Acceleration methods for numeric CSPs

Yahia LEBBAH, Olivier LHOMME
Ecole des Mines de Nantes - La Chantrerie
4, rue Alfred Kastler - B. P. 20722
44 307 Nantes Cedex 3 - France
{Yahia.Lebbah, Olivier.Lhomme}@emn.fr

Abstract

This paper introduces a new way of accelerating the convergence of numeric CSP filtering algorithms, through the use of extrapolation methods. Extrapolation methods are used in numerical analysis to accelerate the convergence of real number sequences. We will show how to use them for solving numeric CSPs, leading to drastic improvement in efficiency.

Introduction

Many industrial and engineering problems can be seen as constraint satisfaction problems (CSPs). A CSP is defined by a set of variables, each with an associated domain of possible values and a set of constraints on the variables. This paper deals with CSPs where the constraints are numeric relations and where the domains are continuous domains (numeric CSPs). Numeric CSPs can be used to express a large number of problems, in particular physical models involving imprecise data or partially defined parameters.

In general, numeric CSPs cannot be tackled with computer algebra systems, and most numeric algorithms cannot guarantee completeness. The only numeric algorithms that can guarantee completeness – even when floating-point computations are used – are coming either from the interval analysis community or from the AI community (CSP).

Those constraint-solving algorithms are typically a search-tree exploration where a pruning (or filtering) technique is applied at each node. The challenge is to find the best compromise between a filtering technique that achieves a strong pruning at a high computational cost and another one that achieves less pruning at a lower computational cost.

The filtering technique used may be a kind of arc-consistency filtering adapted to numeric CSPs (Davis 1987; Hyvönen 1992; Lhomme 1993). Other works (Lhomme 1993; 1994; Haroud & Faltings 1996) aim at defining concepts of higher order consistencies similar to k-consistency (Freuder 1978). (Faltings 1994; Faltings & Gelle 1997) propose in a sense to merge the constraints concerning the same variables, giving one "total" constraint (thanks to numerical analysis techniques) and to perform arc-consistency on the total constraints. (Benhamou, McAllester, & Van-Hentenryck 1994; Van-Hentenryck, Mc Allester, & Kapur 1997) aim at expressing interval analysis pruning as partial consistencies. All the above works address the issue of finding a new partial consistency property that can be computed by an associated filtering algorithm with a good efficiency (with respect to the domain reductions performed).

Another direction in the search of the best compromise, is to try to optimize the computation of already existing consistency techniques. Optimizing a given consistency technique may itself be of great interest, but providing general methods for optimizing a class of consistency techniques is better. In this category, (Lhomme *et al.* 1996; Lhomme, Gotlieb, & Rueher 1998) propose a method for dynamically detecting and simplifying cyclic phenomena during the running of a consistency technique. In this paper, we are pursuing the same goal of giving some general methods for accelerating numeric CSPs consistency techniques. Let us outline our approach in very general terms.

1. The computations achieved by numeric CSPs consistency techniques are seen as a sequence.

2. Since numerical analysis provides different mathematical tools for accelerating the convergence of sequences, we study the use of such tools for optimizing our sequences. More specifically, the paper focuses on the use of extrapolation methods.

3. Filtering techniques never lose a solution, they are said to be *complete*. A direct use of extrapolation

methods fails to preserve completeness. We will show how to preserve completeness while using extrapolation methods for accelerating filtering techniques.

The paper is organized as follows. In section 2, some definitions concerning numeric CSPs are summarized. Section 3 presents extrapolation methods. Section 4 shows how to use extrapolation methods to accelerate the convergence of CSP filtering algorithms. Section 5 contains an additional discussion.

Numeric CSPs

This section presents numeric CSPs in a slightly non-standard form, which will be convenient for our purposes.

A numeric CSP is a triplet $< \mathcal{X}, \vec{\mathcal{D}}, \mathcal{C} >$ where:

- X is a set of n variables $x_1, \ldots, x_n$
- $\vec{\mathcal{D}} = (D_1, \ldots, D_n)$ denotes a vector of domains. The i^{th} component of $\vec{\mathcal{D}}$, D_i, is the domain containing all acceptable values for x_i.
- $\mathcal{C} = \{C_1, \ldots C_m\}$ denotes a set of constraints. $var(C)$ denotes the variables appearing in C.

