Generic Gram-Schmidt Orthogonalization by Exact Division[*]

Úlfar Erlingsson[1] Erich Kaltofen[2] David Musser[1]

[1]Department of Computer Science, Rensselaer Polytechnic Institute
Troy, New York 12180-3590, USA; {ulfar,musser}@cs.rpi.edu

[2]Department of Mathematics, North Carolina State University
Raleigh, North Carolina 27695-8205, USA; kaltofen@eos.ncsu.edu

Abstract

Given a vector space basis with integral domain coefficients, a variant of the Gram-Schmidt process produces an orthogonal basis using exact divisions, so that all arithmetic is within the integral domain. Zero-division is avoided by the assumption that in the domain a sum of squares of nonzero elements is always nonzero. In this paper we fully develop this method and use it to illustrate and compare a variety of means for implementing generic algorithms. Previous generic programming methods have been limited to one of compile-time, link-time, or run-time instantiation of type parameters, such as the integral domain of this algorithm, but we show how to express generic algorithms in C++ so that all three possibilities are available using a single source code. Finally, we take advantage of the genericness to test and time the algorithm using different arithmetics, including three huge-integer arithmetic packages.

1 Introduction

Given a basis $B = \{b_1, \ldots, b_n\}$ for $\mathbb{R}^n$ the Gram-Schmidt orthogonalization process, as described in e.g. [3], computes an orthogonal basis $B^* = \{b_1^*, \ldots, b_n^*\}$ for $\mathbb{R}^n$ such that $\langle b_i^*, b_j^* \rangle = 0$ for $1 \le j < i \le n$ using the ordinary inner product. For simplicity, we shall restrict ourselves to n-dimensional spaces; all our algorithms are easily transferred to work on lower-dimensional subspaces. In [9], statement following proof of (1.28) on p. 523, the authors hint at a method for computing B^* from B using exact division in the case where B spans a subspace of $\mathbb{D}^n$. The domain $\mathbb{D}$ is an integral domain with the added property that $\sum_i x_i^2 \neq 0$ when $x_i \neq 0$. This additional property guarantees that no division by zero occurs in any of the exact divisions. In this paper we fully develop this method.

Examples of such domains, which have great importance in the symbolic context, are the ring of polynomials over the integers. If the Gram-Schmidt process were performed without exact division on bases with such parametric entries, reductions by polynomial GCD computations would be performed on the intermediate rational function entries. Exact divisions can avoid the costly GCD computations, as in the more classical examples by Bareiss for Gaussian elimination [1] and by Brown and Traub in the subresultant PRS algorithm [7].

We also consider how the exact-division Gram-Schmidt orthogonalization can be programmed generically with different techniques of instantiation of the integral domain. These techniques can be briefly characterized as compile-time, run-time, or link-time instantiation.

Compile-time instantiation uses a programming language feature such as templates (in C++) or generic units (in Ada). Different instances of an algorithm produce separate copies of the code in the executable, each tailored to the particular instance. This is the method used exclusively in the C++ Standard Template Library [12, 11]. Since functions can be inlined and optimized, compile-time instantiation generally produces the fastest code but has the disadvantages of requiring recompilation of the full code for any changes and produces "code bloat" from the repetition of the code when more than one instance is required.

Run-time instantiation uses the more traditional method of passing pointers to functions; by varying the pointers at run time, different instances of an algorithm are invoked at different times. The indirection and function calling make run-time instantiation slower than compile-time methods, but avoid code bloat and require fewer full recompilations.

Link-time instantiation is a compromise between the other methods. An algorithm is written in terms of external functions and a particular set of functions is supplied using a separate compilation unit at link time. This method restricts an executable to having only a single instance of a generic algorithm, and the resulting code is potentially slower than compile-time instantiations since no inlining is possible. On the other hand the code is faster than with run-time instantiation because there is no run-time indirection in function calling. A similar approach was taken by Austin Lobo in his generic implementation of the block Wiedemann algorithm [6].

