Hierarchical representations with signatures for large expression management

Wenqin Zhou[1], J. Carette[2], D.J. Jeffrey[1], M.B. Monagan[3]

[1] University of Western Ontario, London ON, Canada, N6A 5B7
{wzhou7, djeffrey}@uwo.ca
[2] McMaster University, Hamilton, ON, Canada, L8S 4L8
carette@mcmaster.ca
[3] Simon Fraser University, Burnaby, B.C. Canada, V5A 1S6
monagan@cecm.sfu.ca

Abstract. We describe a method for managing large expressions in symbolic computations which combines a hierarchical representation with signature calculations. As a case study, the problem of factoring matrices with non-polynomial entries is studied. Gaussian Elimination is used. Results on the complexity of the approach together with benchmark calculations are given.

Key Words: Hierarchical Representation, Veiling Strategy, Signature, Zero Test, Large Expression Management, Symbolic LU Decomposition, Time Complexity

1 Introduction

One of the attractions of MAPLE is that it allows users to tackle large problems. However, when users undertake large-scale calculations, they often find that expression swell can limit the size of the problems they can solve [31]. Typically, users might meet two types of expression swell: one type we can call *inherent* expression swell, and the other *intermediate* expression swell.

A number of strategies have been proposed for coping with the large expressions generated during symbolic computation. We list a number of them here, but lack of space precludes an extensive discussion.

- *Avoid the calculation.* This strategy delays computation of a quantity whose symbolic expression is large until numerical data is given. For example, if the determinant of a matrix is needed in a computation, one uses an inert function until the point at which the elements of the matrix can be evaluated numerically, and then jumps to a numerical evaluation.
- *Use signatures.* See, for example, [5]. Signatures are one of the ideas used in this paper.

– *Use black-box calculations.* This is a strength of the Linbox project [7].
– *Approximate representations.* This is the growing area of symbolic-numeric computation.
– *Use hierarchical representations.* These are studied in this paper, and the term will be abbreviated to HR.

Each of the above methods is successful for a different class of problems. This paper addresses a class of problems in which large expressions are built up from identifiable sub-expressions, and which as a result are suitable applications for hierarchical representations (HR). Hierarchical representations *per se* are not new in computer algebra. Similar ideas have appeared in the literature under a variety of names. Examples are as follows:

– *Maple DAGs.* Expressions in MAPLE are represented as DAGs with sub-expressions being reused and hence stored only once. For example, [13] uses this data structure to compute the determinant of polynomial matrices.
– *Straight-line programs.* The DAGWOOD [9] system computes with straight-line programs.
– *Common subexpression identification.* The MAPLE command `codegen[optimize]` searches a large expression for common subexpressions. (Also available as an option to commands in `CodeGeneration`) [16]
– *Computation sequences and* MAPLE*'s* `CompSeq`*.* An early example is given by Zippel in 1993 [11]. The function `CompSeq` in MAPLE is a placeholder for representing a computation sequence.
– *Large Expression Management (LEM).* This term was introduced in [10], and is the name of a MAPLE package.

The goal of this work is the combination of HR with signatures. We do this by modifying the `LargeExpressions` package in MAPLE and then applying it to a case study. The case study comes from DYNAFLEX [8], a system which computes the equations of motion for a mechanical device created from rigid or flexible bodies. It uses MAPLE for its computations and requires the factoring of matrices whose elements are multivariate polynomials or non-polynomial functions. In this paper, therefore, we consider the factoring of matrices with elements that are multivariate polynomials and exponential polynomials. We could have considered any application where the algorithm at hand only requires zero-recognition on the elements (as well as basic "arithmetic" operations); if obtaining other information, like degree or structural "shape" is absoluately necessary, this would need new ideas on top of the ones we present here.

2 Hierarchical representation

The first point to establish is the need for a modified HR implementation. We begin by giving our definition of HR for this paper, with the purpose of distinguishing our implementation from similar definitions, such as straight-line programs.

Definition 1 *An exponential polynomial p over a domain* $\mathbb{K}$ *and a set of independent variables* $\{x_1, ..., x_m\}$ *is a polynomial* $p \in \mathbb{K}[x_1, ..., x_m, y_1, ..., y_m]$ *with* $y_k = e^{x_k}, k = 1..m$.

Definition 2 *A hierarchical representation (HR) over a domain* $\mathbb{K}$ *and a set of independent variables* $\{x_1, ..., x_m\}$ *is an ordered list* $[S_1, S_2, ..., S_l]$ *of symbols, together with an associated list* $[D_1, D_2, ..., D_l]$ *of definitions of the symbols. For each* S_i *with* $i \geq 1$, *there is a definition* D_i *of the form* $S_i = f(\sigma_1, \sigma_2, ..., \sigma_k)$ *where* $f \in \mathbb{K}[\sigma_1, ..., \sigma_k]$, *and each* σ_j *is either a symbol in* $[S_1, S_2, ..., S_{i-1}]$ *or an exponential polynomial in the independent variables.*

Hierarchical representation is a more general idea than the (algebraic) straight-line program defined in [14] and used in [12, 9, 23, 27]. A given expression can have different HR, i.e. different lists of definitions $[D_1, D_2, ...]$. The strategy used to assign the symbols during the generation of expressions will be something that can be varied by the implementation. The reason for enquiring an ordered list is to exclude implicit definitions. Details on how to build HR are in the section 4.

