Solving Parametric Linear Systems: an Experiment with Constraint Algebraic Programming

Clemens Ballarin
Institut für Informatik
Technische Universität München
85748 Garching, Germany
ballarin@in.tum.de

Manuel Kauers
RISC-Linz
Johannes-Kepler-Universität
4040 Linz, Austria
manuel@kauers.de

Abstract

Algorithms in computer algebra are usually designed for a fixed set of domains. For example, algorithms over the domain of polynomials are not applicable to parameters because the inherent assumption that the indeterminate X bears no algebraic relation to other objects is violated.

We propose to use a technique from model theory known as *constraint programming* to gain more flexibility, and we show how it can be applied to the Gaussian algorithm to be used for parametric systems. Our experiments suggest that in practice this leads to results comparable to the algorithm for parametric linear systems by Sit (1992) — at least if the parameters are sparse.

1 Introduction

A common feature of many computer algebra systems, in particular general purpose ones, is that the variable symbols they manipulate bear no algebraic or logical relation to other objects. Their semantics is that of *indeterminates,* not of *logical variables* or *parameters.* This semantics is common to many algorithms in computer algebra.

The choice of the semantics of symbols makes an important difference. Linear equation systems are the focus of this paper. The rank of a matrix over a polynomial domain $k[X]$ may become smaller if values are substituted for the indeterminate X, and hence the solution space of the corresponding equation system differs. As a consequence, the *generic* solution obtained over $k[X]$ is not necessarily correct under substitutions over k. Even worse, special solutions can not necessarily be obtained from the generic solution or from the original matrix. The following example, taken from the textbook by Noble and Daniel (1988) illustrates this.

Example 1 *The augmented matrix*

$$\left(\begin{array}{ccc|c} 1 & -2 & 3 & 1 \\ 2 & x & 6 & 6 \\ -1 & 3 & x-3 & 0 \end{array}\right)$$

has the reduced row echelon form

$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & \frac{x+9}{x+4} \\ 0 & 1 & 0 & \frac{4}{x+4} \\ 0 & 0 & 1 & \frac{1}{x+4} \end{array}\right).$$

It is immediate that $x = -4$ is a special case. There is another special case $x = 0$, which is not obvious from the result. For $x = 0$ and $x = -4$ the reduced row echelon forms are

$$\left(\begin{array}{ccc|c} 1 & 0 & 3 & 3 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{array}\right) \quad \textit{and} \quad \left(\begin{array}{ccc|c} 1 & 0 & -5 & 0 \\ 0 & 1 & -4 & 0 \\ 0 & 0 & 0 & 1 \end{array}\right),$$

respectively, and the sets of solutions differ from the general case.

The effect illustrated in the example is known as *specialization problem.*

There are some occasions where computer algebra systems deal with the specialization problem. The integrator of Macsyma addresses the problem by asking the user if, for example, knowledge about the sign of a parameter is necessary for further computations (Davenport *et al.*, 1993). Maple provides an assume facility, which maintains a global context of assumptions on symbolic objects. Assumptions are asserted by the user and can be queried during a computation (Weibel and Gonnet, 1993).

Asking the user is not practical if many queries have to be resolved. An unnerved user likely aborts the computation after the tenth query or so. More importantly, if the symbol was introduced during the computation itself, the user is in no position of answering such queries. This can conceivably happen if the procedure issuing the queries is called as a subroutine by another algorithm that introduced the symbol.

In the present paper, we propose a general approach to the specialization problem that is based on ideas from *constraint logic programming,* and that allows for the exchange of constraints between subprograms in a natural way. The approach is illustrated with the probably simplest algorithm where the specialization problem occurs: Gaussian elimination. The approach is feasible, and the capability of solving linear systems can be improved substantially by combining it with suitable strategies.

Sit (1992) has presented a special purpose algorithm for parametric linear systems. Towards the end of this paper (Section 5.3) a practical comparison of our approach with Sit's algorithm is presented. Both algorithms compute a set of subsets of the parameter space such that in each subset the solution can be presented in a uniform way. These subsets are called *regimes* and cover the entire parameter space. Sit's algorithm proceeds by first computing regime candidates and, in a second step, solving the system for each regime separately. In contrast, in our constraint programming approach, regimes are incrementally built up during the process of solving the system.

2 Constraint Logic Programming

The notion of a *constraint store* plays a central role in constraint logic programming. A constraint store is a data-structure that maintains knowledge about objects that appear in

the current computation. The knowledge may have been asserted as part of the problem specification, or it may have been discovered during the computation. This knowledge is also known as *context.* The purpose of this section is to introduce the notion of constraint store. We do this in the context of logic programming.

The programming language Prolog is based on the resolution principle (Robinson, 1965), which is combined with a fixed control strategy. A Prolog program $\mathcal{C}$ is a set of clauses built over a language of constant, function, variable and predicate symbols, and the negation symbol $\neg$. A *term* consists, in the usual manner, of constants, functions and variables. An *atom* consists of a predicate symbol and a number of terms. A *literal* is either an atom or a negated atom. A *clause* is a set of literals and represents their disjunction. The program itself denotes the conjunction of its clauses. Execution of the program determines whether a contradiction can be derived from the clause set. If this is the case, an *answer substitution* σ is returned that maps variables in $\mathcal{C}$ to terms, such that $\mathcal{C}\sigma$ is not satisfiable. Satisfiability of clause sets is not decidable in general, therefore in Prolog clauses are required to be in *Horn form* — that is, only one literal may be non-negative.