To the best of our knowledge, all previous discussions of generic programming have considered these methods mutually exclusive. In this paper we show how to combine them,

[*]This material is based on work supported in part by the National Science Foundation under Grants No. CCR–9319776 (second author) and No. CCR–9308016 (third author).

by expressing our Gram-Schmidt algorithm as a single C++ template that can be instantiated at compile time, link time, or run time.

We tested our exact-division Gram-Schmidt algorithm and our generic programming methods using several types of arithmetic. We discuss the details of creating different instances of the algorithm with these different arithmetics in section 4. Section 5 then gives some timing results for the different instances and instantiation methods with two C++ compilers.

2 Preliminaries

We assume a basis $B = \{b_1, \ldots, b_n\}$ with $b_i \in \mathbb{D}^n$ for $1 \le i \le n$ and will find its Gram-Schmidt basis $B^* = \{b_1^*, \ldots, b_n^*\}$ with $b_i^* \in \mathbb{D}^n$ for $1 \le i \le n$.

We first define a few quantities. If we set $B_l := (b_1, \ldots, b_l) \in \mathbb{D}^{n\times l}$ we can set, for $0 \le l \le n$,

$$d_l := \det(B_l^T B_l) = \det\left(\langle b_i \,,\, b_j\rangle_{\substack{1\le i\le l\\ 1\le j\le l}}\right) \tag{1}$$

We now note that $d_0 = 1$ and if we set $B_l^* := (b_1^*, \ldots, b_l^*) \in \mathbb{D}^{n\times l}$ we have that $\det(B_l^T B_l) = \det(B_l^{*T} B_l^*)$, and thus

$$d_l = \prod_{j=1}^{l} \|b_j^*\|_2^2 \quad \text{for} \quad 0 \le l \le n. \tag{2}$$

It is convenient to use the last observation to define $\beta_i := \|b_i^*\|_2^2 = \langle b_i^* \,,\, b_i^*\rangle$ and thus

$$\beta_i = \frac{d_i}{d_{i-1}} \quad \text{for} \quad 1 \le i \le n. \tag{3}$$

By the Gram-Schmidt process and (3) we have that

$$b_i^* = b_i - \sum_{j=1}^{i-1} \mu_{i,j} b_j^* \quad \text{for} \quad 1 \le i \le n, \tag{4}$$

$$\mu_{i,j} = \frac{\langle b_i \,,\, b_j^*\rangle}{\langle b_j^* \,,\, b_j^*\rangle} = \frac{\langle b_i \,,\, b_j^*\rangle}{\beta_j} \quad \text{for} \quad 1 \le j < i \le n. \tag{5}$$

We can now state the first of two lemmas (proofs of our lemmas are in appendix A):

Lemma 1. *Let d_l be as in* (1)*, b_i^* be as in* (4) *and $\mu_{i,j}$ as in* (5)*; we then have*

$$d_{i-1} b_i^* \in \mathbb{D}^n \quad \textit{for} \quad 1 \le i \le n, \tag{6}$$

$$d_l\Big(b_i - \sum_{j=1}^{l} \mu_{i,j} b_j^*\Big) \in \mathbb{D}^n \quad \textit{for} \quad 1 \le l < i \le n, \tag{7}$$

$$d_j \mu_{i,j} \in \mathbb{D} \quad \textit{for} \quad 1 \le j < i \le n. \tag{8}$$

