Recognizing overdetermined subsystems of systems of equations

Arrigo Triulzi and Thomas Wolf

Mathematics Research Centre
Queen Mary & Westfield College, University of London,
Mile End Road, London E1 4NS, UK

A.Triulzi@qmw.ac.uk, T.Wolf@qmw.ac.uk

1 Introduction

Even if a system of equations to be solved is of the simplest possible form, i.e. consisting of linear algebraic equations, one may have problems using modern computers and computer algebra systems even when solving systems with only few unknowns (called functions in the text) if the coefficients are symbolic expressions in some independent variables. What is required is a memory efficient way of determining the sequence of functions to be eliminated. It can make a substantial difference whether one solves the linear system by inverting the coefficient matrix or by eliminating functions in a particular sequential order.

For n functions there are $n!$ different orders of elimination. Which of these is the computationally "least expensive"? What is meant by "least expensive"? As the computation time for manipulating symbolic expressions grows with their size, any saving in memory usage also speeds up the computation. But the size of expressions is hard to forecast as terms may unexpectedly cancel. Usually systems of equations start out sparse but with each computational step gradually all unknowns appear in all equations, so that on average the size of expressions grows. The aim is to find an elimination sequence which minimises the number of computational steps which unavoidably mix equations and increase the density of functions in them.

This problem was motivated when one of the authors worked on the computation of space time symmetries in General Relativity. It was possible to design a prolongation method which transformed the system of partial differential equations for the generators of these symmetries (Killing vectors but also Killing tensors) into a form where all integrability conditions are linear algebraic equations (see [2]). For example, for Killing tensors of rank two (with two indices) in a 4-dimensional space-time the set of integrability conditions consists of 300 algebraic conditions for 50 functions. Although these are 'only' 300 linear algebraic equations for 50 unknowns, with many of them being identities $0 \equiv 0$, the coefficients in the remaining conditions are not only numbers but also symbolic expressions in the 4 space-time coordinates. For many space-time metrics this over determined homogeneous system has only the trivial solution and hence no Killing tensors. On the other extreme for all space-time metrics with constant curvature all conditions reduce to identities with 50 solutions. For in-between cases like the Schwarzschild metric (describing a static single black hole) the systems has 11 solutions but for the more general Kerr metric (describing a single *rotating* black hole) 5 solutions are known but it is not clear whether these are all because the corresponding linear algebraic system is currently too complex to solve.

In section 3.1 we present an initial algorithm which we then refine in section 3.2. In section 4 results are shown which are obtained by timing different levels of sophistication of our algorithm.

This algorithm coded in RLISP[1] will find decompositions of any system of equations into subsystems and in this way give an indication as to the order in which to eliminate functions or solve equations. The work described is experimental as the authors did not find published work to build on. On the other hand it is application-driven and resulting implementations will be incorporated in the computer algebra package CRACK[3].

2 Initial simplifications

We will assume we have n equations $e_i,\ i = 1 \ldots n$ and m functions $f_j, j = 1 \ldots m$. A so-called dependency matrix $A_{ij}, i = 1 \ldots n, j = 1 \ldots m$ has components $A_{ij} = 1$ if e_i involves f_j otherwise the value is 0.

We will try to find a "decomposition" of the full problem into the solution of a system in, say, k equations ($k < n$) involving in total $\leq k$ functions followed by the solution of the remaining $n - k$ equations for the remaining functions.

In a different type of decomposition we try to find k functions ($k < m$) which occur in only $< k$ equations. This set of functions would be solved last.

The problem is more general than dealing with equations and functions. One equally could think of n curves passing through m discrete points. Given are the points through which each curve passes and consequently the curves going through each point. We could then ask for a set of k curves ($k < n$) which pass through at most k points in total.

A first simplifying step is to disregard all equations that involve at least one function that does not occur in any other equation. By dropping such an equation from further consideration, other functions may occur in only one of the remaining equations and so on. These equations would be solved last after the value of all functions which occur in other equations are known.

The other extreme would be equations which involve only one function. They would be solved first and the function be substituted in all other equations.

We see that the problem has a discrete symmetry:

1. Find subsets of k equations containing not more than k functions. (These equations would be solved first.)
2. Find subsets of k functions occuring in not more than k equations. (These equations would be solved last.)

