

\[
\begin{align*}
y_1 + 3 y_2 + s_1 &= 0, \ y_2 &= x_6 x_7, \\
y_1 &= y_3 y_4, \ y_3 &= x_2 x_3, \ y_4 &= x_4^2, \\
y_1 &= y_5 y_6, \ y_5 &= x_2 x_4, \ y_6 &= x_3 x_4, \\
y_1 &= x_2 y_7, \ y_7 &= x_3 y_4, \ y_7 &= x_4 y_6, \\
y_1 &= x_3 y_8, \ y_8 &= x_2 y_4, \ y_8 &= x_4 y_5, \\
y_1 &= x_4 y_9, \ y_9 &= x_2 y_6, \ y_9 &= x_3 y_5, \ y_9 &= x_4 y_3
\end{align*}
\]

where \( s_1 = \sin(x_1) \).

A quadrification for polynomial problems was introduced by Shor [58]. Sherali and Tuncbilek [57] have proposed a direct reformulation/linearisation (RLT) of the whole polynomial constraints without quadrifying the constraints. They did prove the dominance of their direct reformulation/linearisation technique over Shor’s quadrification [56].

A complete quadrification generates as many new variables as the direct RLT. Linearisations proposed in RLT are built on every non-ordered combination of \( \delta \) variables, where \( \delta \) is the highest polynomial degree of the constraint system.
The complete quadrification generates linearisations on every couple of non-ordered combined variables \([v_i, v_j]\) where \(v_i\) (resp. \(v_j\)) is the variable that has been introduced for linearising the non-ordered combination of variables.

Complete quadrification and direct RLT yields a tighter linearisation than the single quadrification but the number of generated linearisations grows in an exponential way for non-trivial polynomial constraint systems. More precisely, the number of linearisations depends directly on the number of generated new variables.

To sum up, the linearisation of polynomial systems offers two main possibilities: the transformation of the initial problem into an equivalent quadratic constraint system through a process called quadrification, or the direct linearisation of polynomial terms by means of RLT. Theoretical considerations, as well as experimentations, have been conducted to exclude as practical a complete quadrification which produces a huge amount of linear inequalities for non trivial polynomial systems. The next two subsections presents our choices for the linearisation of product and power terms.

6.3.2. Product terms. For the product term

\[ x_1 x_2 \ldots x_n \] (6.3)

we use a two steps procedure: quadrification and bilinear relaxations.

Since many single quadrifications exist, an essential point is the choice of a good heuristic that captures most of the semantics of the polynomial constraints. We use a “middle” heuristic to obtain balanced degrees on the generated terms. For instance, consider \( T \equiv x_1 x_2 \ldots x_n \), a monomial of degree \( n \), the middle heuristic will identify two monomials \( T_1 \) and \( T_2 \) of highest degree such that \( T = T_1 T_2 \). It follows that \( T_1 = x_1 x_2 \ldots x_{n+2} \) and \( T_1 = x_{n+2+1} \ldots x_n \).

The quadrification is performed by recursively decomposing each product \( x_1 \ldots x_j \) into two products \( x_1 \ldots x_d \) and \( x_{d+1} \ldots x_j \). Of course, there are many ways to choose the position of \( d \). Sahni et al. \([49, 51]\) use what they call \( \text{rAI} \), “recursive interval arithmetic”, which is a recursive quadrification. Sheri and Tuncbilek \([57]\) have proposed a promising direct reformulation/linearisation technique (RLT) of the whole polynomial constraints without quadrifying the constraints. Applying RLT on the product term \( x_1 x_2 \ldots x_n \) generates the following \( n \)-ary inequalities \(^{10}\):

\[
\prod_{i \in J_1} (x_i - x) \prod_{i \in J_2} (x_i - x) \geq 0, \quad \text{for all } J_1, J_2 \subseteq \{1, \ldots, n\} : |J_1 \cup J_2| = n
\] (6.4)

where \( \{1, \ldots, n\} \) is to be understood as a multiset and where \( J_1 \) and \( J_2 \) are multisets.