We now note that for $1 \le i \le n$

$$\begin{aligned}\beta_i &= \left\langle b_i - \sum_{j=1}^{i-1} \mu_{i,j} b_j^* \,,\, b_i - \sum_{j=1}^{i-1} \mu_{i,j} b_j^* \right\rangle \\ &= \langle b_i \,,\, b_i\rangle - \sum_{j=1}^{i-1} \mu_{i,j}^2 \beta_j,\end{aligned} \tag{9}$$

and that for $1 \le j < i \le n$

$$\begin{aligned}\mu_{i,j} &= \frac{\left\langle b_i \,,\, b_j - \sum_{k=1}^{j-1} \mu_{j,k} b_k^* \right\rangle}{\beta_j} \\ &= \frac{\langle b_i \,,\, b_j\rangle - \sum_{k=1}^{j-1} \mu_{j,k}\mu_{i,k}\beta_k}{\beta_j},\end{aligned} \tag{10}$$

which allows us to state the second lemma:

Lemma 2. *Let d_l be as in* (1)*, b_i^* be as in* (4) *and $\mu_{i,j}$ as in* (5)*; we then have*

$$d_l \sum_{k=1}^{l} \mu_{j,k}\mu_{i,k}\beta_k \in \mathbb{D} \quad \textit{for} \quad 1 \le l < j < i \le n, \tag{11}$$

$$d_l \sum_{k=1}^{l} \mu_{i,k}^2 \beta_k \in \mathbb{D} \quad \textit{for} \quad 1 \le l < i \le n. \tag{12}$$

3 The Algorithm

We are now able to state our algorithm:

Algorithm 1. *Let $B = \{b_1, \ldots, b_n\}$ form a basis for $\mathrm{QF}(\mathbb{D})^n$, where $\mathrm{QF}(\mathbb{D})$ is the field of quotients of $\mathbb{D}$. The following algorithm computes:*

- *The d_k's of* (1)*.*
- *Vectors $\tilde{b}_1, \ldots, \tilde{b}_n$ such that $\tilde{b}_i = d_{i-1} b_i^*$ for $1 \le i \le n$, with b_i^* as in* (4)*.*
- *The values $\tilde{\mu}_{i,j} = d_j \mu_{i,j}$ with $\mu_{i,j}$ as in* (5)*.*

```
          $d_0 \leftarrow 1$
          for $i \leftarrow 1, \ldots, n$
              for $j \leftarrow 1, \ldots, i-1$
Loop 1:           $\sigma \leftarrow 0$
                  for $l \leftarrow 1, \ldots, j-1$
                      $\sigma \leftarrow \frac{d_l\sigma + \tilde{\mu}_{i,l}\tilde{\mu}_{j,l}}{d_{l-1}}$
                  $\tilde{\mu}_{i,j} \leftarrow d_{j-1}\langle b_i , b_j\rangle - \sigma$
Loop 2:       $\sigma \leftarrow 0$
              for $l \leftarrow 1, \ldots, i-1$
                  $\sigma \leftarrow \frac{d_l\sigma + \tilde{\mu}_{i,l}^2}{d_{l-1}}$
              $d_i \leftarrow d_{i-1}\langle b_i , b_i\rangle - \sigma$
              $\tilde{\mu}_{i,i} \leftarrow d_i$
Loop 3:       $a \leftarrow d_1 b_i - \tilde{\mu}_{i,1} b_1$
              for $l \leftarrow 1, \ldots, i-2$
                  $a \leftarrow \frac{d_{l+1}a - \tilde{\mu}_{i,l+1}\tilde{b}_{l+1}}{d_l}$
              $\tilde{b}_i \leftarrow a$
          $\tilde{b}_1 \leftarrow b_1$
```

A proof of the algorithm is to be found in appendix A.

4 A Generic C++ Implementation

Our C++ implementation of the preceding algorithm, which can be seen in appendix B, is generic in IDE, the integral domain element type. The algorithm itself is a C++ template function in IDE taking matrices of IDE as parameters. The template function is written using the normal C++ arithmetic operators, e.g., the product of two IDE variables a

and b is denoted $a * b$. This requires the IDE type to have the overloaded C++ assignment, addition, subtraction, multiplication and division operators defined on it

We designed our implementation so it could use any of the three instantiation strategies discussed earlier, compile-time, run-time and link-time. This was achieved by adding a "function-wrapper" around the underlying arithmetic functions, to provide a consistent interface, and by use of compile-time directives.