This paper focuses on CSP where the domains are intervals. The following notation is used throughout the paper. The lower bound and the upper bound of an interval D_i are respectively denoted by $\underline{D_i}$ and $\overline{D_i}$. $\vec{\mathcal{D}'} \subset \vec{\mathcal{D}}$ means $D'_i \subset D_i$ for all $i \in 1 \ldots n$. $\vec{\mathcal{D}'} \cap \vec{\mathcal{D}''}$ means $(D'_1 \cap D''_1, \ldots, D'_n \cap D''_n)$.

A k-ary constraint $C(x_1, \ldots, x_k)$ is a relation over the reals. The algorithms used over numeric CSPs typically work by narrowing domains and need to compute the projection of a constraint $C(x_1, \ldots, x_k)$ over each variable x_i in the space delimited by $D_1 \times \ldots \times D_k$. Such a projection cannot be computed exactly due to several reasons, such as: (1) the machine numbers are floating point numbers and not real numbers so round-off errors occur; (2) the projection may not be representable as floating point numbers; (3) the computations needed to have a close approximation of the projection of only one given constraint may be very expensive; (4) the projection may be discontinuous whereas it is much more easy to handle only closed intervals for the domains of the variables.

Thus, what is usually done is that the projection of a constraint over a variable is approximated. Let $\pi_i(C, D_1 \times \ldots \times D_k)$ denote such an approximation. All that is needed is that $\pi_i(C, D_1 \times \ldots \times D_k)$ includes the exact projection; this is possible thanks to interval analysis (Moore 1966). $\pi_i(C, D_1 \times \ldots \times D_k)$ hides all the problems seen above. In particular, it allows us not to go into the details of the relationships between floating point and reals numbers (see for example (Alefeld & Hezberger 1983) for those relationships) and to consider only reals numbers.

```
proc 2B(inout D⃗)
    while (Op2B(D⃗) ≠ D⃗)
        D⃗ ← Op2B(D⃗)
    endwhile

funct Op2B(D⃗)
    return ∩_{j=1..m} Π_{C_j}(D⃗)
```

Figure 1: $2B$–consistency filtering algorithm

For notational convenience, the k projections of a k-ary constraint C over $(x_{i_1}, \ldots, x_{i_k})$ are computed by only one operator $\Pi_C(\vec{\mathcal{D}})$. This is a filtering operator over *all* the domains. It performs reductions, if any, over the domains $D_{i_1}, \ldots, D_{i_k}$ and leaves the other domains unchanged. That is: let $\vec{\mathcal{D}'} = \Pi_C(\vec{\mathcal{D}})$

$\forall j \in \{1, \ldots, n\}$

$$D'_j = \begin{cases} D_j & \text{if } x_j \notin var(C) \\ \pi_j(C, D_{i_1} \times \ldots \times D_{i_k}) & \text{if } x_j \in var(C) \end{cases}$$

Most of the numeric CSPs systems (e.g., BNR-prolog (Older & Velino 1990), CLP(BNR) (Benhamou & Older 1997), PrologIV (Colmerauer 1994), UniCalc (Babichev *et al.* 1993), Ilog Solver (Ilog 1997) and Numerica (Van-Hentenryck, Michel, & Deville 1997) compute an approximation of arc-consistency (Mackworth 1977) which will be named $2B$–consistency in this paper[1]. $2B$-consistency states a local property on a constraint and on the bounds of the domains of its variables (B of $2B$-consistency stands for *bound*). Roughly speaking, a constraint C is $2B$-consistent if for any variable x_i in C the bounds $\underline{D_i}$ and $\overline{D_i}$ have a support in the domains of all other variables of C (*wrt* the approximation given by π). $2B$-consistency can be defined in our formalism as:

Definition 1 ($2B$-consistency) *We say that a* CSP $< \mathcal{X}, \vec{\mathcal{D}}, \mathcal{C} >$ *is $2B$-consistent iff* $\forall C, \forall i$
$(C \in \mathcal{C} \wedge x_i \in var(C) \Rightarrow \pi_i(C, D_1 \times \ldots \times D_k) = D_i)$