We now introduce proposition 6.1 that states the number of new variables and relaxations respectively generated by the quadrification and RLT process on the product term (6.3).

**Proposition 6.1.** Let \( T \equiv x_1 x_2 \ldots x_n \) be some product of degree \( n \geq 1 \) with \( n \) distinct variables. The RLT of \( T \) will generate up to \((2^n - n - 1)\) new variables and \(2^n\) inequalities whereas the quadrification of \( T \) will only generate \((n - 1)\) new variables and \(4(n - 1)\) inequalities.

\(^{10}\)Linearisations proposed in RLT on the whole polynomial problem are built on every non-ordered combination of \( \delta \) variables, where \( \delta \) is the highest polynomial degree of the constraint system.
Bilinear terms require four relaxations, thus we get \(4^n\) for the case \(n=1\). These combinations. Thus, the number of variables is bounded by \(\sum_{i=2}^n C_i = \sum_{i=0}^n C_i - n - 1\), that is to say \(2^n - n - 1\) since \(\sum_{i=0}^n C_i = 2^n\). In (6.4), for each variable we consider alternatively lower and upper bound, thus there are \(2^n\) new inequalities.

For the quadrification process, the proof can be done by induction. For \(n = 1\), the formula is true. Now suppose that for length \(i\) (with \(1 \leq i < n\)), \((i - 1)\) new variables are generated. For \(i = n\), we can split the term at the position \(d\) with \(1 \leq d < n\). It results from the induction hypothesis that we have \(d - 1\) new variables for the first part, and \(n - d - 1\) new variables for the second part, plus one more new variable for the whole term. So, \(n - 1\) new variables are generated. Bilinear terms require four relaxations, thus we get \(4(n-1)\) new inequalities.

Proposition 6.2 states that quadrification with bilinear relaxations provides convex and concave envelopes with any \(d\). This property results from the proof given in [49] for the \(r\)\(\text{AI}\) heuristic.

**Proposition 6.2.** \([x_1,x_2,...,x_n]_{RLT}\) provides convex and concave envelopes of the product term \(x_1x_2...x_n\).

Generalisation for sums of products, the so-called multi-linear terms \(\sum_{i=1...t} a_i \prod_{j \in J} x_j\), have been studied recently [14, 52, 48, 49]. It is well known that finding the convex or concave envelope of a multi-linear term is a NP-hard problem [14]. The most common method of linear relaxation of multi-linear terms is based on the simple product term. However, it is also well known that this approach leads to a poor approximation of the linear bounding of the multi-linear terms. Sherali [52] has introduced formulae for computing convex envelopes of the multi-linear terms. It is based on an enumeration of vertices of a pre-specified polyhedra which is of exponential nature. Rikun [48] has given necessary and sufficient conditions for the polyhedrality of convex envelopes. He has also provided formulae of some faces of the convex envelope of a multi-linear function. To summarise, it is difficult to characterise convex and concave envelopes for general multi-linear terms. Conversely, the approximation of “product of variables” is an effective approach; moreover, it is easy to implement [51, 50].

**6.3.3. Power terms.** A power term of the form \(x^n\) can be approximated by \(n+1\) inequalities with a procedure proposed by Sherali and Tunçbilek [57], called “bound-factor product RLT constraints”. It is defined by the following formula:

\[
[x^n]_R = \{[(x - z)^i(x - x)_{n-1} \geq 0]_L, i = 0...n\}
\]  

(6.5)

The essential observation is that this relaxation generates a tight relation between variables on their upper and lower bounds. More precisely, suppose that some original variable takes a value equal to either of its bounds. Then all the corresponding new RLT linearisation variables that involve this original variable take a relative values that conform with actually fixing this original variable at its particular bound in the nonlinear expressions represented by these new RLT variables [57]. Note that relaxations (6.5) of the power term \(x^n\) are expressed with \(x^i\) for all \(i \leq n\), and thus provide a fruitful relationship on problems containing many power terms involving some variable.