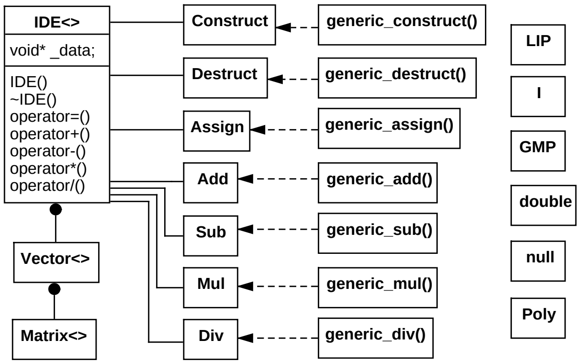

Figure 1: An overview of the C++ implementation.

Figure 1 shows an overview of our implementation. The underlying arithmetic is wrapped in the "*generic_*" functions to establish a consistent calling convention based on void pointers. If we are compiling for compile-time instantiation the bodies of the *generic_* functions are included in the source, otherwise they are compiled separately and linked in.

We then create function objects (see [12]) for the *generic_* functions and use them to instantiate the arithmetic type IDE, which in turn provides the required overloaded operators. The creation of these function objects takes place at run time in the case of run-time instantiation, but at compile time in the case of compile-time or link-time instantiation.

In appendix C we show the C++ code required for the addition function and two underlying implementations of it when using our strategy. The C++ code makes heavy use of compile-time directives to select the instantiation type. There are two function wrappers, *gmp_add* and *lip_add*, the source code for which is included if compile-time instantiation is required. One of the two wrappers is selected for use through renaming. We define a type *Add*, which is a pointer-to-function object in the case of run-time instantiation, but a derived ternary-function object otherwise. We finally define the *IDE* class, which has a member of the *Add* type that it uses for the addition operation.

To use the C++ algorithm, we instantiate simple *Vector* and *Matrix* classes with the IDE type. The template function for our algorithm, as mentioned earlier, accepts these matrices as parameters.

The implementation leaves many factors up to the underlying arithmetic, such as handling division by zero. The algorithm as presented in appendix B differs from our presentation in section 3 in a few minor ways. For instance the vectors and matrices are initialized to zero by default, and the a vector has been removed for the sake of efficiency. There are some additional differences due to the current lack of support for matrices in C++, such as having indices start at zero, but these should be clear from context.

5 Timing Results

We instantiated the C++ implementation with several arithmetics, in order to see what effects the instantiation strategy would have on the resulting running time. We chose three of the most popular freely-available long-integer arithmetic packages and two simple arithmetics for our trials. The long-integer packages were: Lenstra's LIP package [8], GNU's GMP package [4], and the I arithmetic of the LiDIA project [10]. The simple arithmetics were normal C++ double floating-point arithmetic, and null arithmetic, which only counts the number of calls to each operation.

We compiled the long-integer packages using GNU's gcc 2.7 C compiler, using full architecture-specific optimizations. We compiled the simple arithmetics and other code using the Apogee apCC 3.0 C++ compiler on SUNs, and the IBM xlC C++ compiler on the IBM. The C++ code was compiled for each architecture platform with full architecture-specific optimization and inlining turned on.

Operation	apCC	xlC
Constructions	3244359	6401509
Destructions	3163053	6320203
Assignments	753877	3911027
Additions	671650	671650
Subtractions	500150	500150
Multiplications	1833550	1833550
Divisions	651750	651750

Figure 2: Operation counts for optimized executables.

Figure 2 shows the results of a null-arithmetic run of executables created by the two compilers. The figure shows counts of the number of basic operations in a run, including the number of IDE constructor, destructor and assignment calls. As can be seen the apCC compiler proved far better at preventing the unnecessary creation of temporary variables, with only half as many IDE variables being constructed as with xlC.