The fixpoint algorithm in Figure 1 achieves the filtering by $2B$–consistency. The operator $Op2B$ applies on the *same* vector $\vec{\mathcal{D}}$ all the Π_C operators. This has the drawback of increasing the upper bound of the complexity, but has the advantage of generating a much

[1] We have a lot of freedom to choose $\pi_i(C, D_1, \ldots, D_k)$, so the definition of $2B$-consistency here both abstracts $2B$-consistency in (Lhomme 1993) and *box*-consistency in (Benhamou, McAllester, & Van-Hentenryck 1994)

more "regular" sequences of domains (see section 5). For the sake of simplicity, AC3-like optimization, which consists in applying only those projection functions that may reduce a domain, does not appear explicitly in this algorithm schema.

We say that a CSP is $2B$–satisfiable if the $2B$ filtering of this CSP does not produce an empty domain.

In the same way that arc–consistency has been generalized to higher consistencies (e.g. path–consistency), $2B$–consistency can be generalized to kB–consistency (e.g. $3B$–consistency (Lhomme 1993)). In the rest of this paper, we focus on $3B$–consistency, but the results can be straightforwardly generalized to kB–consistencies.

$3B$–consistency ensures that when a variable is instantiated to one of its two bounds, then the CSP is $2B$–satisfiable. More generally, $3B(w)$–consistency ensures that when a variable is forced to be close to one of its two bounds (more precisely, at a distance less than w), then the CSP is $2B$–satisfiable.

Definition 2 ($3B(w)$-consistency) *We say that a* CSP $< \mathcal{X}, \vec{\mathcal{D}}, \mathcal{C} >$ *is $3B(w)$-consistent iff :*

$$\begin{array}{l} \forall i,\ i \in \{1,\ldots,n\} \Rightarrow \\ < \mathcal{X}, \underline{\psi}(\vec{\mathcal{D}}, i, w), \mathcal{C} > \textit{ is 2B-satisfiable} \\ \quad \textit{and} \\ < \mathcal{X}, \overline{\psi}(\vec{\mathcal{D}}, i, w), \mathcal{C} > \textit{ is 2B-satisfiable} \end{array}$$

Where $\underline{\psi}(\vec{\mathcal{D}}, i, w)$ (resp. $\overline{\psi}(\vec{\mathcal{D}}, i, w)$) denotes the same domain as $\vec{\mathcal{D}}$ except that D_i is replaced by $D_i \cap [\underline{D_i}, \underline{D_i} + w]$ (resp. D_i is replaced by $D_i \cap [\overline{D_i} - w, \overline{D_i}]$).

$3B$-consistency filtering algorithms, used for example in Interlog (Dassault-Electronique. 1991), Ilog Solver or Numerica, are based on the schema of algorithms given in Figure 2, which uses a kind of proof by contradiction: the algorithm tries to increase the lower bound of D_i by proving that $\underline{\psi}(\vec{\mathcal{D}}, i, w)$ is not $2B$-satisfiable and it tries to decrease the upper bound in a symmetric way.

Implementations using this schema may be optimized considerably, but we do not need to go into details here.

Acceleration by extrapolation methods

This section summarizes the field of extrapolation methods of accelerating the convergence of sequences; for a deeper overview see (Brezinski & Zaglia 1990). Let $\{S_n\} = (S_1, S_2, \ldots)$ be a sequence of real numbers. A sequence $\{S_n\}$ converges if and only if it has a limit S: $\lim_{n\to\infty} S_n = S$.

Accelerating the convergence of a sequence $\{S_n\}$ amounts of applying a transformation $\mathcal{A}$ which produces a new sequence $\{T_n\}$: $\{T_n\} = \mathcal{A}(\{S_n\})$.

```
proc 3B(in w, inout D⃗)
   while (Op3B(D⃗) ≠ D⃗)
      D⃗ ← Op3B(D⃗)
   endwhile

funct Op3B(in D⃗)
   for i := 1 to n do
         if ¬2B–satisfiable(ψ̲(D⃗, i, w))
            D_i ← D_i + w
         if ¬2B–satisfiable(ψ̄(D⃗, i, w))
            D̄_i ← D̄_i − w
   endfor
   return D⃗
```