During the execution of a Prolog program, the resolution rule is repeatedly applied to pairs of clauses, and new clauses are derived and added to the clause set. The new set is satisfiable if and only if the old set is satisfiable. The resolution rule has the following form.

$$\frac{R \cup \{p(s_1, \ldots, s_n)\} \qquad S \cup \{\neg p(t_1, \ldots, t_n)\}}{R\tau \cup S\tau} \tag{1}$$

Here τ denotes the *most general unifier (mgu)* of $p(s_1, \ldots, s_n)$ and $p(t_1, \ldots, t_n)$, a substitution such that $(s_1\tau = t_1\tau) \wedge \ldots \wedge (s_n\tau = t_n\tau)$. The rule can only be applied if such a substitution exists. The resolution process terminates when an empty clause is derived. This means that the set is not satisfiable. Unifiers are accumulated along the computation and provide the answer substitution.

To a reader not familiar with logic programming this introduction to resolution may seem a bit sketchy. The only notable point for the purpose of this paper is that the resolution rule, together with the control strategy, describes a step-wise computation. Constraint logic programming is based on the observation that matching two literals and unifying their terms can be disentangled in the computation step (Frühwirth and Abdennadher, 1997). Unification decides if $(s_1 = t_1) \wedge \ldots \wedge (s_n = t_n)$ is consistent with syntactic equality of terms built of function and constant symbols. This is also known as *Clark's equation theory.*

In ordinary logic programming, this theory is fixed. The generalised resolution rule used in constraint logic programming operates on clauses that are enriched by *constraints.* We denote this by $\langle R, C\rangle$, where R is a clause and C contains a number of equations that are logically connected by conjunction. It is possible to extend Prolog to other equation theories T_e where satisfiability can be decided. This is achieved by a new resolution rule.

$$\frac{\langle R \cup \{p(x_1, \ldots, x_n)\}, C\rangle \qquad \langle S \cup \{\neg p(y_1, \ldots, y_n)\}, D\rangle}{\langle R \cup S, C \wedge D \wedge (x_1 = y_1) \wedge \ldots \wedge (x_n = y_n)\rangle} \tag{2}$$

The rule can only be applied if $C \wedge D$ is satisfiable. Note that literals now only may contain variables, not arbitrary terms.

From an abstract point of view, the decision procedure for T_e provides a service that can be encapsulated in a separate module, which can also be used in applications other than logic programming — for example, in this paper, to Gaussian elimination. Often, for reasons of efficiency, it is required that the decision procedure is incremental. This leads to the concept of a *reasoning specialist* for a theory T_e with the following interface functionality.

- cs-init(): returns a T_e-valid constraint store — for example, the empty constraint store.
- cs-unsat(C): true only if C is T_e-unsatisfiable.
- cs-simp(C, D): the main functionality of the reasoning specialist. This function adds the assertions in D to the constraint store C, obtaining a new constraint store. For soundness it is required that if cs-simp$(C, D) = C'$ then $C, D \models_{T_e} C'$.

The interface functionality of the reasoning specialist is sometimes extended by other functions — for example, by a function cs-normal(C, t) that simplifies term t with respect to the facts in constraint store C. It is not required that cs-normal computes a normal form, even though the name of the function suggests that.

3 Gaussian Elimination for Parametric Systems

Our extension of the Gaussian algorithm to parametric systems is based on constraint programming. Assumptions on parameters are maintained in constraint stores, and a suitable reasoning specialist is used to decide problems in the parameter domain.

The common Gaussian elimination algorithm works as follows. In a given matrix A a non-zero entry a_{ij}, the *pivot,* is selected. Then, using suitable row transformations, all other entries in column j are reduced to zero. Finally, the algorithm is applied recursively to the submatrix obtained by deleting row i and column j, until the remaining submatrix is a 1×1-matrix, or contains only zero-entries.

Our constraint variant of Gaussian elimination takes both a matrix A (representing a homogeneous system) with parametric entries and a constraint store C as arguments. The critical modification concerns pivot selection. Any entry p that is not the zero-term is a suitable candidate. If $p \neq 0$ is inconsistent with C (that is, C entails $p = 0$ or $C \models p = 0$) then p is not suitable as a pivot. If $p = 0$ is inconsistent with C (that is, $C \models p \neq 0$) then p is a suitable pivot. Otherwise, that is both $p = 0$ and $p \neq 0$ are consistent with C, the algorithm branches. On one branch, $p = 0$ is added to the constraint store, and this relation may be used to simplify matrix entries. On the other branch, the assumption $p \neq 0$ is added.

New matrix entries are computed during elimination as sums of products of other entries and are thus polynomial expressions over the entries of the original matrix. The smallest suitable theory for the reasoning specialist is thus the theory of polynomial equations and inequations (if an elimination scheme is used that does not introduce fractions). This theory can be decided with the radical membership test, which is based on Hilbert's Nullstellensatz and Buchberger's algorithm (Cox *et al.*, 1992), and leads to the following implementation of the reasoning specialist, where the constraint store is represented by a set of polynomials.

- cs-init(): the empty set of polynomials.

- cs-simp$(C, p = 0)$: returns $C \cup \{p\}$.

  cs-simp$(C, p \neq 0)$: returns $C \cup \{py - 1\}$ for a new variable y.

- cs-unsat(C): returns true if and only if 1 is element of the radical ideal generated by C.

- cs-normal(C, t): returns the ideal reduction of t modulo the ideal $\langle C \rangle$ generated by C.

Note that cs-normal could be extended to return a normal form of t with respect to C but this would require to compute the radical ideal generated by C, a fairly expensive operation, which is not necessary for the radical membership test.