Remark 1 *An important part of the creation of HRs is the order in which assignments happen. For instance to use* `codegen[optimize]`, *an expression is completely generated first. Clearly, some expressions will be too large to be generated explicitly, in which case* `codegen[optimize]` *would have nothing to work with.*

Remark 2 *There are many types of computational procedures which naturally generate HR. One example is Gaussian elimination, which we study here. Another is the calculation described in [10]. Other computations that are known to generate large expressions, for example Gröbner basis calculations, do not have a obvious hierarchy, although [13] hints at one.*

Remark 3 *One can understand HRs as a compromise between full computations and no computations. Enough of the computation is performed to give a correct result, but not so much that a closed-form can be output. It is a compromise between immediately returning a placeholder and never returning a giant result.*

The key issue is control over expression simplification; this includes the identification of a zero expression. In an ordinary computer algebra system, the usual way this proceeds is by normalizing expressions, a step which frequently destroys the HR and causes the appearance of additional expression swell. For example, most systems will normalize[1] the expression

$$(2781 + 8565x - 4704x^2)^{23}(1407 + 1300x - 1067x^2)^{19} - \alpha$$
$$(1809 + 9051x + 9312x^2)^{19}(2163 - 2162x + 539x^2)^{19} * (27 + 96x)^4(103 - 49x)^4$$

[1] normalization is often confused with simplification, but [29] argues otherwise.

by expanding it. The same strategy would be used by the system whether $\alpha = +1$ or $\alpha = -1$. However, in one case the result is zero, while in the other it is just a large expression, which now fills memory.

Consequently, the main purpose of creating user-controlled HR is to control normalization and to integrate more different (often more efficient) zero-testing strategies into a computation in a convenient way. As well as creating a HR, one must give equal importance to the prevention of its destruction.

The original `LargeExpressions` package in MAPLE was created as a result of the investigations in [10]. The authors had external mathematical reasons for knowing that their expressions were nonzero, and hence no provision was made for more efficient testing. In the current implementation, we intend to apply the resulting code more widely, with the consequent need to test efficiently for zero. This we do by incorporating signature testing.

The basic action is the creation of a label for a sub-expression. The command for this was given the name `Veil` in the original `LargeExpressions` package, and so this will be used as the general verb here. Once an expression has been veiled, the system treats it as an inert object unless the user or the program issues an unveiling command, which reveals the expression associated with the label.

3 Signatures

The idea of using signatures is similar to the probabilistic identity testing of Zippel-Schwartz theorems [2, 1], and to the basis of `testeq` in MAPLE by Gonnet [3, 4], also studied in [5, 6]. The original polynomial results of Zippel-Schwartz were extended to other functions in [3, 4, 15].

Since we need to apply our method to matrices containing exponential polynomials, we first define a signature function that is appropriate for this class of functions.

Definition 3 *Given an expression e, an exponential polynomial, the signature $s(e)$ with characteristic prime p is defined in the following steps.*

- *If e is a variable, then its signature equals a random value of $\mathbb{Z}/p\mathbb{Z}$.*
- *If $e = e_1 + e_2$ then $s(e) = s(e_1) + s(e_2) \mod p$.*
- *If $e = e_1 * e_2$ then $s(e) = s(e_1) * s(e_2) \mod p$.*
- *If $e = e_1^n$, where n is a positive integer, then $s(e) = s(e_1)^n \mod p$.*
- *If $e = a^x$ is an exponential function a^x, where a could be the base of natural logarithms or any (non-zero) number less than p, then $s(e) = r^t \mod p$, where r is a primitive root of p, and $t = x \mod \phi(p)$. Here $\phi(p)$ is Euler's totient function.*

Note that unlike [3], we explicitly do not treat towers of exponentials, but only simple exponentials, which is frequently sufficient in applications.

Proposition 1 *For all non-zero $y \in \mathbb{Z}/p\mathbb{Z}$, there exists a unique $x \in \mathbb{Z}/\phi(p)\mathbb{Z}$, s.t. $s(a^x) = y$.*

PROOF: By the definition of a signature $s(a^x)$, $r = s(a)$ is a primitive root modulo p. By the definition of a primitive root of a prime [26], the multiplicative order of r modulo p is equal to $\phi(p) = p - 1$. So the powers r^i, $i = 1..p-1$ range over all elements of $\mathbb{Z}/p\mathbb{Z} - \{0\}$. ■

For the following theorems, we suppose that all random choices are made under the uniform probability distribution.

Theorem 1 *(Zippel-Schwartz theorem) Given* $F \in \mathbb{Z}[x_1, ..., x_n]$, $F \mod p \neq 0$, *and* $\deg(F) \leq d$, *the probability* $Pr\{s(F) = 0 | F \neq 0\} \leq \frac{d}{p}$.