The univariate term \(x^n\) is convex when \(n\) is even, or when \(n\) is odd and the value of \(x\) is negative; it is concave when \(n\) is odd and the value of \(x\) is positive. Sahinidis and Twarmalani [50] have introduced the convex and concave envelopes when \(n\) is odd by taking the point where the power term \(x^n\) and its under-estimator have the same slope. These convex/concave relaxations on \(x^n\) are expressed with only \([x^n]_L\) and \(x\). In other words, they do not generate any relations with \(x^i\) for \(1 < i < n\).

That is why we suggest to implement the approximations defined by formulae (6.5). Note that for the case \(n = 2\), (6.5) provides the concave envelope.
7. A safe rounding procedure for the Quad-algorithm. This section details the rounding procedure we propose to ensure the completeness of the Quad algorithm [33]. First, we show how to compute safe coefficients for the generated linear constraints. In the second subsection we explain how a recent result from Neumaier and Shcherbina allows us to use the simplex algorithm in a safe way.


a) Approximation of $L1$. The linear constraint $L1(y, \alpha) \equiv y - 2\alpha x + \alpha^2 \geq 0$ approximates a term $x^2$ with $\alpha \in [x, \overline{x}]$. $L1(y, \alpha)$ corresponds to the tangent lines to the curve $y = x^2$ at the point $(\alpha, \alpha^2)$.

Thus, the computation over the floats of the coefficients of $L1(y, \alpha)$ may change the slope of the tangent line as well as the intersection points with the curve $y = x^2$. Consider the case where $\alpha$ is negative: the solutions are above the tangent line; thus we have to decrease the slope to be sure to keep all of the solutions. It follows that we have to use a rounding mode towards $+\infty$. Likewise, when $\alpha$ is positive, we have to set the rounding mode towards $-\infty$. More formally, we have:

$$L1_{\text{IF}}(y, \alpha) \equiv \begin{cases} y - \nabla(2\alpha)x + \Delta(\alpha^2) \geq 0, & \text{if } \alpha \geq 0 \\ y - \Delta(2\alpha)x + \Delta(\alpha^2) \geq 0, & \text{if } \alpha < 0 \end{cases}$$

where $\nabla(x)$ (resp. $\Delta(x)$) denotes a rounding mode of $x$ towards $-\infty$ (resp. $+\infty$).

b) Approximation of $L2$. The case of $L2$ is a bit more tricky since the “rotation axis” of the line defined by $L2$ is between the extremum values of $x^2$ ($L2(y)$ is an overestimation of $y$). Thus, to keep all the solutions we have to strengthen the slope of this line at its smallest extremum. It follows that:

$$L2_{\text{IF}} \equiv \begin{cases} \Delta(x + \overline{x})x - y - \nabla(\overline{x}) \geq 0, & \text{if } x \geq 0 \\ \nabla(x + \overline{x})x - y - \nabla(\overline{x}) \geq 0, & \text{if } x < 0 \\ \Delta(x + \overline{x})x - y - \nabla(\overline{x}) + Ulp(\Delta(x + \overline{x})\overline{x}) \geq 0, & \text{if } x > 0, x < 0, |x| \leq |\overline{x}| \\ \nabla(x + \overline{x})x - y - \nabla(\overline{x}) - \Delta(Ulp(\Delta(x + \overline{x})\overline{x})) \geq 0, & \text{if } x > 0, x < 0, |x| > |\overline{x}| \end{cases}$$

where $Ulp(x)$ computes the distance between $x$ and the float following $x$.

c) Approximation of $BIL1, BIL2, BIL3$ and $BIL4$. The general form of $BIL1, BIL2, BIL3$ and $BIL4$ is $x, x_1 + s_1 b_1 x_1 + s_2 b_2 x_j + s_3 b_1 b_2 \geq 0$ where $b_1$ and $b_2$ are floating point numbers corresponding to bounds of $x_1$ and $x_j$ whereas $s_i \in \{-1, 1\}$.