Figure 2: $3B$(w)–consistency filtering schema

As given in (Brezinski & Zaglia 1990), in order to present some practical interest the new sequence $\{T_n\}$ must exhibit, at least for some particular classes of convergent sequences $\{S_n\}$, the following properties:

1. $\{T_n\}$ must converge
2. $\{T_n\}$ converges to the same limit as $\{S_n\}$: $\lim_{n\to\infty} T_n = \lim_{n\to\infty} S_n$
3. $\{T_n\}$ converges faster than $\{S_n\}$: $\lim_{n\to\infty} \frac{T_n - S}{S_n - S} = 0$

As explained in (Brezinski & Zaglia 1990), these properties do not hold for all converging sequences. Particularly, a universal transformation $\mathcal{A}$ accelerating all converging sequences cannot exist (Delahaye & Germain-Bonne 1980). Thus any transformation can accelerate a limited class of sequences. This leads us to a so-called *kernel*[2] of the transformation which is the set of convergent sequences $\{S_n\}$ for which $\exists N, \forall n \succeq N, T_n = S$ where $\{T_n\} = \mathcal{A}(\{S_n\})$.

A famous transformation is the iterated Δ^2 process from Aitken (Aitken 1926) $\{T_n\} = \Delta^2(\{S_n\})$, which gives a sequence $\{T_n\}$ of n^{th} term:

$$T_n = \frac{S_n S_{n+2} - S_{n+1}^2}{S_{n+2} - 2S_{n+1} + S_n}$$

The kernel of Δ^2 process is the set of the converging sequences which have the form:

$$S_n = S + \alpha\lambda^n$$

where $\alpha \neq 0$ and $\lambda \neq 1$.

Let us see the behavior of Δ^2 transformation on the following sequence:

$$S_n = \begin{cases} 1 & \text{if } n = 0 \\ S_{n-1}/1.001 & \text{if } n > 0 \end{cases}$$

[2]The definition of the kernel given here considers only converging sequences.

Let $\{T_n\} = \Delta^2(\{S_n\})$, we have:

$$
\begin{array}{lll}
S_0 = 1 & & \\
S_1 = 0.9990009990009991 & & \\
S_2 = 0.9980029960049943 & \longmapsto & T_0 = 0.0 \\
S_3 = 0.9970059900149795 & \longmapsto & T_1 = 0.0 \\
\ldots & & \\
S_{1000} = 0.36843136759310596 & \longmapsto & T_{998} = 0.0
\end{array}
$$

Δ^2 process enables the limit of that sequence to be found immediately.

Of course we can apply several times the transformation leading to a new transformation. For example, we can apply Δ^2 twice, giving $\Delta^2(\Delta^2(\{S_n\}))$.

Many acceleration transformations (G-algorithm, ϵ-algorithm, θ-algorithm, *overholt*-process,...) are multiple application of transformations (see (Brezinski & Zaglia 1990).

Such accelerating convergence methods have been generalized to sequences of vectors or matrices.

Extrapolation for filtering

Filtering algorithms can be seen as sequences of domains. For example, the sequence $\{\vec{\mathcal{D}}_n\}$ generated by $2B$-filtering is:

$$\vec{\mathcal{D}}_n = \begin{cases} \vec{\mathcal{D}} & \text{if } n = 0 \\ Op2B(\vec{\mathcal{D}}_{n-1}) & \text{if } n > 0 \end{cases}$$

and the sequence $\{\vec{\mathcal{D}}_n\}$ generated by $3B$-filtering is:

$$\vec{\mathcal{D}}_n = \begin{cases} \vec{\mathcal{D}} & \text{if } n = 0 \\ Op3B(\vec{\mathcal{D}}_{n-1}) & \text{if } n > 0 \end{cases}$$

So, in order to optimize a filtering algorithm, it suffices to accelerate the convergence of its associated domains sequence $\{\vec{\mathcal{D}}_n\}$. $\{\vec{\mathcal{D}}_n\}$ is a sequence of interval vectors. There does not exist any method to accelerate interval sequences, but an interval can be seen as two reals and $\vec{\mathcal{D}}$ can be seen as a 2-columns matrix of reals (the first column is the lower bounds, and the second the upper bounds). Thus we can apply the acceleration methods $\{\Delta^2, \epsilon, \theta, \ldots\}$ seen in the previous section.