A proof can be found in [2, 1].

Theorem 2 *Let* $F \in \mathbb{Z}[y], y = a^x$, *where a could be the base of the natural logarithms or any (non-zero) number less than p,* $F \mod p \neq 0, \deg(F) = d$, *the probability* $Pr\{s(F) = 0 | F \neq 0\} \leq \frac{d}{p-1}$.

PROOF: The polynomial $F \in \mathbb{Z}[y]$ has at most d roots in $\mathbb{Z}/p\mathbb{Z}$. For $z \in \mathbb{Z}/p\mathbb{Z}$ such that $F(z) = 0$, Proposition 1 gives that there exists unique $u_z \in \mathbb{Z}/\phi(p)\mathbb{Z}$, s.t. $s(a^{u_z}) = z$. Thus the number of values x such that $s(F(a^x)) = 0$ is at most d. Because the total number of choices for nonzero y is $p - 1$, the probability $\Pr\{s(F) = 0 | F \neq 0\} \leq \frac{d}{p-1}$. ■

Theorem 3 *Let* $F \in \mathbb{Z}[x, y], y = a^x$, *where a could be the base of natural logarithms or any non-zero integer less than p,* $F \mod p \neq 0$, *and* $\deg(F) = d$, *the probability* $Pr\{s(F) = 0 | F \neq 0\} \leq \frac{d}{p-1}$.

PROOF: The polynomial $F \in \mathbb{Z}[x, y]$ has at most dp roots in $\mathbb{Z}/p\mathbb{Z}$. For (x_i, y_i) such that $F(x_i, y_i) = 0$, based on Proposition 1, there exists unique x_u, s.t. $s(a^{x_u}) = y_i$. If $x_u = x_i$, then the solution (x_i, y_i) is the one to make $s(F(x_i, a^{x_i})) = 0$. Therefore the number of roots for x, such that $s(F(x, a^x)) = 0$ is at most dp.

As the total number of (indpendent) choices for (x, y) is $p(p-1)$, the probability $\Pr\{s(F) = 0 | F \neq 0\} \leq \frac{d.p}{p(p-1)} = \frac{d}{p-1}$. ■

Signatures can be used to test if an expression is zero, as `testeq` does. However, `testeq` always starts fresh for each new zero-test. This is a source of inefficiency when the signature is part of a continuing computation, and will be seen in later benchmarks which use `testeq`.

The signature of the expression is computed before veiling an expression in HR. This value then becomes the signature of the veiling symbol. When that symbol itself appears in an expression to be veiled, the signature of the symbol is used in the calculation of the new signature. In particular, it is not necessary to unveil any symbol in order to compute its signature.

Other important references on this topic are [22, 21, 24]. Applications of this basic test is the determination of singularity and rank of a matrix [25] shows two applications of this basic technique: determining whether a matrix (of polynomials) is singular, and determining the rank of a polynomial matrix.

4 An Implementation of HR with signatures

The simplest method for tracking HRs is to maintain an association list between an expression and its (new) label. This is easily implemented via (hash) tables; one table associates a "current" number to a symbol (used as an indexed name to generate fresh labels), and another table which associates to each indexed name to the underlying (unveiled) expression. The indexed names play the role of the ordered list of symbols in definition 2. The main routine is `Veil[K](A)`. Here K is the symbol and A is the expression to be veiled. This routine stores the expression A in the slot associated to $K[c]$ where c is the "current" number, increments c and returns the new symbol. For interactive use, a wrapper function `subsVeil` can be used.

```
> subsVeil:=(e,A)->algsubs(e=Veil[op(procname)](e),A);
> A:= (x+y)^{10} + e^{x+y} + (x^2+1)^5 - 1:
> B:= subsVeil[K](x^2+1,A):
> C:= subsVeil[K](x+y, B);
```

$$k_2^{10} + e^{k_2} + k_1^5 - 1$$

Notice that there is no longer a danger of expanding the expression $(x^2+1)^5 - 1$ in a misguided attempt to simplify it. In order to retrieve the original expression, one uses `Unveil`.

```
> Unveil[K](C) ;
```

$$(x+y)^{10} + e^{x+y} + (x^2+1)^5 - 1$$

At present, the expressions corresponding to K are stored in the memory space of the implementation module[2]. After a computation is completed and the intermediate results are no longer needed. The memory occupied by K can be cleared using the command `forgetVeil(K)`.

The signature must be remembered between calls to `Veil`, as commented above. The signature could be attached more directly to K, or kept in a separate array specified by the user. The above implementation seemed to provide the best *information hiding*. Until we see, with more experience of case studies, which method is best, we have for the present implementation used the MAPLE facility `option remember` internally for handling some of the tables, for convenience and efficiency. Thus after a call to the routine `SIG`, the signature of any veiled expression is stored in an internal remember table and not re-computed.

The use of `Veil` to generate HRs together with the calculation of signatures will be called Large Expression Management (LEM). In fact it is just expression management, because the `Veil` tool can be used even on expressions which are not large, for the convenience they give to understanding algebraic structure.