The term $s_3 b_1 b_2$ is the only term which results from a computation: all the other terms use constants which are not subject to round off errors. Thus, these linear constraints can be rewritten in the following form: $Y + s_3 b_1 b_2$.

A rounding of $s_3 b_1 b_2$ towards $+\infty$ enlarges the solution space, and thus ensures that all these linear constraints are safe approximations of $x^2$.

It follows that $BIL\{1..4\}_{\text{IF}} \equiv Y + \Delta(s_3 b_1 b_2) \geq 0$.

d) Approximation of multivariate linearisations. We are now in position to introduce the corrections of multivariate linearisations as introduced for the power of $x$. Such linearisations could be rewritten in the following form:

$$\sum_{i=1}^{n} a_i x_i + b \geq 0$$
where \( a_i \) denotes the expression used to compute the coefficient of variable \( x_i \) and \( b \) the expression used to compute the constant value. Proposition 7.1 takes advantage of interval arithmetic to compute a safe linearisation with coefficients over the floating point numbers:

**Proposition 7.1.**

\[
\sum_{i=1}^{n} \overline{a}_i x_i + \sup(b + \sum_{i=1}^{n} \sup(\overline{a}_i x_i - \overline{a}_i x_i)) \geq \sum_{i=1}^{n} a_i x_i + b \geq 0, \forall x_i \in x_i.
\]

**Proof.** \( \forall x_i \in x_i \), we have

\[
\sum_{i=1}^{n} \overline{a}_i x_i + \sup(b + \sum_{i=1}^{n} \sup(\overline{a}_i x_i - \overline{a}_i x_i)) \geq \sum_{i=1}^{n} \overline{a}_i x_i + b + \sum_{i=1}^{n} (\sup(\overline{a}_i x_i - \overline{a}_i x_i))
\]

and

\[
\sum_{i=1}^{n} \overline{a}_i x_i + b + \sum_{i=1}^{n} (\sup(\overline{a}_i x_i - \overline{a}_i x_i)) = \sum_{i=1}^{n} (\overline{a}_i (x_i - \underline{x}_i) + \sup(a_i x_i)) + b
\]

As \( \forall i \in \{1, \ldots, n\} \), we have \( \overline{a}_i \geq a_i, \sup(a_i x_i) \geq a_i x_i \), and \( \forall x_i \in x_i, x_i - \underline{x}_i \geq 0 \). Therefore :

\[
\sum_{i=1}^{n} (\overline{a}_i (x_i - \underline{x}_i) + \sup(a_i x_i)) + b \geq \sum_{i=1}^{n} (a_i (x_i - \underline{x}_i) + a_i x_i) + b = \sum_{i=1}^{n} a_i x_i + b
\]

\( \square \)

This proposition provides a safe approximation of a multivariate linearisation which holds for any \( a_i, x_i \) and \( b \). This result could be refine by means of the previous approximations. For instance, whenever \( \underline{x}_i \geq 0, \overline{a}_i x_i \geq a_i x_i \). In this case, there is no need for an additional correction.

e) **Approximation of initial constant values.** Initial constant values are real numbers that may not have an exact representation within the set of floating point numbers. Thus, a safe approximation is required.

Constant values in inequalities have to be correctly rounded according to the orientation of the inequality. The result presented in the previous paragraph sets the rounding directions which have to be used.

Equations must be transformed into inequalities when their constant values have to be approximated.

#### 7.2. Computation of safe bounds with linear programming algorithm.

Linear programming methods can solve problems of the following form:

\[
\min \quad C^T X
\]

\[
\text{such that} \quad \underline{B} \leq AX \leq \overline{B}
\]

\[
\underline{X} \leq X \leq \overline{X}
\]

The solution of such a problem is a vector \( X_r \in \mathbb{R}^n \). However, the solution computed by solvers like CPLEX or SOPLEX is a vector \( X_f \in \mathbb{R}^n \) that may be different from \( X_r \) due to the rounding errors. More precisely, \( X_f \) is safe for the objective only if \( C^T X_r \geq C^T X_f \).