In our experiments, scalar extrapolations, which consider each element of the matrix — each bound of a domain– independently of the others, appear to be much more robust than vectorial extrapolations. Consequently the results given in the rest of the paper are for scalar extrapolations. For example, the scalar Δ^2 process uses for each bound of domain the last three different values to extrapolate a value.

Two kinds of optimization for filtering algorithms are now given. The first makes a direct use of extrapolation methods while the second is a somewhat tricky one.

First kind of optimization

Let $\{\vec{\mathcal{D}}_n\}$ be a sequence generated by a filtering algorithm. Accelerating the convergence of $\{\vec{\mathcal{D}}_n\}$ can dramatically boost the convergence, as illustrated in the following problem:

$$x * y + t - 2 * z = 4, \quad x * \sin(z) + y * \cos(t) = 0,$$
$$x - y + \cos(z)^2 = 0, \quad x * y * z - 2 * t = 0$$
$$x \in [0, 1000], y \in [0, 1000], z \in [0, \pi], t \in [0, \pi]$$

The following table shows the domain of the variable t in the 278^{th}, 279^{th}, 280^{th} and 281^{th} iterations of $3B$–filtering (after a few seconds). The precision obtained is about 10^{-4}.

it	*t*
278	$[\mathbf{3.14133342842583}\cdots, \mathbf{3.14159265358979}\cdots]$
279	$[\mathbf{3.14134152921220}\cdots, \mathbf{3.14159265358979}\cdots]$
280	$[\mathbf{3.14134937684900}\cdots, \mathbf{3.14159265358979}\cdots]$
281	$[\mathbf{3.14135697924715}\cdots, \mathbf{3.14159265358979}\cdots]$

By applying the Δ^2 process on the domains of the iterations 278, 279 and 280 we obtain the domain below. The precision of this extrapolated domain is 10^{-8}. Such a precision has not been obtained after 5 hours of the $3B$–filtering algorithm without extrapolation.

t
$[\mathbf{3.14159265358977}\cdots, \mathbf{3.14159265358979}\cdots]$

The extrapolated sequence may or may not converge to the same limit as the initial sequence. This can be explained by the kernel of the transformation: when the initial sequence belongs to the kernel, then we are sure that the extrapolated sequence converges to the same limit. Furthermore, intuition suggests that, if the initial sequence is "close" to the kernel then there are good hopes to get the same limit. However it may be the case that the limits are quite different. This is cumbersome for the filtering algorithms which must ensure that no solution is lost.

We must now address the question of how to apply acceleration methods on filtering algorithms without losing any solution.

Second kind of optimization

We propose below an answer to the question above, for the domain sequences generated by algorithms that are based on the $3B$–algorithm schema of Figure 2.

This answer is built on the proof–by–contradiction mechanism used in $3B$–algorithm: it tries to prove that no solution exists in a subpart of a domain. If such a proof is found, then the subpart is removed from the domain, else the subpart is not removed.

The point is that we may waste a lot of time trying to find a proof that does not exist. If we could predict

```
proc 3B–acc(in w, inout D⃗)
    while (OpAcc3B(D⃗) ⊂ D⃗)
          D⃗ ← OpAcc3B(D⃗)
    endwhile

funct OpAcc3B(in D⃗)
    for i := 1 to n do
    (1).   if ¬2B–sat–predict(ψ̲(D⃗, i, w))
    (2).       if ¬2B-satisfiable(ψ̲(D⃗, i, w))
                  D̲i ← D̲i + w
    (3).   if ¬2B–sat–predict(ψ̄(D⃗, i, w))
    (4).       if ¬2B-satisfiable(ψ̄(D⃗, i, w))
                  D̄i ← D̄i + w
    endfor
    return D⃗

funct 2B–sat–predict(in D⃗)
    if it predicts 2B–satisfiability
          by extrapolation methods.
       return true
    else return false
```

Figure 3: Accelerated $3B$–acc(w) filtering schema

with a good probability that such a proof does not exist, we could save time in not trying to find it. Well, extrapolation methods can do the job !

The idea is simply that if an extrapolated sequence converges to a $2B$–satisfiable CSP (which can be quickly known), then it is probably a good idea not to try to prove the $2B$–unsatisfiability.