Because of this symmetry we will concentrate on the first of both questions. For the second question the data for functions and equations simply have to be swapped, i.e. the transpose A_{ji} of the dependency matrix A_{ij} would be used.

The complete problem 1. amounts to checking all subsets of equations or subsets of functions whether they are suitable.

3 The method

3.1 A first procedure

In the following $|S|$ will denote the number of elements of the set S.

A recursive procedure to check all subsets of equations whether they are not under determined is:

1. Consider 3 sets of equations:
   - a set E_t of equations which turn up in the non-under determined subset which is to be found,
   - a set E_n of equations which definitely do not turn up in E_t and
   - a set E_u of equations not allocated yet to either E_t or E_n.
2. Initialize them as:
$$\begin{aligned} E_u &:= \{\text{all equations}\}, \\ E_t &:= \{\}, \\ E_n &:= \{\} \end{aligned}$$
3. If $E_t \neq \{\}$ and $|E_t| \geq |\text{functions in } E_t|$ then return E_t (success) else
4. if $E_n = \{\}$ then return *nil* (failure) else
5. pick one equation e_i of E_u and consider both of the following cases:
   (a) *either* move e_i from E_u to E_t and continue at step 3,
   (b) *or* move e_i from E_u to E_n and continue at step 4.

This method leads to 2^n case distinctions. Equivalently one could allocate functions by working with three sets of functions F_t, F_n and F_u. Success would then be recognized if $|\text{equations containing no functions other than } F_t| \geq |F_t|$. This leads to 2^m cases, i.e. fewer cases for $m < n$ but still too many for, say, $\min(m, n) = 20$.

3.2 Essential improvements

The improvements to the basic algorithm shown above involve incorporating a number of ideas:

1. *"Complete updating"*: working with and updating all six sets $E_t, E_n, E_u, F_t, F_n, F_u$ simultaneously, i.e. updating E_t, E_n, E_u too when allocating a function and updating F_t, F_n, F_u too when allocating an equation,
2. recognizing failure earlier: when $|E_u| = 0$ or $|F_u| = 0$ or $|F_t| > |E_t| + |E_u|$,
3. *"Dynamic switching"*: switching dynamically between allocating a function from F_u and allocating an equation from E_u according to some heuristic rule still to be defined.

Let us now describe both the complete updating and the dynamic switching in greater detail.

3.2.1 "Complete updating" of both equation and function lists

In order to achieve a maximal speed up we have to exploit any information about the six sets that is available.

Therefore the update of the six sets under step 2. in algorithm 3.1 now becomes:

- If an equation e_i is allocated to E_t then:
  1. all functions of e_i which are not already in F_t are moved to F_t,
  2. all equations involving only functions of the new F_t are moved to E_t as well.
- If an equation e_i is allocated to E_n then additionally move all equations from E_u to E_n which involve all functions of e_i which are not in F_t (Justification: One of the functions of e_i is the reason for allocating e_i to E_n, therefore all equations which involve all the $\{\text{functions of } e_i\}/F_t$ must also be moved to E_n.). If only one function of e_i is not in F_t then this function must be responsible for moving e_i to E_n and therefore this function is to be moved to F_n.

- If a function f_j is allocated to F_t then move all equations from E_u to E_t which only involve functions of the new (enlarged) F_t.
- If a function f_j is allocated to F_n then move all equations from E_u to E_n which involve f_j.

The overall aim is to lower the number of unallocated functions and equations ($|F_u|$ and $|E_u|$) as quickly as possible.

3.2.2 "Dynamic switching" between allocating a function or an equation

In each recursive call of steps 3–5 in the first version of our algorithm we have the choice of whether to allocate an equation from E_u to E_t and E_n or to allocate a function from F_u to F_t and F_n and furthermore the choice of which function or equation to allocate. Different heuristics were tested with different levels of sophistication.

The simplest choice is to always allocate only functions or only equations and, once the choice has been made, randomly pick a function or equation respectively (labelled **1f** and **1e** in figures 1, 2, and 3).

One improvement is to allocate a function if $|F_u| < |E_u|$ or an equation otherwise, in each case picking a random one as in the previous case (labelled **2** in the figures).

Another improvement is to choose the function or the equation respectively no longer at random from their set but to choose either the function occurring in as many equations as possible or the equation involving the fewest possible functions (labelled **3** in the figures).