Neumaier and Shcherbina [42] provide a cheap method to obtain a rigorous bound of the objective and certificates of infeasibility. The essential observation is that the dual of (7.1) is

\[
\max \quad \underline{B}^T Z' + \overline{B}^T Z''
\]

\[
\text{such that} \quad \overline{A}^T (Z' - Z'') = C
\]
Let \( Y = Z' - Z'' \), and the residue \( R = A^T Y - C \in \mathbb{R} = [\mathbb{R}, \overline{\mathbb{R}}] \). It follows that
\[
C^T X = (A^T Y - R)^T X = Y^T A X - R^T X \in Y^T [\mathbb{R}, \overline{\mathbb{R}}] - R^T [\mathbb{X}, \overline{\mathbb{X}}]
\]
and the value of \( \mu \), the lower bound of the value of the objective function is:
\[
\mu = \inf(Y^T B - R^T X) = \vee(Y^T B - R^T X) \tag{7.3}
\]
Formula (7.3) is trivially correct by construction. Note that the precision of such a safe bound depends on the width of the intervals \([\mathbb{X}, \overline{\mathbb{X}}]\).

So, we have just to apply this correction before updating the lower bound and the upper bound of each variable.

However, the linear program (7.1) may be infeasible. In that case, Neumaier and Shcherbina show that whenever \( d = \inf(R^T X - Y^T B) > 0 \) where \( R' = A^T Y \in \mathbb{R}' \), then it is certain that no feasible point exist. However, the precision of interval arithmetic does not always allow to get a positive value for \( d \) while the linear program is actually infeasible. In the latter case, we consider it as feasible. Note that box-consistency may be able to reject most, if not all, of the domains of such variables.

8. Experimental results. This section reports experimental results of Quad on a variety of twenty standard benchmarks. Benchmarks eco6, katsura5, katsura6, katsura7, tangents2, ipp, assur44, cyclic5, tangents0, chemequ, noon5, geneig, kinema, reimer5 and cameras were taken from Verschelde’s web site\textsuperscript{11}, kin2 from [60], didrit from [15], lee from [29], and finally yama194, yama195 and yama196 from [63]. The most challenging benchmark is stewgou40 [16]. It describes the 40 possible positions of a Gough-Stewart platform as a function of the values of the actuators. The proposed modeling of this problem consists of 9 equations with 9 variables.

The experimental results are reported in Tables 8.1 and 8.2. Column \( n \) (resp. \( \delta \)) shows the number of variables (resp. the maximum polynomial degree). \( BP(\Phi) \) stands for a Branch and Prune solver based on the \( \Phi \) filtering algorithm, that is to say, a search-tree exploration where a filtering technique \( \Phi \) is applied at each node. \( \text{quad}(\mathbb{H}) \) denotes the Quad algorithm where bilinear terms are relaxed with formulae (6.2), power terms with formulae (6.5) and product terms with the quadrification method; \( \mathbb{H} \) stands for the heuristic used for decomposing terms in the quadrification process.

The performances of the following five solvers have been investigated:
1. RealPaver: a free Branch and Prune solver\textsuperscript{12} that dynamically combines optimised implementations of box-consistency filtering and 2b-consistency filtering algorithms [5]
2. BP(box): a Branch and Prune solver based on the ILOG\textsuperscript{13} commercial implementation of box-consistency
3. BP(box+simplex): a Branch and Prune solver based on box and a simple linearisation of the whole system without introducing linear relaxations of the nonlinear terms
4. BP(box+quad(Qmid)): a Branch and Prune solver which combines box and the Quad algorithm where product terms are relaxed with the Qmid heuristic
5. BP(box+quad(rAI)): a Branch and Prune solver which combines box and the Quad algorithm where product terms are relaxed with the rAI heuristic

Note that the BP(box+simplex) solver implements a strategy that is slightly different to Yamamura’s approach [63].