4 Strategies for Sparse Parametric Systems

A subset of the parameter space for which a uniform solution to the equation system can be computed is called a regime. Sit (1992) has pointed out that the number of regimes computed by Gaussian elimination and branching is exponentially higher than necessary. Any optimisation therefore must aim at reducing the number of regimes. The minimum number of regimes needed to cover the parameter space is small if the number of symbolic matrix entries is small. Again, the number of regimes needed to cover the parameter space may grow exponentially with the number of symbolic entries. Therefore the fundamental assumption for the following optimisations is that the matrix is *symbolically sparse* — that is, only a small number of its entries is symbolic. This assumption is important also because of the expression swell in symbolic Gaussian elimination: because the entries in the matrix tend to become larger, solving large symbolic systems is a hard problem, even if these systems are not parametric. While this swell is only polynomial and as such negligible for parametric systems, it is conceivable that larger expressions may lead to more, or in some sense more complicated regimes.

4.1 The Markowitz Criterion Goes Symbolic

A good strategy to prevent expression swell in a sparse matrix is to select a pivot such that the number of entries that stay zero during elimination is maximal, or, phrased the other way round, that the *fill-in* created by the operation is minimal. A good heuristic, used in numerics for non-symbolic matrices, is to choose a pivot, say a_{ij}, where the number of non-zero entries r_i in the row and the number of non-zero entries c_j in the column are minimal. More precisely, the so-called *Markowitz Criterion* is to choose a non-zero entry for which $(r_i - 1)(c_j - 1)$ is minimal (Duff *et al.*, 1986).

For symbolic matrices the criterion needs to be changed. Our aim is to keep the symbolic fill-in small in order to reduce the likelihood of branching. Although analogous to Markowitz's criterion, the criterion that we propose is more complex. Four classes of matrix entries are distinguished:

0: This class consists of the zero element only.

$\mathbf{1}$: The class of constant, non-zero polynomials.

$\boldsymbol{x}$: Polynomials that are not constant but known to be non-zero relative to the current constraint store.

$\boldsymbol{X}$: All other polynomials.

The *symbolic fill-in* caused by an elimination step is a triple $(f^{\boldsymbol{X}}, f^{\boldsymbol{x}}, f^{\mathbf{1}})$ where f^c denotes the number of entries that change to class c from a simpler class — that is, a class defined earlier.

The fill-in is estimated from the class of the pivot and from the classes of the entries in the pivot row and column. The other entries of the matrix and their values are not taken into account, and the estimate is an upper bound of the actual fill-in. Let the matrix considered in an elimination step be an $n \times m$-submatrix. Let a_{ij} be the pivot, in row i and column j. Let r_i^c denote the entries of class c in row i and c_j^c the number of entries in column j (excluding the pivot — that is, $m = \sum_c r_i^c + 1$ and $n = \sum_c c_j^c + 1$). The *estimated fill-in* in class c is denoted by f_{ij}^c.

A pivot a_{ij} is chosen, such that $(f_{ij}^{\boldsymbol{X}}, f_{ij}^{\boldsymbol{x}}, f_{ij}^{\mathbf{1}})$ is minimal wrt. lexicographic order. If this does not lead to a unique choice then the product of total degree and number of monomials are compared and the pivot with the smaller value is chosen.

Formulae for the estimated fill-in can be obtained by application of the elimination scheme to classes instead of values. They are given in Table 1. Let us illustrate the derivation of $f_{ij}^{\mathbf{1}}$, $f_{ij}^{\boldsymbol{x}}$ and $f_{ij}^{\boldsymbol{X}}$, where the pivot is of class $\mathbf{1}$, as an example. The elimination scheme used in our implementation is division-free elimination with the update formula

$$a_{kl} \leftarrow a_{ij}a_{kl} - a_{kj}a_{il}$$

for all rows $k \neq i$. In order to estimate the fill-in, this is applied to a generic matrix, where $*$ denotes arbitrary entries.

$$\begin{array}{c}
\begin{array}{ccccc} \mathbf{1} & \boldsymbol{X} & \boldsymbol{x} & \mathbf{1} & \mathbf{0} \end{array} \\
\begin{pmatrix}
\mathbf{1} & \boldsymbol{X} & \boldsymbol{x} & \mathbf{1} & \mathbf{0} \\
\boldsymbol{X} & * & * & * & * \\
\boldsymbol{x} & * & * & * & * \\
\mathbf{1} & * & * & * & * \\
\mathbf{0} & * & * & * & *
\end{pmatrix}
\end{array}
\begin{array}{l}
 \quad -\boldsymbol{X} \quad -\boldsymbol{x} \quad -\mathbf{1} \\
\mid \cdot 1 \leftarrow + \\
\mid \cdot 1 \leftarrow + \\
\mid \cdot 1 \leftarrow + \\

\end{array}$$

$$\rightsquigarrow \begin{pmatrix}
\mathbf{1} & \boldsymbol{X} & \boldsymbol{x} & \mathbf{1} & \mathbf{0} \\
\mathbf{0} & * \cdot \mathbf{1} - \boldsymbol{X} \cdot \boldsymbol{X} & * \cdot \mathbf{1} - \boldsymbol{x} \cdot \boldsymbol{X} & * \cdot \mathbf{1} - \mathbf{1} \cdot \boldsymbol{X} & * \cdot \mathbf{1} \\
\mathbf{0} & * \cdot \mathbf{1} - \boldsymbol{X} \cdot \boldsymbol{x} & * \cdot \mathbf{1} - \boldsymbol{x} \cdot \boldsymbol{x} & * \cdot \mathbf{1} - \mathbf{1} \cdot \boldsymbol{x} & * \cdot \mathbf{1} \\
\mathbf{0} & * \cdot \mathbf{1} - \boldsymbol{X} \cdot \mathbf{1} & * \cdot \mathbf{1} - \boldsymbol{x} \cdot \mathbf{1} & * \cdot \mathbf{1} - \mathbf{1} \cdot \mathbf{1} & * \cdot \mathbf{1} \\
\mathbf{0} & * & * & * & *
\end{pmatrix}$$