Sparc7	I	LIP	GMP	double	null
Compile	2320.2s	3007.1s	2115.9s	59.9s	1.0s
Link	2328.9s	3008.8s	2115.2s	63.6s	7.7s
Run	2334.5s	3007.5s	2115.1s	65.1s	9.5s

Sparc8	I	LIP	GMP	double	null
Compile	512.6s	980.0s	450.0s	26.3s	0.4s
Link	499.9s	979.7s	451.8s	28.8s	4.1s
Run	501.3s	986.8s	442.1s	28.4s	4.3s

POWER	I	LIP	GMP	double	null
Compile	3558.2s	2883.6s	2637.0s	159.4s	4.0s
Link	3579.1s	2897.0s	2636.1s	173.2s	11.4s
Run	3545.7s	2848.6s	2620.7s	196.2s	20.1s

Figure 3: Execution times on three architectures, in seconds.

We timed the generated code on three different platforms: a Sparc7 architecture SUN SparcStation 2, a Sparc8 architecture SUN SparcStation 20, and an IBM POWER architecture IBM PowerStation 320H. The two SUNs differ in that the earlier Sparc architecture does not support full integer multiplication, only providing instructions to perform a multiplication in several separate steps. The inputs to the long-integer executables were lattices from the factoring algorithm of [9]. The input to the double and null executables was a random 100×100 matrix of single decimal digits.

The results of the timed runs can be seen in the tables of figure 3. The times shown are averages of several individual runs on the same input, run on the same workstation under identical conditions. Even though direct comparison between times on different architectures are not really meaningful, the difference in relative performance between the long-integer arithmetics is of some interest. Because of the difficulty of obtaining accurate and repeatable timings on multitasking workstations, the times of the longer runs are only accurate to about 2%.

The difference between instantiation strategies can be clearly seen with both of the simple arithmetics, double and null. With null arithmetic compile-time instantiation is much faster than link-time instantiation, with run-time instantiation being the slowest as expected. This difference disappears, however, when using non-trivial arithmetic, the slight differences there being due to timing inaccuracies. Even when using double arithmetic, run-time instantiation is only slightly slower than instantiating at compile time. Since the main difference between the three instantiation strategies is in function-call overhead, the strategies perform the same when using expensive arithmetic. However, the large difference for the null arithmetic shows that this overhead can be a critical factor in some situations.

The running times of the three long-integer arithmetics used for instantiation varies somewhat between the three platforms. GNU's GMP package runs fastest overall, with LiDIA's I arithmetic being about 10% slower on the SUNs. This is quite surprising, since the I arithmetic uses hand-optimized assembly code for its basic operations on the SUNs, but GMP is written purely in C. Lenstra's LIP arithmetic is the slowest of the three, which is likely due to the fact that portability was a main goal in its implementation. Even so it is faster than the I arithmetic on the IBMs, where the I arithmetic has no assembly code support.

The GMP long-integer arithmetic package seems to be overall fastest of those we tested. Even so users may want to benchmark the available arithmetics for their particular application, e.g. using the techniques presented in this paper, to see which one performs best for the operations required by it.

An early version of this paper, presented at the Dagstuhl Computer Algebra Systems meeting, had timing results based on optimized executables which were not optimized for a specific architecture. Those executables were in general about three times slower than those compiled with full architecture-specific optimizations. Therefore, setting the right compiler flags can be a more critical factor than selecting the fastest long-integer package.

To try our algorithm on more complex integral domains we also tried instantiating it with polynomial arithmetic. We used the Factory C++ polynomial arithmetic of [13] and ran the algorithm on an input of small univariate random polynomials, with good results. Adding support for polynomial arithmetic to our C++ implementation required only the addition of new wrapper functions to make calls to the Factory library. The inputs used in all the tests, along with the complete source code, are available on the Internet at URL `http://www.cs.rpi.edu/~ulfar/edgs/`.