Following that idea, the algorithm schema for $3B(w)$–consistency can be modified, as given in Figure 3. The main difference is that before testing for $2B$–satisfiability, it tries to predict $2B$–satisfiability by extrapolation methods.

We artificially separate step (1) from step (2) and step (3) from step (4) in the schema for reasons of clarity only. In an implementation of that algorithm schema, these steps are better taken together: that is $2B$–sat–predict is permanently maintained during the computation of $2B$–satisfiability. The computation may thus be interrupted as soon as the prediction allows it.

The following proposition means that this algorithm schema allows acceleration methods to be applied while keeping the completeness property of filtering algorithms.

proposition 1 (Completeness) *The accelerated $3B$–acc(w) algorithm does not lose any solution.*

The proof is built on the fact that a domain is reduced only when we have a proof — by $2B$–satisfiability and without extrapolation— that no solution exists for the removed part.

The counterpart of this result is that improvements in efficiency of $3B$–acc(w) compared with $3B$ may be less satisfactory than improvement provided by the first kind of optimization.

Nevertheless, the improvement in efficiency may also be of several orders of magnitude. For example, let us consider the following example from (Brezinski 1978):

$$-t + 4 * x - 1/2 = 0, \quad -x + y^2/2 + t = 0,$$
$$-y + \sin(x) + \sin(y-1) + 1 = 0$$
$$x \in [-3, 3], y \in [-100, 100], t \in [-100, 100]$$

The acceleration method used is Δ^2. To reach a precision of 10^{-8}, we obtained the following results.

algorithm	$3B$	$3B$–acc
CPU–time(sec)	154	0.1
nb–calls–of–$Op2B$	699735	284
nb–calls–of–$Op2B$–$true$	699642	152
nb–calls–of–$Op2B$–$false$	93	132

As we see, the improvement in time is about three orders of magnitude. We gain about 99.95% on successful calls to $Op2B$, thanks to extrapolation methods. However, we have a few more failed calls to $Op2B$ than the standard algorithm.

The first experimentations that have been done over some standard benchmarks of numeric CSPs lead to an average improvement in time of about 3. The overhead in time has always been negligible.

Discussion

Extrapolation methods are worth applying for filtering algorithms when:

- *convergence is slow.* When the convergence of the filtering algorithms is fast, there is little or nothing to gain in applying extrapolation methods. At first sight, one could think that slow convergences do not occur very often in filtering algorithms. This is not true. $2B$–consistency filtering leads relatively often to slow convergence and kB–filtering most of the time leads to slow convergence.

- *sequences to be extrapolated follow a kind of "regularity".* All the work of an extrapolation method is to capture that regularity early in the sequence for predicting the fixpoint of the sequence. Of course, the concept of regularity is only intuitive and may greatly vary depending on the extrapolation method used.

In our experiments, we used a $2B$–filtering as given in Figure 1, which applies the projection functions to the *same* vector of domains. It is in no way mandatory, but its advantage, against a standard AC3-like

algorithm that applies the projection functions sequentially, was mainly for studying different extrapolation methods. In fact, the order in which the projection functions are applied in a standard AC3-like algorithm may hide the regularity of the sequence, whereas in our version, this cannot happen. Also, it allows vectorial extrapolation methods to be used much more easily.

The first kind of optimization can be applied over all filtering algorithms. The second kind of optimization for $3B$ consistency algorithms can be straightforwardly generalized to higher consistency algorithms.

Conclusion

A product like Numerica (Van-Hentenryck, Michel, & Deville 1997) has shown that an efficient way of solving numeric CSPs is to put many different algorithmic tools to work together. This paper proposes a new family of tools that can be combined with existing ones easily and at a negligible overhead cost. They consist in applying extrapolation methods over the sequence of domains given by filtering algorithms.

Extrapolation methods can be directly applied, leading to drastic improvements in efficiency, but solutions of the CSP may be lost.

Extrapolation methods can also be used in a more subtle way, allowing solutions of the CSP to be preserved. The idea is that before doing a hard work that may succeed or may fail, we may want to have a strong presumption that it is worth doing. Extrapolation methods are used to give such a presumption.