The lower L-shaped region in the resulting matrix indicates entries where no fill-in occurs. The rectangle in the middle has constant fill-in and consists of $(r_i^{\mathbf{1}} - 1)(c_j^{\mathbf{1}} - 1)$ entries, hence $f_{ij}^{\mathbf{1}} = (r_i^{\mathbf{1}} - 1)(c_j^{\mathbf{1}} - 1)$. In the remaining box the fill-in may be of class $\boldsymbol{X}$, and $f_{ij}^{\boldsymbol{X}} = (m - r_i^{\mathbf{0}} - 1)(n - c_j^{\mathbf{0}} - 1) - (r_i^{\mathbf{1}} - 1)(c_j^{\mathbf{1}} - 1)$. Note that, for example, $* \cdot \mathbf{1} - \boldsymbol{x} \cdot \mathbf{1}$ may belong to class $\boldsymbol{X}$ even if $*$ is of class $\mathbf{1}$.

Class of pivot	$f_{ij}^{\mathbf{1}}$	$f_{ij}^{\boldsymbol{x}}$	$f_{ij}^{\boldsymbol{X}}$
$\mathbf{1}$	$(r_i^{\mathbf{1}}-1)(c_j^{\mathbf{1}}-1)$	0	$(m-r_i^{\mathbf{0}}-1)(n-c_j^{\mathbf{0}}-1)$ $-(r_i^{\mathbf{1}}-1)(c_j^{\mathbf{1}}-1)$
$\boldsymbol{x}$	0	$r_i^{\mathbf{0}}(n-c_j^{\mathbf{0}}-1)$	$(m-r_i^{\mathbf{0}}-1)(n-c_j^{\mathbf{0}}-1)$

Table 1: Estimated fill-in depending on the pivot.

4.2 Deviating from the Gaussian Elimination Scheme

The Gaussian elimination scheme sometimes introduces unnecessary case splits of regimes that cannot be avoided even by a clever pivot selection strategy. Consider the matrix

$$\begin{pmatrix} x & 2-x \\ -1-x & -1+x \end{pmatrix}.$$

All entries are symbolic and any choice of pivot leads to branching. On the other hand, the following sequence of row transformations yields a matrix with no symbolic entries, hence there is a uniform solution of the corresponding equation system for the entire parameter space.

$$\begin{pmatrix} x & 2-x \\ -1-x & -1+x \end{pmatrix} \begin{matrix} \urcorner \\ \leftarrow\!\lrcorner_{+} \end{matrix} \rightsquigarrow \begin{pmatrix} x & 2-x \\ -1 & 1 \end{pmatrix} \begin{matrix} \leftarrow\!\urcorner^{+} \\ \lrcorner_{x} \end{matrix} \rightsquigarrow \begin{pmatrix} 0 & 2 \\ -1 & 1 \end{pmatrix}$$

The observation that sometimes an arbitrary sequence of row transformations is superior to the Gaussian elimination scheme is exploited by an algorithm we call *Column Simplification* and that is invoked whenever the symbolic Markowitz Criterion fails to select a pivot that is constant, or at least constant with respect to the current context. The algorithm focuses on one column of the matrix — hence its name — and applies a sequence of row transformations that reduce the degrees of the entries in this column.

The basic idea is to successively eliminate leading monomials until no more simplification of the column elements is possible. This is similar to Buchberger's algorithm for computing Gröbner bases. For efficiency first only a sequence of row operations is determined. In a second phase, these transformations are applied to the entire matrix. Note that all transformations in this sequence are by design equivalence transformation. They never multiply a row by a polynomial directly, but only add multiples of one row to another row.

4.3 Further Optimisations

Two optimisations concern the simplification of matrix entries after row transformations.

It is possible to divide all entries of a row by their gcd g, but when dividing by a symbolic expression, it has to be ensured that this expression cannot be zero under the current context. If $g \neq 0$ is not entailed by the context, then g is factored and the reduction is restricted to the factors that can be shown to be non-zero. An alternative strategy is to apply the cs-normal operation provided by the reasoning specialist to all matrix entries.

It is fairly unclear in which situations cs-normal is better than dividing by row gcds and vice versa. In our implementation, division by the row gcd is performed after each row

operation whereas cs-normal is only applied immediately after a branch. The latter will be referred to as *Simplify-after-Branch.*

In order to increase the likelihood of finding pivots that do not lead to branching, the implementation employs block pivot search and performs column exchanges.

5 Experimental Results

A first step in the evaluation of the proposed method is to measure its performance on equation systems that appear in practice. The algorithm was tested with an implementation in the MuPAD computer algebra system. Despite some effort by the authors to use a state-of-the-art package for Gröbner bases computation, none of these packages could be readily used within MuPAD 2.0.0 on our hardware platform. Instead, MuPAD's own implementation of Gröbner bases was used. Recomputation of Gröbner bases was avoided where possible. All experiments were conducted on a Sparc Ultra 5 with 350 MHz and 192 MBytes of main memory.