6 Generic Bit Complexity Analysis

In this section we give the running time analysis in terms of bit complexity for our algorithm of section 3. Clearly, the algorithm performs $O(n^3)$ arithmetic operations in $\mathbb{D}$. The question to be answered is whether the lengths of operands remain bounded by some (possibly polynomial) function in the lengths of the entries in b_i and the dimension n. As the precise implementation of arithmetic in $\mathbb{D}$ is unknown, several considerations are in order.

Canonical representation: As an example, suppose that $\mathbb{D} = \mathbb{Q}$, the rational numbers and that the entries in b_i are integers on input. In a non-canonical model for $\mathbb{Q}$ each intermediate rational number is a pair of integers. No reduction to lowest terms is performed and instead in the computation of σ in loop 1, for example, the denominator of $d_l\sigma + \tilde{\mu}_{i,l}\tilde{\mu}_{j,l}$ is multiplied by d_{l-1}. In other words, no division is actually performed. A famous phenomenon, first described by [7], p. 414, eq. (27) for the polynomial GCD algorithm, occurs: the bit lengths of the intermediate numerators and denominators grow exponentially in n. Such immense size growth cannot occur when the rationals are reduced to relatively prime numerators and denominators. In fact, even in the polynomial GCD algorithm one obtains polynomial time bit complexity. We must therefore assume that the representation of any element in $\mathbb{D}$ is independent of how it is computed.

Bounded size growth: Under the canonical representation assumption for each element $x \in \mathbb{D}$ there is a unique bit-size $\ell(x)$ designating the storage requirements of x in the given model of arithmetic. It is, however, not sufficient to assume that the size of the result of each arithmetic operation is polynomially bounded by the combined size of the operands. Take, for example, as the computational arithmetic model for multivariate polynomials the sparse representation with respect to the power basis. Size growth for multiplication is then quadratic but even high powers of sparse polynomials can have exponential size, for instance $(x_1 + \cdots + x_n)^n$. For integer entries, the standard arithmetic operations have linear size growth and as we will see below this would be generically a sufficient assumption for obtaining polynomial bit complexity. Many useful domains, however, have superlinear growth for multiplication. An example is $\mathbb{Z}[x]$. The product of two degree d polynomials whose coefficients are integers with s bits, thus size $O(ds)$, may have size $\Theta(ds \log d)$. Furthermore, our generic analysis should let us distinguish between growth in multiplication/division and growth in addition/subtraction.

Efficient arithmetic: Clearly, each individual arithmetic operation must be efficient is some sense, say polynomial-time. One may consider efficiency in relation to the bit size of the answer. We observe that exact division can produce super-polynomially sized results. An example is again the sparse multivariate model and the division $\prod_{i=1}^{n}(x_i^n - 1) \Big/ \prod_{i=1}^{n}(x_i - 1)$. Both operands have 2^n terms while the quotient has n^n terms. Although we consider output-sensitive analysis somewhat unnatural, the formulation of our analysis can account for such cases.

We follow the approach of [5] and estimate the degrees and coefficients when the algorithm is performed on symbolic inputs. To make our estimates, we use 1-norms of multivariate polynomials over $\mathbb{Z}$, which for a given polynomial are defined as the sum of the absolute values of its integral coefficients. For all $f, g \in \mathbb{Z}[y_1, \ldots, y_k]$ we have the useful properties

$$\|fg\|_1 \leq \|f\|_1 \|g\|_1 \quad \text{and} \quad \|f+g\|_1 \leq \|f\|_1 + \|g\|_1.$$

Theorem 1. *Let $X = (x_{1,1}, \ldots, x_{n,n})$ and suppose that the j-th coordinate of b_i is set to $x_{i,j}$. Then if the algorithm is executed over the domain $\mathbb{D} = \mathbb{Z}[X]$, for any result τ that is assigned to a variable we have*

$$\deg_X(\tau) \leq 2n \quad \text{and} \quad \|\tau\|_1 \leq 2n^{2n}. \tag{13}$$

A proof is given in appendix A. We note that the estimates are quite crude and can be improved using Hadamard-like inequalities on the symbolic determinants. Our main point is that this theorem captures the bit complexity for most any model of arithmetic. For instance, if the inputs are integers bounded in absolute value by M, then any intermediately assigned value has no more than $O(n(\lg n + \lg M))$ bits.