All the solvers have been parameterised to get solutions or boxes with precision of \( 10^{-8} \). That is to say, the width of the computed intervals is smaller than \( 10^{-8} \). A solution is said to be safe

\textsuperscript{11}The database of polynomial systems available at http://www.math.uic.edu/~jan/Demo/
\textsuperscript{12}See http://www.sciences.univ-nantes.fr/info/perso/permanents/granvill/realpaver/main.html
\textsuperscript{13}See http://www.ilog.com/products/jsolver
if we can prove its uniqueness within the considered box. This proof is based on the well known
Brouwer fixed point theorem (see [20]) and just requires a single test.
Sols, \( Ksplit \) and \( T(s) \)) are respectively the number of solutions, the number of thousands of
branchings (or splittings) and the execution time in seconds. The number of solutions is followed
with a number of safe solutions between brackets. A "-" in the column \( T(s) \)) means that the solver
was unable to find all the solutions within eight hours. All the computations have been performed
on a PC with Pentium IV processor at 2.66GHz running Linux. The compiler was GCC 2.9.6 used
with the -O6 optimisation flag.
Table 8.1 displays the performances of \texttt{RealPaver}, \texttt{BP(box)} and \texttt{BP(box+quad(Qmid))}. The
benchmarks have been grouped into three sets. The first group contains problems where the
\texttt{QuadSolver} does not behave very well. These problems are quite easy to solve and the overhead
of the relaxation and the calls to a linear solver does not pay off. The second group contains a
set of benchmarks for which the \texttt{QuadSolver} compares well with the two other constraint solvers:
the \texttt{QuadSolver} requires always much less splitting and often less time than the other solvers.
In the third group, which contains difficult problems, the \texttt{QuadSolver} outperforms the two other
constraint solvers. The latter were unable to solve most of these problems within eight hours
whereas the \texttt{QuadSolver} managed to find all the solutions for all but two of them in less than
8 minutes. For instance, \texttt{BP(box)} requires about 74 hours to find the four solutions of the \texttt{Lee}
benchmark whereas the \texttt{QuadSolver} managed to do the job in a couple of minutes. Likewise, the
\texttt{QuadSolver} did find the forty safe solutions of the \texttt{stewgou40} benchmark in about 15 minutes
whereas \texttt{BP(box)} required about 400 hours. The essential observation is that the \texttt{QuadSolver}
spends more time in the filtering step but it performs much less splitting than classical solvers.
This strategy pays off for difficult problems.

<table>
<thead>
<tr>
<th>Name</th>
<th>( n )</th>
<th>( \delta )</th>
<th>\texttt{BP(box+quad(Qmid))}</th>
<th>\texttt{BP(box)}</th>
<th>\texttt{RealPaver}</th>
</tr>
</thead>
<tbody>
<tr>
<td>cyclev5</td>
<td>5</td>
<td>5</td>
<td>10(10)</td>
<td>10(10)</td>
<td>13.4</td>
</tr>
<tr>
<td>eco6</td>
<td>6</td>
<td>3</td>
<td>4(4)</td>
<td>4(4)</td>
<td>1.7</td>
</tr>
<tr>
<td>assur44</td>
<td>8</td>
<td>3</td>
<td>10(10)</td>
<td>10(10)</td>
<td>15.8</td>
</tr>
<tr>
<td>ip5</td>
<td>8</td>
<td>2</td>
<td>10(10)</td>
<td>10(10)</td>
<td>4.6</td>
</tr>
<tr>
<td>katsura5</td>
<td>6</td>
<td>2</td>
<td>16(11)</td>
<td>41(11)</td>
<td>8.2</td>
</tr>
<tr>
<td>katsura6</td>
<td>7</td>
<td>2</td>
<td>60(24)</td>
<td>182(24)</td>
<td>136.6</td>
</tr>
<tr>
<td>kn5</td>
<td>8</td>
<td>2</td>
<td>10(10)</td>
<td>10(10)</td>
<td>3.5</td>
</tr>
<tr>
<td>noon5</td>
<td>5</td>
<td>3</td>
<td>11(11)</td>
<td>11(11)</td>
<td>50.2</td>
</tr>
<tr>
<td>tangents2</td>
<td>6</td>
<td>2</td>
<td>24(24)</td>
<td>24(24)</td>
<td>14.1</td>
</tr>
</tbody>
</table>