5.1 The Corpus

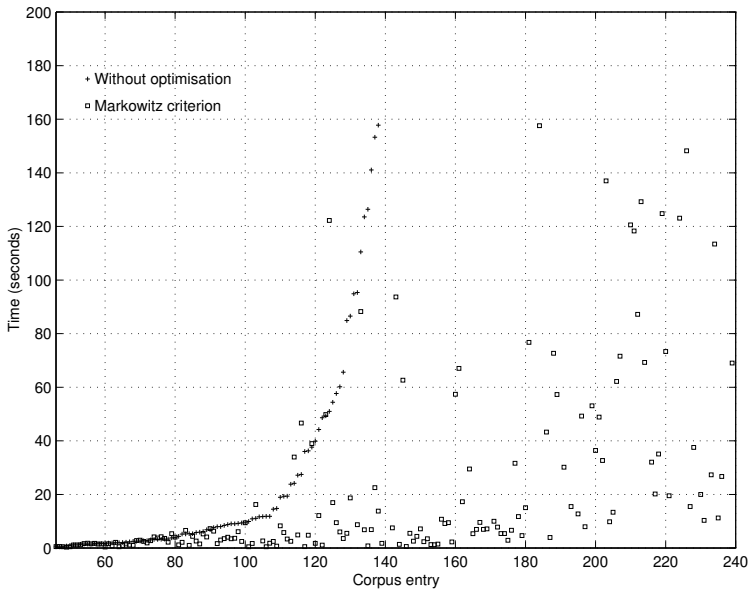
Although parametric linear systems arise in the solution of differential equations — for example, when computing the characteristic solutions (Dautray and Lions, 1988) — and also in coloured Petri nets (Jensen, 1996) it turned out that a collection of test data was not available. We were only able to obtain a few symbolic matrices, and their number was insufficient to assess the effectiveness of our method. Instead, a corpus of randomly generated matrices was used.

For the experiments, a corpus of 540 matrices was randomly generated and then fixed for all experiments. These matrices vary in size from 4×4 to 6×6, in number of parameters from 0 to 3, in the maximum total degree of symbolic entries from 2 to 5, in number of symbolic entries from 0 to 12 and in number of zero entries from 0 to 12. Coefficients and constant entries are uniformly distributed from the set $\{-9, -8, \ldots, -1, 1, \ldots, 9\}$. Polynomials of degree k were generated by adding k randomly generated monomials of maximum degree k. More details on the composition of the corpus can be found in Table 4. The entire corpus is available at http://www4.in.tum.de/~ballarin/ in various formats. There also the raw timing data and the various solutions can be obtained.

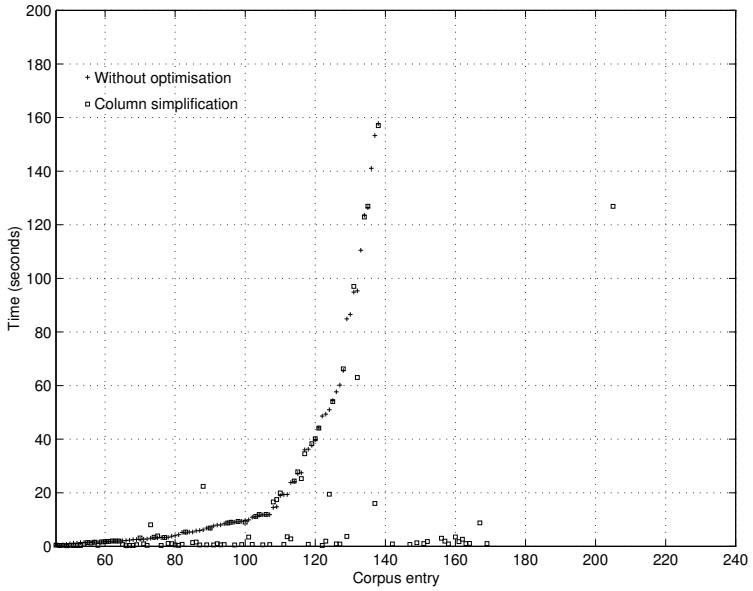
5.2 Effectiveness of the Strategies

Within a time limit of 450 seconds, 145 matrices of the corpus could be solved without optimisations. This increased to 226 when the combination of all three strategies was used. See Table 4 for the distribution of solved matrices on the corpus' classes. The success rate was even higher when the Markowitz criterion was used as the only optimisation, but then the number of computed regimes is usually higher.

Figure 1 and Table 2 give more insight in the effect contributed by the strategies. These experiments are based on the 239 corpus elements that could be solved in at least one experiment (including by Sit's algorithm, see Section 5.3). Timeout was always 450 seconds.

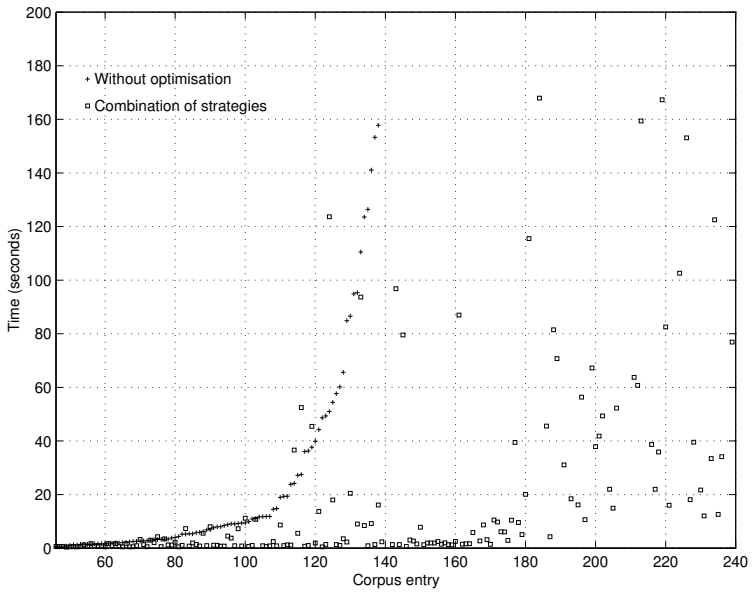


(a) Markowitz Criterion



(b) Column Simplification

Figure 1: Speed-up achieved by the strategies. Timeout 450 seconds. Corpus elements are sorted by runtime without use of strategies.