[2] In other words, it is a stateful module, à la Parnas, which is also rather like a singleton class in OO.

5 LU Factoring with LEM

A well-known method for solving matrix equations is LU factoring, in which a matrix A is factored such that $PA = LU$, where L and U are triangular matrices and P is a permutation matrix; see [18] for further details. The MAPLE command `LUDecomposition` uses large amounts of memory and is very slow for even moderately sized matrices of polynomials. The large expression trees generated internally are part of the reason for this slowdown, but equally significant is the time taken to check for zero. For example,

```
> M :=Matrix(10,10,symbol=m);
> LinearAlgebra[LUDecomposition](M);
```

This LU factoring will not terminate. If the environment variable `Normalizer` is changed from its default of `normal` to the identity function, i.e. Normalizer:= $x- > x$, then the LU factoring can complete. This is why Large Expression Management requires both HR and signatures for its zero-test.

We modified the standard code for LU decomposition to include veiling and signature calculations. At the same time, we generalized the options for selecting pivots and added an option to specify a veiling strategy. One can see [30] for even more design points, and a general design strategy, for this class of algorithms.

Our LU factoring algorithm in high-level pseudo-code:

```
Get maximum_column, maximum_row for matrix A
  For current_column from 1 to maximum_column
    for current_row from current_column to maximum_row
      Check element for zero.
      Test element for being ‘‘best’’ pivot
      Veil pivot [invoke Veiling strategy]
      move pivot to diagonal, recording interchanges.
      row-reduce matrix A with veiling strategy
      store multipliers in L
    end do:
  end do:
return permutation_matrix, L, reduced matrix A
```

The function has been programmed with the following calling sequence.

```
   LULEM(A, K, p, Pivoting, Veiling, Zerotesting)
Parameters
   A           - square matrix
   K           - unassigned name to use as a label
   p           - prime
   Pivoting    - decide a pivot for a column
   Veiling     - decide to veil an expression or not
   Zerotesting - decide if the expression is zero.
```

5.1 Pivoting Strategy

The current MAPLE LUDecomposition function selects one of two pivoting strategies on behalf of the user, based on data type. Thus, at present, we have

```
> LUDecomposition(<<12345,1>|<1,1>>);
```

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 1/12345 & 1 \end{bmatrix}, \begin{bmatrix} 12345 & 1 \\ 0 & 12344/12345 \end{bmatrix}$$

even though

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 12345 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 1 \\ 0 & -12344 \end{bmatrix}$$

is more attractive. If the matrix contains floating-point entries, partial pivoting is used.

```
> LUDecomposition(<<1,12345.>|<1,1>>);
```

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 1. & 0. \\ (8.1)10^{-5} & 1. \end{bmatrix}, \begin{bmatrix} 12345. & 1 \\ 0 & 0.99992 \end{bmatrix}$$

Since we wished to experiment with different pivoting strategies, we made it an option. Rather than make up names, such as 'partial pivoting' or 'non-zero pivoting', to describe strategies, we allow the user to supply a function which takes 2 arguments. The function returns true if the second argument is a preferred pivot to the first argument. For example, the preferred pivoting strategy for the example above (choose the smallest pivot) can be specified by the function `(p1,p2)->evalb(abs(p2)<abs(p1))`. In a symbolic and veiling context there are a number of conceivable strategies which one might wish to try. These can be based on operation count, size of expression or number of indeterminants. However, the definition of LU factors only allows pivoting on one column, so no form of full pivoting is offered.

5.2 Veiling Strategy

In the same spirit of experimentation, we have used a function to specify a veiling strategy. This function takes one argument and returns true if the expression should be veiled. The current `LargeExpressions` package, for example, follows a strategy of ignoring integers. Thus an integer, however large, cannot be veiled at present. Similarly, integer content is extracted from expressions before veiling. Rather than make these decisions in advance, we leave them to the declaration of a veiling-strategy function.

Of particular interest is the 'granularity' of the HR, namely whether one veils every pairwise operation, or whether one waits until an expression of a pre-determined size is allowed to accumulate. In the former case, the HR would look similar to a straight-line program as defined in [14]. For our experiments, we have based our strategies on the MAPLE `length` command, as being a convenient measure of expression complexity.

5.3 Zero Test Strategy

We need to do zero tests to find pivots. This can also help us simplify our expressions, if needed. During the LU factoring, we use signatures to perform this test quickly (more precisely, in random polynomial time). It is important to note that for LU factoring, we only need to find a provably non-zero pivot, so that a false positive (an entry which seems to be zero but in fact is not) rarely leads to a problem. And, in that case, we can always resort to a full zero-test.

We use the signatures computed along with the hierarchical representations to do the zero test for the expressions in HR. But a user could choose any MAPLE commands, like `Normalizer`, `testeq`, `simplify` or `evalb`, to do the zero test. Which one is best depends on the application at hand.

6 Time complexity analysis for LU with veiling and signatures

Since our case study compares current LU factoring and LU factoring with expression management, it is important to have some measure of the time complexity of each procedure. We therefore start with the time complexity of conventional Gaussian elimination (see [19, 20, 17] for early work). Although some cases of the following theorems are "well known", there seem to be no convenient published statement of them.