A further refinement to the above allocates the function which occurs in as many equations in E_u as possible or the equation which involves as few as possible functions in F_u (labelled **4** in the figures). This will be referenced as our "final version".

The power of dynamical switching becomes apparent, for example, if the distribution of functions is dense. Then allocating one equation to E_t would allocate many functions to F_t leaving few functions in F_u and therefore we would have only few case distinctions left.

4 Results

The improvements suggested in 3.2.1 are of benefit whatever the allocation strategy as we simply make use of all the information available. The choice of allocation strategy detailed in 3.2.2 required setting up a testing framework. It generates systems of m equations with n functions and different mean numbers of functions in each equation which we will refer to in the following as "densities". For each of these systems we then ran different allocation heuristics from section 3.2.2.

All measurements were done on a Digital PWS 500au running Digital UNIX 4.0D and the PSL REDUCE 3.7.

4.1 Tests of the different versions

Our first example is based upon timings given by the following system sizes:

1. 20 equations with 20 functions, density $2\ldots19$,
2. 12 equations with 28 functions, density $2\ldots19$,
3. 28 equations with 12 functions, density $2\ldots11$.

The last two unbalanced cases are not purely academic. For example the Killing Tensor problem mentioned earlier involves up to 300 conditions for 50 functions.

In figure 1 we have the plot given for case 1. where we can see that our final version significantly outperforms the others for low densities and is marginally better at higher densities.

If we then examine the plots (Figure 2 and 3), for cases 2. and 3. respectively, we notice that in these unbalanced cases the choice of allocation strongly favours equations or functions as is to be expected. Our final version is only slightly slower than the naïve version of picking only functions or equations due to the overhead involved with the heuristics. This is acceptable given that the difference is only minimal and our version has the best all-round performance.

4.2 Testing the final version for differing system sizes

We now wish to examine the behaviour of running time against differing system sizes. Ideally we would want the runtime to increase as little as possible with the system size. We consider systems with 6 to 26 equations and equally many functions. For each system densities are given as percentages on the horizontal axis. The plot in figure 4 shows this relationship.

It is noticeable that for high densities ($> 60\%$) where the likelihood of finding a proper subsystem is small there is very little increase in runtime which means that the risk of loosing time in unsuccessful tests is very small. Only when the density is low, i.e. when the chance of finding true subsystems is high then computation times increase with the size of the problem.

5 Conclusions

We presented an algorithm which performs a complete search of whether a system of equations can be decomposed into smaller subsystems to be solved one after the other.

The method is not specific to systems of equations but can be used for any similar combinatorial problems. The next step in this ongoing work will be to consider incomplete searches of much larger systems of 100 or more equations and functions.

A useful feature of the current version is that computation times grow only with the size of the system if the density of functions per equation is low, i.e. if the probability of finding a subsystem is high.

We are looking forwards to applying the program to a range of Killing Vector and Killing Tensor problems.

References

[1] Anthony C. Hearn. *REDUCE User's Manual: Version 3.7*. RAND Corp., 1998.

[2] Wolf T. Structural equations for Killing tensors of arbitrary rank. *Comp. Phys. Commun.*, 115:316–329, 1998.

[3] Wolf T. and Brand A. *The Computer Algebra package CRACK for investigating PDEs.* Queen Mary and Westfield College, November 1998. Part of the REDUCE 3.7 manual and also `http://reduce.maths.qmw.ac.uk/packages/crack/`.