(c) Combination of all strategies

Figure 1 continued: Speed-up achieved by the strategies. Timeout 450 seconds. Corpus elements are sorted by runtime without use of strategies.

Regimes computed with Markowitz criterion	1	2	3	4	5	6	7	8	9	10	11	12	N
N	·	·	·	·	2	·	·	·	1	·	·	·	307
10	·	·	·	·	·	·	·	·	·	·	·	·	4
9	·	·	·	·	·	·	·	·	·	·	·	·	4
8	·	·	·	·	·	·	·	1	·	·	·	1	3
7	·	·	·	·	·	·	2	·	·	·	·	·	6
6	·	·	·	·	·	3	1	1	·	·	·	·	12
5	·	·	·	1	15	1	2	6	·	·	1	·	44
4	·	·	1	13	5	1	2	1	·	·	·	1	9
3	·	·	7	16	·	1	·	·	·	·	·	·	2
2	·	3	1	6	2	·	2	·	·	·	·	·	4
1	45	·	·	·	·	·	·	·	·	·	·	·	·

Regimes computed without optimisation

(a) Markowitz Criterion

Table 2: Comparision of the number of regimes computed with and without optimisation. Entries denote number of solved matrices from the corpus. N refers to entries that were not solved within 450 seconds.

Regimes computed with column simplification													
N	·	·	·	·	·	·	1	2	1	·	·	2	379
11	·	·	·	·	·	·	·	·	·	·	1	·	·
10	·	·	·	·	·	·	·	1	·	·	·	·	·
9	·	·	·	·	·	·	·	·	·	·	·	·	·
8	·	·	·	·	·	·	·	4	·	·	·	·	·
7	·	·	·	·	·	·	7	·	·	·	·	·	·
6	·	·	·	·	·	4	1	·	·	·	·	·	1
5	·	·	·	·	19	·	·	2	·	·	·	·	1
4	·	·	·	6	2	·	·	·	·	·	·	·	1
3	·	·	2	8	2	·	·	·	·	·	·	·	6
2	·	3	7	22	1	2	·	·	·	·	·	·	7
1	45	·	·	·	·	·	·	·	·	·	·	·	·
	1	2	3	4	5	6	7	8	9	10	11	12	N

Regimes computed without optimisation

(b) Column Simplification

Regimes computed by combination of strategies													
N	·	·	·	·	2	·	·	·	1	·	1	·	310
10	·	·	·	·	·	·	·	·	·	·	·	·	3
9	·	·	·	·	·	·	·	·	·	·	·	·	5
8	·	·	·	·	·	·	·	1	·	·	·	·	1
7	·	·	·	·	·	·	2	·	·	·	·	1	6
6	·	·	·	·	·	2	1	1	·	·	·	·	12
5	·	·	·	·	14	1	2	6	·	·	·	·	38
4	·	·	1	·	1	1	1	1	·	·	·	1	·
3	·	·	·	6	3	·	1	·	·	·	·	·	8
2	·	3	8	30	4	2	2	·	·	·	·	·	12
1	45	·	·	·	·	·	·	·	·	·	·	·	·
	1	2	3	4	5	6	7	8	9	10	11	12	N

Regimes computed without optimisation

(c) Combination of Strategies

Table 2 continued: Comparision of the number of regimes computed with and without optimisation. Entries denote number of solved matrices from the corpus. N refers to entries that were not solved within 450 seconds.

Computation of row gcds was turned on. In the diagrams the 45 matrices without parametric entries are omitted. Corpus elements are sorted by runtime without use of strategies. The remaining 94 matrices where elimination without strategies timed out follow after those.

The symbolic version of the Markowitz Criterion alone leads to a considerable reduction of computation time. Column Simplification can also lead to dramatic reduction of runtime. Figure 1(b) shows that Column Simplification does either have almost no effect or, when it can be applied, is very effective. It turns out that Column Simplification is most effective in the univariate case because then the likelihood of finding suitable row transformations is rather high. Simplify-after-Branch itself usually leads to a small increase of runtime. Its main benefit is that it reduces the degree of polynomials describing the regimes and lets appear solutions more natural.

A close inspection of the data reveals that most gain in runtime is linked to the reduction of regimes achieved by the strategies, and the connected savings in Gröbner basis computations.

5.3 Experimental Comparison with Sit's Algorithm

Sit's algorithm is based on the observation that suitable regimes can be constructed from the determinants of the minors of the matrix. Let A be the $n \times n$-matrix representing the linear system and r denote the greatest integer such that the ideal generated by all $r \times r$ subdeterminants of A is the whole polynomial ring. Each $c \times c$-subdeterminant ($c = r, \ldots, n$) gives rise to a regime candidate for the solution. In a second step, the system is solved for each consistent regime candidate. For a detailed description of Sit's algorithm, including refinements for reducing the number of regime candidates computed in the first step, we refer to the original article. Note that Gaussian elimination always returns disjoint regimes, while this is not the case for the regimes returned by Sit's algorithm.