Table 8.1

Experimental results: comparing \texttt{Quad} and Constraint solvers
All the problems, except cyclic5 and reimer5, contain many quadratic terms and some product and power terms. cyclic5 is a pure multi-linear problem that contains only sums of products of variables. The Quad algorithm has not been very efficient for handling this problem. Of course, one could not expect an outstanding performance on this benchmark since product term relaxation is a poor approximation of multi-linear terms. reimer5 is a pure power problem of degree 6 that has been well solved by the Quad algorithm.

Table 8.2 displays the performances of solvers combining box-consistency and three different relaxation techniques. There is no significant difference between the solver based on the Qmid heuristics and the solver based on the rAI heuristics. Indeed, both heuristics provide convex and concave envelopes of the product terms. The QuadSolver with relaxations outperforms the BP(box+simplex) approach for all benchmarks but yama195, which is a quasi-linear problem. These performances on difficult problems illustrate well the capabilities of the relaxations.

Note that Verscheile's homotopy continuation system, PHCpack [62], required 115s to solve lee and 1047s to solve stegovou40 on our computer. PHCpack is a state of the art system in the solving of polynomial systems of equations. Unfortunately, it is limited to polynomial systems and does not handle inequalities. PHCpack searches for all the roots of the equations, whether real or complex, and it does not restrict its search to a given subspace. The homotopy continuation approach also suffers from an exponential growing computation time which depends on the number of nonlinear terms (PHCpack failed to solve yama195 which contains 3600 nonlinear terms). In contrast to homotopy continuation methods, QuadSolver can easily be extended to non-polynomial systems.

Thanks to Arnold Neumaier and Oleg Shcherbina, we had the opportunity to test BARON [50] with some of our benchmarks. QuadSolver compares well with this system. For example, BARON
6.014 and QuadSolver require more or less the same time to solve cameras, didrit, kinema and lee. BARON needs only 1.59s to find all the solutions of yama196 but it requires 859.6s to solve yama195. Moreover, BARON loses some solutions on reimer5 (22 solutions found) and stewgou40 (14 solutions found) whereas it generates numerous wrong solutions for these two problems. We must also underline that BARON is a global optimisation problem solver and that it has not been built to find all the solutions of a problem.

9. Conclusion. In this paper, we have exploited an RLT schema to take into account specific semantics of nonlinear terms. This relaxation process is incorporated in the Branch and Prune process [60] that exploits interval analysis and constraint satisfaction techniques to find rigorously all solutions in a given box. The reported experimental results show that this approach outperforms the classical constraint solvers.

G. Pesant and M. Boyer [44, 45] first introduced linear relaxations in a CLP language to handle geometrical constraints. However, the approximation of the constraints was rather weak. The approach introduced in this paper is also related to recent work that has been done in the interval analysis community as well as to some work achieved in the optimisation community.

In the interval analysis community, Yamamura et al. [63] used a simple linear relaxation procedure where nonlinear terms are replaced by new variables to prove that some box does not contain solutions. No convex/concave outer-estimations are proposed to obtain a better approximation of the nonlinear terms. As pointed out by Yamamura, this approach is well adapted to quasi-linear problems: “This test is much more powerful than the conventional test if the system of nonlinear equations consists of many linear terms and a relatively small number of nonlinear terms” [63].

The global optimisation community worked also on solving nonlinear equation problems by transforming them into an optimisation problem (see for example chapter 23 in [17]). The optimisation approach has the capability to take into account specific semantics of nonlinear terms by generating a tight outer-estimation of these terms. The pure optimisation methods are usually not rigorous since they do not take into account rounding errors and do not prove the uniqueness of the solutions found.

Acknowledgements. We thank Arnold Neumaier for his fruitful comments on an early version of this paper. We are also grateful to Arnold Neumaier and Oleg Shcherbina for their help in testing BARON.

REFERENCES


The tests was performed on an Athlon XP 1800 computer.