We are now working on three directions: an efficient AC3-like implementation using only scalar extrapolations; other ways of extrapolating a sequence of domains without loss of solution; the study of the improvement of different extrapolation methods on the different filtering algorithms.

References

Aitken, A. C. 1926. On bernoulli's numerical solution of algebraic equations. In *Proc. R. Soc. Edinb.*, 46:289–305.

Alefeld, G., and Hezberger, J., eds. 1983. *Introduction to Interval Computations.* Academic press.

Babichev, A.; Kadyrova, O. B.; Kashevarova, T. P.; Leshchenko, A. S.; and Semenov, A. 1993. Unicalc, a novel approach to solving systems of algebraic equations. *Interval Computations* 2:29–47.

Benhamou, F., and Older, W. 1997. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming* 32(1):1–24.

Benhamou, F.; McAllester, D.; and Van-Hentenryck, P. 1994. Clp(intervals) revisited. In *Proceedings of the 1994 International Symposium*, 109–123.

Brezinski, C., and Zaglia, R., eds. 1990. *Extrapolation methods.* Studies in Computational Mathematics. North-Holland.

Brezinski, C., ed. 1978. *Algorithmes d'accélération de la convergence: étude numériques.* Technip.

Colmerauer, A. 1994. Spécifications de prolog iv. Technical report, GIA, Faculté des Sciences de Luminy,163, Avenue de Luminy 13288 Marseille cedex 9 (France).

Dassault-Electronique. 1991. Interlog 1.0: Guide d'utilisation. Technical report, Dassault Electronique, 55 Quai M. Dassault, 92214 Saint Cloud, France.

Davis, E. 1987. Constraint propagation with interval labels. *Journal of Artificial Intelligence* 32:281–331.

Delahaye, J. P., and Germain-Bonne, B. 1980. Résultats négatifs en accélération de la convergence. *Numer. Math* 35:443–457.

Faltings, B., and Gelle, E. 1997. Local consistency for ternary numeric constraints. In *Proceedings of International Joint Conference on Artificial Intelligence.*

Faltings, B. 1994. Arc consistency for continuous variables. *Journal of Artificial Intelligence* 60(2):363–376.

Freuder, E. C. 1978. Synthesizing constraint expressions. *Communications of the ACM* 21:958–966.

Haroud, D., and Faltings, B. 1996. Consistency techniques for continuous constraints. *Constraints* 1(1–2):85–118.

Hyvönen, E. 1992. Constraint reasoning based on interval arithmetic: the tolerance propagation approach. *Journal of Artificial Intelligence* 58:71–112.

Ilog., ed. 1997. *ILOG Solver 4.0, Reference Manual.* Ilog.

Lhomme, O.; Gotlieb, A.; Rueher, M.; and Taillibert, P. 1996. Boosting the interval narrowing algorithm. In *Proc. of the 1996 Joint International Conference and Symposium on Logic Programming*, 378–392. MIT Press.

Lhomme, O.; Gotlieb, A.; and Rueher, M. 1998. Dynamic optimization of interval narrowing algorithms. *Journal of Logic Programming* Forthcoming.

Lhomme, O. 1993. Consistency techniques for numeric CSPs. In *Proceedings of International Joint Conference on Artificial Intelligence*, 232–238.

Lhomme, O. 1994. *Contribution à la résolution de contraintes sur les réels par propagation d'intervalles.* Ph.D. Dissertation, Université de Nice — Sophia Antipolis BP 145 06903 Sophia.

Mackworth, A. 1977. Consistency in networks of relations. *Journal of Artificial Intelligence* 8(1):99–118.

Moore, R., ed. 1966. *Interval Analysis.* Prentice Hall.

Older, W. J., and Velino, A. 1990. Extending prolog with constraint arithmetic on real intervals. In *Proc. of IEEE Canadian conference on Electrical and Computer Engineering*, 14.1.1–14.1.4. IEEE Computer Society Press.

Van-Hentenryck, P.; Mc Allester, D.; and Kapur, D. 1997. Solving polynomial systems using branch and prune approach. *SIAM Journal on Numerical Analysis* 34(2):797–827.

Van-Hentenryck, P.; Michel, L.; and Deville, Y., eds. 1997. *Numerica, a modeling language for global optimization.* the MIT press.