The number of regimes in the final result is bounded by the number of regime candidates obtained in the first step. These are at most $\sum_i \binom{n}{i}^2 = \binom{2n}{n}$ (Sit, 1992, Theorem 4.1). In contrast, Gaussian elimination with branching introduces a lot of additional regimes in the worst case. A sharp bound for their total number is $\sum_i \binom{n}{i}^2 i!$ (Sit, 1992, Theorem 9.1). Nevertheless, we will see that in practice, the discrepancy is not as dramatic as suggested by the worst case analysis.

A direct runtime comparison of Sit's and our algorithm has to be done with care, even though both programs were run on the same machine, because the underlying software architectures of MuPAD and Axiom differ: MuPAD programs are interpreted while Axiom programs are compiled. We assume that this difference amounts to a linear factor in execution speed.[1] In addition to timings we again compare the number of regimes needed to cover the parameter space. The number of regimes is a measure of how adequate the analysis of the parameter space is: the algorithm that returns fewer regimes provides the better analysis.

Table 3 shows the comparision of branching Gaussian elimination (where all strategies are effective) with Sit's algorithm. The comparision is for the 190 matrices of the corpus

[1] Comparing the speed of Gröbner basis computation in both systems on an additional set of 150 polynomial basis consisting of polynomials with similar characteristics than those in the corpus showed that Axiom was about 2.5 times faster on average. However, the variation was quite large, being 2.4 on average.

Regimes computed by Sit's algorithm											
N	·	·	·	·	24	6	3	1	1	1	307
10	·	·	·	·	·	·	1	·	·	·	·
9	·	·	·	·	·	·	·	·	1	1	2
8	·	·	·	·	·	·	·	1	1	1	·
7	·	·	·	·	·	1	3	·	1	·	2
6	·	·	·	2	1	3	1	·	·	·	3
5	·	·	·	·	9	5	1	·	·	·	·
4	·	1	2	2	24	1	·	·	1	·	·
3	·	8	9	2	3	·	·	·	·	·	·
2	·	52	7	·	·	·	·	·	·	·	·
1	45	·	·	·	·	·	·	·	·	·	·
	1	2	3	4	5	6	7	8	9	10	N

Regimes computed by combination of strategies

Table 3: Comparision of the number of regimes computed by Sit's algorithm and ours. Entries denote number of solved matrices from the corpus. N refers to entries that were not solved within 450 seconds by the respective program.

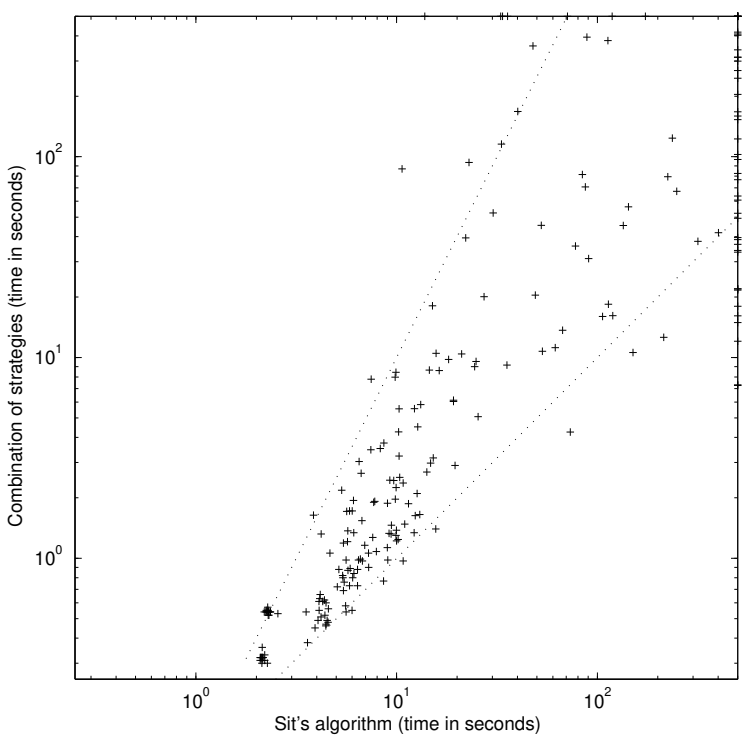


Figure 2: Comparison with Sit's algorithm. Timeout 450 seconds. Corpus entries where only one algorithm timed out are indicated on the upper and right hand side axis.

that could be solved with both our and Sit's algorithm in 450 seconds. The table shows that our algorithm usually only generates slightly more regimes than Sit's. On the other hand, there are also matrices where our algorithm needs less regimes to cover the parameter space. This underlines the effectiveness of the strategies.

The comparision of timings is shown in Figure 2. Each entry in the diagram documents, for one corpus element, the runtime of both Sit's algorithm and ours. By using a doubly logarithmic scale, we focus on an asymptotic comparision. To aid readability, the diagram also shows the curves $y = \frac{1}{10}x$ and $y = \frac{1}{10}x^2$ as dotted lines. The experiment shows that for the corpus data, apart from being faster by a constant factor, our algorithm is only about quadratically slower than Sit's. For some matrices this comparison is even linear. In view of Sit (1992, Theorem 9.1), which points out that branching Gaussian elimination is exponentially slower than Sit's algorithm in the worst case, this observation gives evidence that our strategies serve as very efficient heuristics for the problems under consideration.

6 Conclusions

Constraint algebraic programming can be applied to parametric linear equation systems successfully. The constraint algebraic programming version of the Gaussian algorithm computes a complete cover of the parameter space and — for each regime — a solution. The algorithm is also able to compute a cover for only part of the parameter space, if a suitable contraint store is supplied with the matrix.