Here we consider the time complexity measure is the number of bit operations, which can be rigorously defined as the number of steps of a Turing or register machine or the number of gates of a Boolean circuit implementing the algorithm. [28] Throughout, we make the simplifying assumption that entries grow linearly, in both degree and in coefficient size. This is actually optimistic, as growth is usually worse than this.

Theorem 4 *For a matrix $A = [a_{i,j}]_{n\times n}$, where n is the size of A, and $a_{i,j} \in \mathbb{Z}[x_1, \ldots, x_m]$, the time complexity of LU factoring for A is at least $\Omega(n^{2m+5})$ for naive arithmetic.*

PROOF: Let $\deg(a_{i,j}, x_i) < d_i, i = 1..m$, and the length of coefficient is at most l. Suppose the sizes of entries are growing, linearly, i.e. $d_{1k} = k.d_1, d_{2k} = k.d_2, ..., d_{mk} = k.d_m, l_k = k.l$. When we do LU factoring, at each step k, we have $(n-k)^2$ entries to manipulate. For each new entry from step $k-1$ to step k, we need to do at least one multiplication, one substraction and one division. The cost will be at least $\Omega((d_{1k}...d_{mk}.l_k)^2)$ for naive arithmetic.

The total cost for the LU factoring will be at least $\sum_{k=1}^{n-1}(n-k)^2 \times \Omega((k.d_1 \times k.d_2 \times ... \times k.d_m \times k.l)^2) = \Omega(d_1^2 d_2^2 ... d_m^2 l^2 n^{2m+5})$ (for naive arithmetic). ■

Corollary 1 *For a univariate matrix $A = [a_{i,j}]_{n\times n}$, where $a_{i,j} \in \mathbb{Z}[x]$, the time complexity of LU factoring is at least $\Omega(n^7)$ for naive arithmetic. It is $\Omega(n^5 \log n \log\log n)$ for using FFT multiplication.*

PROOF: Let $\deg(a_{i,j}(x)) < d$, the length of coefficient is at most l. For naive arithmetic, this is just the $m = 1$ case of the previous theorem. For the FFT bound, we only need duplicate the above proof (for $m = 1$ again) but use $c = \Omega((k.d.k.l)\log(k.d.k.l)\ \log\log(k.d.k.l))$ for the arithmetic cost (based on Schönhage & Strassen (1971), or Schönhage (1977) or Cantor & Kaltofen (1991) [28]). For FFT, the total cost for the LU factoring will be at least $\sum_{k=1}^{n-1}(n-k)^2 \times c$, which is $\Omega(n^5 \log n \log\log n)$. ■

With respect to the time complexity for LU factoring with veiling and signatures, we separate the time complexity analysis for LU factoring into two parts. Lemma 1 shows the time complexity for LU with veiling but without signature. Lemma 2 gives the time complexity for LU with signatures. The total cost will be the complexity for LU with veiling and signatures in Theorem 5.

This first lemma is valid for the following veiling strategy: we veil any expression whose coefficient length is larger than c_1, or whose degree in x_i is larger than c_2, where c_1, c_2 are positive constants. The cost for veiling an expression is $\Omega(1)$. Then the length of each coefficient will be less than $c = c_1 * c_2^m$ and the degree in x_i will be less than c_2.

Lemma 1 *For matrix $A = [a_{i,j}]_{n\times n}$, where n is the size of matrix A. The time complexity of LU factoring with large expression management (and the above veiling strategy) is $O(n^3)$.*

PROOF: Let $a_{i,j} \in \mathbb{Z}[x_1, ..., x_m]$, $\deg(a_{i,j}, x_k) < d_k$, $k = 1..m$, and the length of coefficient is at most l. At each step there are at most two multiplications, one division and one substraction. The cost of each step will be less than $4 \times O((c_1.c_2^m)^2) + O(1)$ for naive arithmetic. For each step k, one must perform arithmetic on $(n-k)^2$ elements, for a total cost of $\sum_{k=0}^{n-1}(n-k)^2 \times O((c_1.c_2^m)^2) = O(n^3)$ ■

To prevent the cost from growing exponentially with the number of variables, the above computation clearly shows that it is best to choose $c_2 = 1$.

Lemma 2 *For matrix $A = [a_{i,j}]_{n\times n}$, where n is the size of matrix A, and $a_{i,j} \in \mathbb{Z}[x_1, ..., x_m]$. The time complexity for computing signatures along the LU factoring will be $O((\log p)^2 n^3)$.*

PROOF: Let $\deg(a_{i,j}, x_k) < d_k, k = 1..m$, the length of coefficient is at most l, T be the maximum number of operations needed to evaluate $a_{i,j}$, $i = 1..n, j = 1..n$, for the original Matrix A. After this initial evaluation of all the entries of A, we only need at most four operations in $\mathbb{Z}/p\mathbb{Z}$ for computing the other entries' signatures at step k. These operations can all be done in $O((\log p)^2)$ for naive arithmetic.