Appendix

A Proofs

Proof of Lemma 1. To prove (6), following [9] we write $b_i^* = b_i - \sum_{j=1}^{i-1} \lambda_{i,j} b_j$ and take the inner product of both sides with b_k where $1 \leq k < i \leq n$. Since $\langle b_i^*, b_k \rangle = 0$ we have

$$\langle b_i, b_k \rangle = \sum_{j=1}^{i-1} \lambda_{i,j} \langle b_j, b_k \rangle,$$

which is a linear system of $i-1$ equations with $\lambda_{i,1}, \ldots, \lambda_{i,i-1}$ as the unknowns. Solving this system by Cramer's rule gives us $\det(B_{i-1}^T B_{i-1}) \lambda_{i,j} \in \mathbb{D}$ for $1 \leq j < i$ or by (1) $d_{i-1}\lambda_{i,j} \in \mathbb{D}$, thus (6).

The proof of (7) follows similarly to (6). We write

$$b_i - \sum_{j=1}^{l} \mu_{i,j} b_j^* = b_i - \sum_{j=1}^{l} \kappa_{i,j}^{(l)} b_j \tag{14}$$

and observe that $b_1, \ldots, b_l$ are orthogonal to this partial projection of b_i. In particular, we obtain

$$d_l \kappa_{i,j}^{(l)} \in \mathbb{D}. \tag{15}$$

To prove (8) we use (1), (5) and (6) and write

$$d_j \mu_{i,j} = d_j \frac{\langle b_i, b_j^* \rangle}{\beta_j} = d_{j-1} \langle b_i, b_j^* \rangle = \langle b_i, d_{j-1} b_j^* \rangle \in \mathbb{D},$$

which concludes the proof. □

Proof of Lemma 2. To prove (11) we make use of (1), (14), and (15) and get that

$$d_l \sum_{k=1}^{l} \mu_{j,k} \mu_{i,k} \beta_k = \left\langle b_i, d_l \sum_{k=1}^{l} \mu_{j,k} b_k^* \right\rangle = \left\langle b_i, d_l \sum_{k=1}^{l} \kappa_{j,k}^{(l)} b_k \right\rangle \in \mathbb{D}.$$

To prove (12) we consider the basis

$$B' = \{b_1, \ldots, b_l, b_i, b_{l+2}, \ldots, b_{i-1}, b_{l+1}, b_{i+1}, \ldots, b_n\},$$

for $1 \leq l < i \leq n$, which we form from the basis B by swapping the vectors b_i and b_{l+1}. We note that $d_l' = d_l$, $\mu_{l+1,j}' = \mu_{i,j}$ and $\beta_j' = \beta_j$ for $1 \leq j \leq l$. We then have, using (9) that

$$d_l \sum_{k=1}^{l} \mu_{i,k}^2 \beta_k = d_l \sum_{k=1}^{l} (\mu_{l+1,k}')^2 \beta_k' = d_l(\beta_{l+1}' - \langle b_{l+1}', b_{l+1}' \rangle) \in \mathbb{D},$$

since $d_l \beta_{l+1}' = d_l' \beta_{l+1}' \in \mathbb{D}$. □

Proof of The Algorithm. Let $\sigma_{i,j}^{(l)}$ denote the value of (11) or the variable σ after loop 1 has been executed for the control variable value l. We have that $\sigma_{i,j}^{(1)} = \tilde{\mu}_{i,1} \tilde{\mu}_{j,1} = d_1 \mu_{i,1} \mu_{j,1} \beta_1$ and since