A feature of contraint algebraic programming is that the base algorithm can be combined with suitable strategies. In the present example, strategies, namely Markowitz Criterion and Column Simplification increase the performance greatly and lead to a favourable comparision to Sit's algorithm — at least, for the corpus of sparse matrices used in our experiments. Understanding the connection between sparseness and the ability of our algorithm to merge regimes would be an interesting, but probably challenging, task.

The greatest advantage of constraint algebraic programming is its flexibility. The above algorithm is not restricted to systems where the parameter domain is polynomials. Any domain for which a reasoning specialist exists is suitable. An extension to parameters involving trigonometric functions over the reals could, for example, not only exploit the algebraic relation $\sin^2 x + \cos^2 x = 1$ but also $-1 \leq \sin x, \cos x \leq 1$. The latter would allow choosing $3 - \sin x$ as a pivot without branching.

To the surprise of the authors, it was hard to obtain symbolic matrices that could be used as test data. Although parametric linear equation systems occur naturally in many problems, a collection of such matrices was not available. We chose not to refine our strategies further, because fine tuning only makes sense in the context of "natural" problems. We make our own corpus of randomly generated matrices publicly available in the Internet (http://www4.in.tum.de/~ballarin/), together with the raw experimental data, and we also would like to encourage others to contribute parametric linear matrices arising in applications. Matrices may be submitted to the first author and will be made publicly available, too.

Matrix properties					Total	Solved with strategies					Sit's
Size	Vars	Deg	Symb	Zero	number	none	m	c	s	all	alg.
4	1	2	8	4	20	20	20	20	20	20	20
5	1	2	10	7	20	17	20	20	17	20	20
6	1	2	12	12	20	8	18	19	8	20	20
4	2	2	8	4	20	7	20	7	7	19	15
5	2	2	10	7	20	2	6	1	1	5	5
6	2	2	12	12	20	0	3	0	0	3	1
4	3	2	8	4	20	2	9	3	2	9	6
5	3	2	10	7	20	0	1	0	0	1	0
6	3	2	12	12	20	0	0	0	0	0	0
4	2	2	8	4	15	7	10	5	7	9	8
5	2	2	10	7	15	2	6	2	2	5	3
6	2	2	12	12	15	0	2	0	0	2	0
4	2	3	8	4	15	2	6	2	2	6	5
5	2	3	10	7	15	0	3	0	0	3	0
6	2	3	12	12	15	0	0	0	0	0	0
4	2	4	8	4	15	1	3	1	1	3	3
5	2	4	10	7	15	0	0	0	0	0	0
6	2	4	12	12	15	0	0	0	0	0	0
4	2	5	8	4	15	0	0	0	0	0	0
5	2	5	10	7	15	0	0	0	0	0	0
6	2	5	12	12	15	0	0	0	0	0	0
4	2	2	0	0	15	15	15	15	15	15	15
5	2	2	0	0	15	15	15	15	15	15	15
6	2	2	0	0	15	15	15	15	15	15	15
4	2	2	8	0	15	3	10	2	3	10	5
5	2	2	10	0	15	0	0	0	0	0	0
6	2	2	12	0	15	0	0	0	0	0	0
4	2	2	8	4	15	9	12	9	10	12	11
5	2	2	10	7	15	0	7	0	0	7	5
6	2	2	12	12	15	1	1	1	1	1	1
4	2	2	4	4	15	14	15	13	14	15	15
5	2	2	7	7	15	5	11	5	4	10	9
6	2	2	12	12	15	0	2	0	0	1	0
				Σ	540	145	230	155	144	226	197

Table 4: Corpus details. Matrix properties are size (Size), number of variables (Vars), maximum degree and simultaneously maximal number of monomials of the polynomial entries (Deg), number of symbolic entries (Symb) and number of zero entries (Zero). Strategies are abbreviated as follows: m Markowitz Criterion, c Column Simplification, s Simplify-after-Branch. The last column shows Sit's algorithm in comparison. The corpus is available at http://www4.in.tum.de/~ballarin/.

Acknowledgement

We would like to thank William Sit for providing the source code of his algorithm, and for valuable comments on a draft of this paper.

References

Cox, D., Little, J., and O'Shea, D. (1992). *Ideals, Varieties, and Algorithms.* Springer.

Dautray, R. and Lions, J.-L. (1988). *Mathematical Analysis and Numerical Methods for Science and Technology.* Springer.

Davenport, J. H., Siret, Y., and Tournier, E. (1993). *Computer Algebra: Systems and algorithms for algebraic computation.* Academic Press, second edition.

Duff, I., Erisman, A., and Reid, J. (1986). *Direct Methods for Sparse Matrices.* Clarendon Press.

Frühwirth, T. and Abdennadher, S. (1997). *Constraint-Programmierung: Grundlagen und Anwendungen.* Springer-Verlag.

Jensen, K. (1996). *Coloured Petri Nets.* Springer.

Noble, B. and Daniel, J. W. (1988). *Applied linear algebra.* Prentice-Hall, 3rd edition.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *Journal of the ACM*, **12**(1), 23–41.

Sit, W. (1992). An algorithm for solving parametric linear systems. *Journal of Symbolic Computation*, pages 353–394.

Weibel, T. and Gonnet, G. H. (1993). An assume facility for CAS, with a sample implementation for Maple. In J. Fitch, editor, *Design and implementation of symbolic computation systems: International Symposium, DISCO '92, Bath, U.K., April 13–15 1992: proceedings*, number 721 in LNCS, pages 95–103. Springer-Verlag.