We will compute all the signatures for the entries at each step, to greatly simplify zero-testing. So the total cost for computing signatures all along the LU factoring is $(n^2 \times T + \sum_{k=1}^{n-1}(n-k)^2 \times 4) \times O((\log p)^2) = (n^3 + (T - \frac{3}{2})n^2 + O(n)) \times O((\log p)^2) = O((\log p)^2 n^3)$ ■

Theorem 5 *For matrix $A = [a_{i,j}]_{n \times n}$, where n is the size of matrix A, and $a_{i,j} \in \mathbb{Z}[x_1, ..., x_m]$. The time complexity of LU factoring with veiling and signatures is $O(n^3(\log p)^2)$.*

PROOF: Immediate from the above two lemmas. ■

From Theorem 4 and Theorem 5, we can see the more the variables and the bigger the size of the matrix, the bigger the difference between the algorithms which are with and without veiling and signatures. These results agree completely with our empirical results.

7 Empirical results

We present some timing results. For the benchmarks described below, we use strategies based on MAPLE's `length` command. As these strategies are heuristics, any reasonable measure of the complexity of an entry is sufficient. The pivoting strategy searches for the element with the largest `length`. The `veiling` strategy depends on the type of matrix. For integer matrices, we veil all integers whose length is greater than 1000, while for polynomial matrices, the treshold is length 30. These constants reflect the underlying constants involved in the arithmetic for such objects.

For all benchmarks, three variations are compared: our own LU factoring algorithm with veiling and signatures, MAPLE's default `LinearAlgebra:-LUDecomposition`, and a version of `LinearAlgebra:-LUDecomposition` where `Normalizer` has been set to be the identity function and `Testzero` has been set to a version of `testeq`. We first had to "patch" MAPLE's implementation of `LUDecomposition` to use `Testzero` instead of an explicit call to `Normalizer(foo) <> 0`, and then had to further "patch" `testeq` to avoid a silly coding mistake that made the code extremely inefficient for large expressions[3]. All tests were first run with a time limit of 300 seconds. Then the first test that timed out at 300 seconds was re-run with a time limit of 1000 seconds, to see if that was sufficient for completion. Further tests in that column were attempted. Furthermore, the sizes of matrices used varies according to the results, to try and focus attention to the sizes where we could gather some meaningful results in (parts of) the three columns. All results are obtained using the TTY version of MAPLE10, running on an 1.8Ghz Intel P4 with 512Megs of memory running Windows XP SP2, and with garbage collection "frequency" set to 20 million bytes used, all results are for dense matrices. In each table, we report the times in seconds, and for the LEM column, the number in parentheses indicates how many[4] distinct labels (ie total number of veiled expressions) were needed by the computation, as an indication of memory requirements.

The reason for including the MapleFix column is to really separate out the effect of arithmetic and signature-based zero-testing from the effects of Large

[3] Both of these deficiencies were reported to MAPLESOFT and will hopefully be fixed in later versions of MAPLE

[4] and we use a postfix K or M to mean 10^3 and 10^6 as appropriate

Size	10	20	30	40	50	60	70	80	90	100	110
LEM	.03	.2	.8	2.3	6.1	12.5	17.8	27.6	42.4	56.4	75.4
	(0)	(0)	(0)	(0)	(148)	(902)	(2788)	(5948)	(12779)	(22396)	(36739)
Maplefix	.07	.2	.7	2.2	5.2	10.7	19.4	33.8	54.0	83.8	124.7
Maple	.04	.2	.7	2.2	5.2	10.5	19.2	32.6	52.8	85.8	123

Table 1. Timings for LU factoring of random integer matrices generated by `RandomMatrix(n,n,generator=`$-10^{12}..10^{12}$`)`. The entries are explained in the text.

Size	5	10	15	20	25	30	35	40	45	50
LEM	.12	.06	.18	.44	.87	1.9	3.0	4.5	7.8	9.1
	(26)	(237)	(872)	(2182)	(4417)	(7827)	(12K)	(19K)	(28K)	(39K)
MapleFix	.06	.07	.16	.30	.56	1.87	332	>1000	–	–
Maple	.53	1.5	9.3	39.2	110.4	269.8	431	845	>1000	–

Table 2. Timings for LU factoring of random matrices with univariate entries of degree 5, generated by `RandomMatrix(n,n,generator=(() -> randpoly(x)))`. The entries are explained in the text.

Size	5	10	15	20	25	30	35	40	45	50
LEM	.05	.09	.23	.49	.99	1.7	2.8	4.2	6.0	8.8
	(26)	(237)	(872)	(2182)	(4417)	(7827)	(12K)	(19K)	(28K)	(39K)
MapleFix	.06	.09	.20	.39	.75	3.2	949	>1000	–	–
Maple	35.3	>1000	–	–	–	–	–	–	–	–

Table 3. Timings for LU factoring of random matrices with trivariate entries, low degree, 8 terms `RandomMatrix(n,n,generator=(() -> randpoly([x,y,z], terms = 8)))`. The entries are explained in the text.