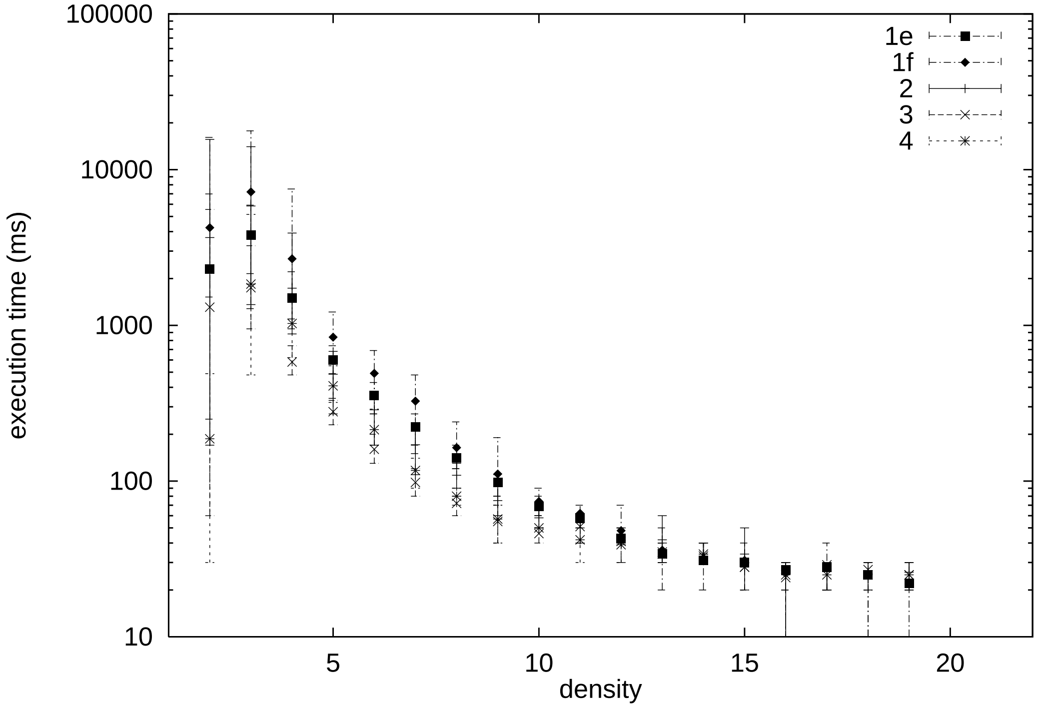

Figure 1: Plot of ten averaged runs of random systems of 20 equations and 20 functions, with density varying between 2 and 19.

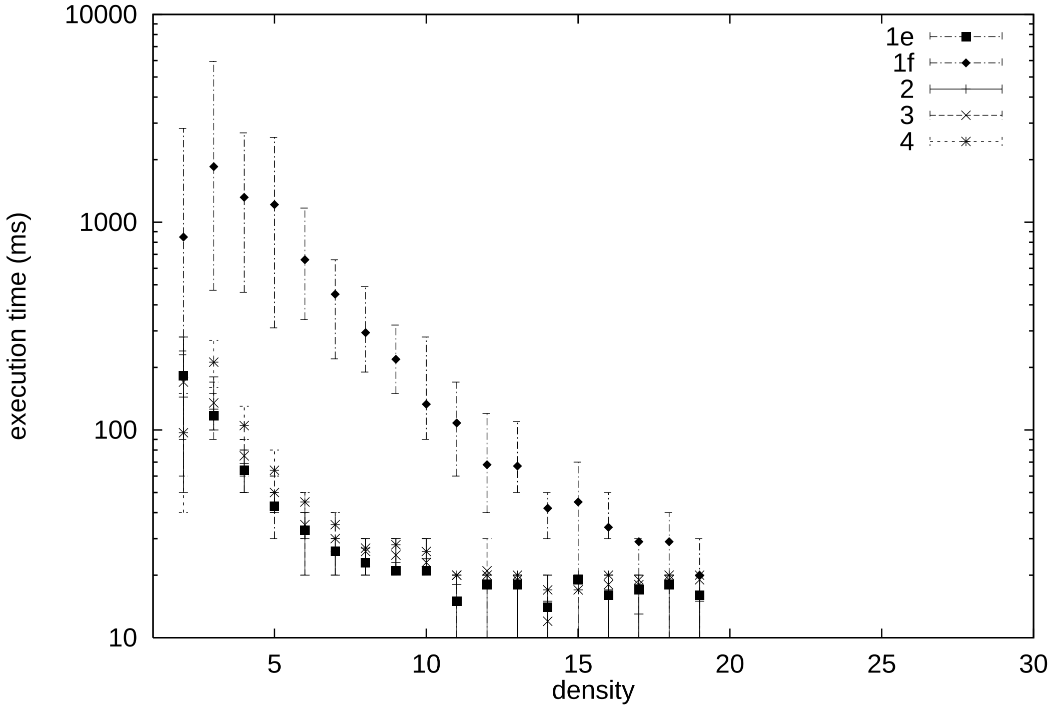

Figure 2: Plot of ten averaged runs of random systems of 28 equations and 12 functions, with density varying between 2 and 11.