Size	5	10	15	20	25	30	35
LEM	.047	.078	.20	.51	.88	1.7	2.95
	(22)	(218)	(858)	(2163)	(4393)	(7798)	(12K)
MapleFix	.03	.08	.14	.30	.58	3.8	>1000
Maple	1.56	>1000	–	–	–	–	–

Table 4. Timings for LU factoring of fully symbolic matrix: `Matrix(n,n,symbol=m)`. The entries are explained in the text.

Size	5	10	15	20	25	30	35
LEM	.031	.094	.22	.50	.99	1.7	2.8
	(26)	(237)	(872)	(2182)	(4417)	(7827)	(12K)
MapleFix	xx	xx	xx	xx	xx	xx	xx
Maple	0.99	117	>1000	–	–	–	–

Table 5. Timings for LU factoring of random matrix with entries over $\mathbb{Z}[x, 3^x]$: `RandomMatrix(n,n,generator=(()->eval(randpoly([x,y],terms=8),y=`3^x`)))`. The entries are explained in the text.

Expression Management; MapleFix measures the effect of not doing polynomial arithmetic and using signatures for zero-recognition, and is thus expected to be a middle ground between the other two extremes.

Table 1 shows the result for random matrices over the integers. Only for fairly large matrices (between 90x90 and 100x100) does the cost of arithmetic, due to coefficient growth, become so large that the overhead of veiling becomes worthwhile, as the LEM column shows. Since integer arithmetic is automatic in MAPLE, it is not surprising that the MapleFix column shows times that are the same as the Maple column. Here the veiling strategy really matters: for integers of length 500, veiling introduces so much overhead that for 110x110 matrices, this overhead is still larger than pure arithmetic. For length 2000, no veiling at all occurs.

Table 2 shows the result for random univariate matrices, where the initial polynomials have degree 5 and small integer coefficients. The effect of LEM here is immediately apparent. What is not shown is that MapleFix uses very little memory (both allocated and "used"), while the Maple column involves a huge amount of memory "used", at all sizes, so that computation time was swamped by garbage collection time. Another item to notice is that while the times in the Maple column grow steadily, the ones in the MapleFix column are at first consistent with the LEM column, and then experience a massive explosion. Very careful profiling[5] was necessary to unearth the reason for this, and it seems to be somewhat subtle: for both LEM and MapleFix, very small DAGs are created, but for LEM we have full control of these, while for MapleFix, the DAGs are small but the underlying expression tree is enormous. All of Maple's operations on matrix elements first involve the element being *normalized* by the kernel (via the user-inaccessible `simpl` function), and then *evaluated*. While normalization follows the DAG, evaluation in a side-effecting language must folllow the expression tree, and thus is extremely expensive. Along with the fact that no information is kept between calls to `testeq`, causes the time to explode for MapleFix for 35x35 (and larger) matrices. Since the veiling strategy used for the last 4 tables is the same, it is not very suprising that the number of veilings is essentially the same. The reason that the all-symbolic is a little lower is because we start with entries of degree 1 and coefficient size 1, and thus these entries do not get veiled immediately. However, one can observe a clear cubic growth in the number of veilings, as expected.

Table 3 shows the result for random trivariate matrices, where the initial polynomials have 8 terms and small integer coefficients. The results here clearly show the effect that multi-variate polynomial arithmetic has on the results. Table 4 shows the results for a matrix with all entries symbolic, further accentuating the results in the trivariate case. Again, MapleFix takes moderate amounts of memory (but a lot of CPU time at larger sizes), while Maple takes huge amounts, causing a lot of swapping and trashing already for 10x10 matrices.

[5] Here we used a combination of procedure-level profiling via `CodeTools[Profiling]` and global profiling via `kernelopts(profile=true)`

Table 5 shows results for matrices with entries over $\mathbb{Z}[x, 3^x]$. Overall the behaviour is quite similar to bivariate polynomials, however the **xx** in the MapleFix entry indicate a weakness in MAPLE's `testeq` routine, where valid inputs (according to the theory in [3]) return `FAIL` instead. Our signature implementation can handle such an input domain without difficulty.

While we would have liked to present memory results as well, this was much more problematic, as MAPLEdoes not really provide adequate facilities to achieve this. One could look at **bytes used**, but this merely reflects the memory asked of the system, the vast majority of which is garbage and immediately reclaimed. This does measure the amount of overall memory *churn*, but does not give an indication of final memory use nor of the true *live set*. **bytes alloc** on the other hand measure the actual amount of system memory allocated. Unfortunately, this number very quickly settles to something a little larger than **gcfreq**, in other words the amount of memory required to trigger another round of garbage collection, for all the tests reported here. This reflects the huge amount of memory used in these computations, but does not reflect the final amount of memory necessary to store the end result. Neither can we rely on MAPLE's **length** command to give an accurate representation of the memory needed for a result because, for some unfathomable reason, **length** returns the expression tree length rather than the DAG length! Thus, for matrices whose results are un-normalized polynomials, we have no easy way to measure their actual size. As a proxy, we can find out the total number of variables introduced by the veiling process.

8 Acknowledgements

We wish to thank the anonymous ISSAC referee from 2005, who suggested we change the environment variable `Normalizer` when calling LUDecomposition. Thank Éric Schost for his suggestions and helps on this paper.

References

1. Schwartz J.T. Fast Probabilistic Algorithms for Verification of Polynomial Identities. *J. ACM*, 27(4): 701-717, 1980.
2. Zippel R. Probabilistic Algorithms for Sparse Polynomials. *Proc. of Eurosam 79'*, Springer-Verlag LNCS, **72**, 216–226, 1979.
3. Gonnet G.H.. Determining Equivalence of Expressions in Random Polynomial Time, Extended Abstract, *Proc. of ACM on Theory of computing*, 334-341, 1984.
4. Gonnet G.H. New results for random determination of equivalence of expressions. *Proc. of ACM on Symbolic and algebraic comp.*, 127–131, 1986.
5. Monagan M.B. Signatures + Abstract Types = Computer Algebra − Intermediate Expression Swell. PhD Thesis, *University of Waterloo*, 1990.
6. Monagan M.B. Gauss: a Parameterized Domain of Computation System with Support for Signature Functions. *DISCO*, Springer-Verlag LNCS, **722**, 81-94, 1993.
7. `http://www.linbox.org`
8. `http://www.maplesoft.com/products/thirdparty/dynaflexpro/`

9. Freeman T.S., Imirzian G., Kaltofen E. and Yagati L. DAGWOOD: A system for manipulating polynomials given by straight-line programs. *ACM Trans. Math. Software*, 14(3):218-240, 1988.
10. Corless R.M., Jeffrey D.J., Monagan M.B. and Pratibha. Two Perturbation Calculation in Fluid Mechanics Using Large-Expression Management. *J. Symbolic Computation*, **11**, 1–17, 1996.
11. Zippel R. Effective Polynomial computation. *Kluwer*, 1993.
12. Kaltofen E. Greatest Common Divisors of Polynomials Given by Straight-Line Programs. *J. of the Association for Computing Machinery*, 35(1): 231-264, 1988.
13. Giusti M., Hägele K., Lecerf G., Marchand J. and Salvy B. The projective Noether Maple package: computing the dimension of a projective variety. *J. Symbolic Computation*, 30(3): 291–307, 2000.
14. Kaltofen E. Computing with polynomials given by straight-line programs I: greatest common divisors. *Proceedings of ACM on Theory of computing*, 131–142, 1985.
15. Monagan M.B. and Gonnet G.H. Signature Functions for Algebraic Numbers. *ISSAC*, 291–296, 1994.
16. Monagan M.B. and Monagan G. A toolbox for program manipulation and efficient code generation with an application to a problem in computer vision. *ISSAC*, 257–264, 1997.
17. Sasaki T. and Murao H. Efficient Gaussian elimination method for symbolic determinants and linear systems (Extended Abstract). *ISSAC*, 155–159, 1981.
18. W. Keith Nicholson. Linear Algebra with Applications, Fourth Edition. *McGraw-Hill Ryerson*, 2003.
19. Bareiss E.H. Sylvester's Identity and Multistep Integer-Preserving Gaussian Elimination. *Mathematics of Computation*, 22(103): 565-578, 1968.
20. Bareiss E.H. Computational Solutions of Matrix Problems Over an Integral Domain. *J. Inst. Maths Applics*, **10**, 68-104, 1972.
21. Heintz J. and Schnorr C.P. Testing polynomials which are easy to compute (Extended Abstract). *Proceedings of ACM on Theory of computing*, 262–272, 1980.
22. Martin W.A. Determining the equivalence of algebraic expressions by hash coding. *Proceedings of ACM on symbolic and algebraic manipulation*, 305–310, 1971.
23. Ibarra O.H. and Leininger B.S. On the Simplification and Equivalence Problems for Straight-Line Programs. *J. ACM*, 30(3): 641–656, 1983.
24. Ibarra O.H. and Moran S. Probabilistic Algorithms for Deciding Equivalence of Straight-Line Programs. *J. ACM*, 30(1): 217–228, 1983.
25. Ibarra O.H., Moran S. and Rosier L.E. Probabilitic Algorithms and Straight-Line Programs for Some Rank Decision Problems. *Infor. Proc. Lett.*, 12(5): 227–232, 1981.
26. Shoup V. A Computational Introduction to Number Theory and Algebra. *Cambridge University Press*, 2005.
27. Kaltofen E. On computing determinants of matrices without divisions. *ISSAC*, 342–349, 1992.
28. Gathen J. von zur and Gerhard J. Modern computer algebra *Cambridge : Cambridge University Press*, 1999.
29. Carette J. Understanding Expression Simplification. *ISSAC*, 72–79, 2004.
30. Carette J. and Kiselyov O. Multi-stage Programming with Functors and Monads: Eliminating Abstraction Overhead from Generic Code. *Generative Programming and Component Engineering*, 256–274, 2005.
31. S. Steinberg, P. Roach Symbolic manipulation and computational fluid dynamics. *Journal of Computational Physics*, 57, pp 251-284, 1985.