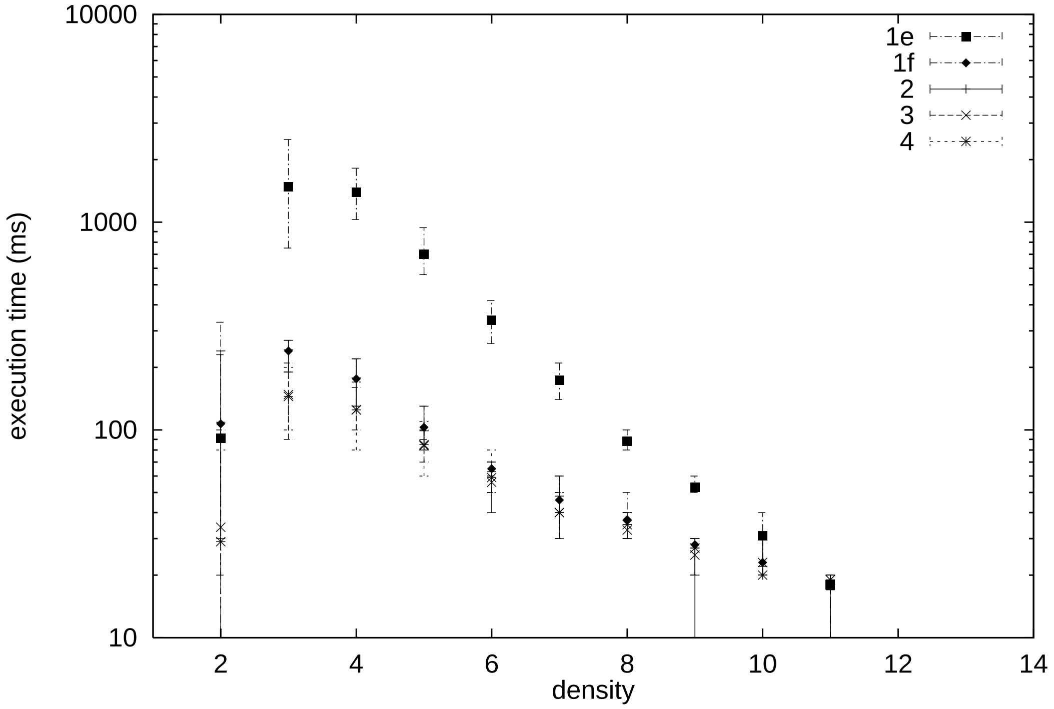

Figure 3: Plot of ten averaged runs of random systems of 12 equations and 28 functions, with density varying between 2 and 27.

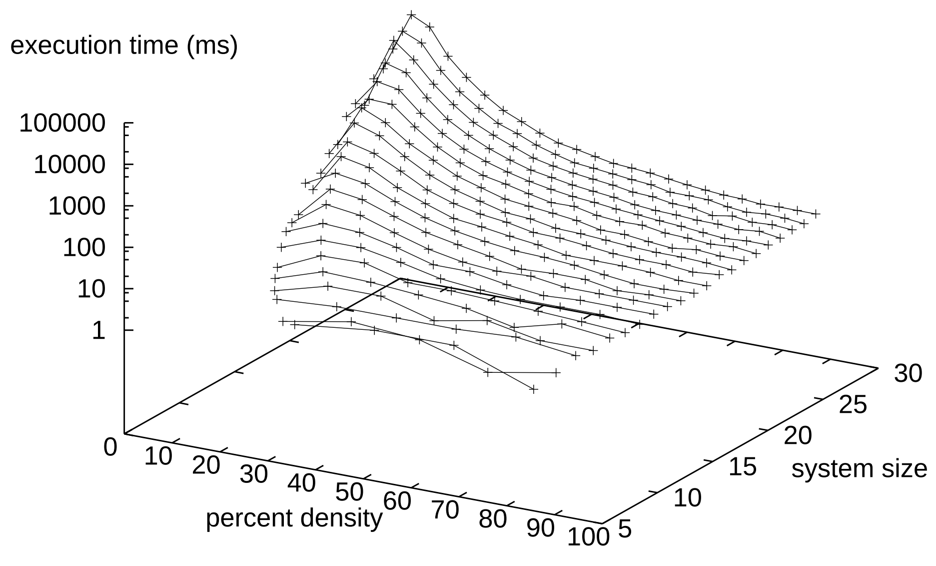

Figure 4: Plot of running time versus system size and density.