$$\sigma_{i,j}^{(l)} = \frac{d_l \sigma_{i,j}^{(l-1)} + \tilde{\mu}_{i,l} \tilde{\mu}_{j,l}}{d_{l-1}} = \beta_l(\sigma_{i,j}^{(l-1)} + d_l \mu_{i,l} \mu_{j,l}) = d_l \sum_{k=1}^{l} \mu_{i,k} \mu_{j,k} \beta_k$$

we have by (10) and (11) that $\tilde{\mu}_{i,j} =$, and thus

$$\tilde{\mu}_{i,j} = d_j \mu_{i,j} = d_{j-1}(\langle b_i, b_j \rangle - \sum_{l=1}^{j-1} \mu_{i,l} \mu_{j,l} \beta_l) = d_{j-1} \langle b_i, b_j \rangle - \sigma_{i,j}^{(j-1)}.$$

Let $\sigma_i^{(l)}$ denote the value of (12) or the variable σ after iteration l of loop 2. We have that $\sigma_i^{(1)} = \tilde{\mu}_{i,1}^2 = \mu_{i,1}^2 \beta_1$ and since

$$\sigma_i^{(l)} = \frac{d_l \sigma_i^{(l-1)} + \tilde{\mu}_{i,l}^2}{d_{l-1}} = \beta_l(\sigma_i^{(l-1)} + d_l \mu_{i,l}^2) = d_l \sum_{k=1}^{l} \mu_{i,k}^2 \beta_k,$$

we have by (9) and (12) that $d_i = d_{i-1} \beta_i$, and thus

$$d_i = d_{i-1}(\langle b_i, b_i \rangle - \sum_{l=1}^{i-1} \mu_{i,l}^2 \beta_l) = d_{i-1} \langle b_i, b_i \rangle - \sigma_i^{(i-1)}.$$

Let $a_i^{(l)}$ denote the value of the variable a before loop 3 has been executed for the control variable value l, which corresponds to the value (7). We note that by (7) and (4) we assign the correct value to $\tilde{b}_2$ and $a_i^{(1)}$ corresponds to (11) for $l = 1$. If we assume that $a_i^{(l-1)}$ and $\tilde{b}_{l-1}$ are already computed correspondingly, we have that

$$a_i^{(l)} = d_l \left(b_i - \sum_{k=1}^{l} \mu_{i,k} b_k^* \right) = d_l \left(\frac{a_i^{(l-1)}}{d_{l-1}} - \mu_{i,l} b_l^* \right) \tag{16}$$

$$= \frac{d_l a_i^{(l-1)} - \tilde{\mu}_{i,l} \tilde{b}_l}{d_{l-1}} \tag{17}$$

and therefore that $\tilde{b}_i = a_i^{(i-1)}$ is correctly assigned. □

Proof of Theorem 1. From (1) we see that

$$\deg_X(d_k) \le 2k \quad \text{and} \quad \|d_k\|_1 \le k!n^k \le n^{2k}.$$

[A sharper Hadamard-like estimate is possible; see [2] and [7], §4.6.2, Exercise 21(a).] Let $\tilde{\kappa}^{(l)}_{i,j} = d_l \kappa^{(l)}_{i,j}$ for $\kappa^{(l)}_{i,j}$ as defined in the proof of Lemma 1, now computed with generic initial entries. By Cramer's rule we obtain from

$$\langle b_i \, , \, b_k \rangle = \sum_{j=1}^{l} \kappa^{(l)}_{i,j} \langle b_j \, , \, b_k \rangle, \quad 1 \le k \le l$$

the determinantal estimates

$$\deg_X(\tilde{\kappa}^{(l)}_{i,j}) \le 2l \quad \text{and} \quad \|\kappa^{(l)}_{i,j}\|_1 \le n^{2l}.$$

Similarly, using the above notation, we also have

$$\sigma^{(l)}_{i,j} = d_l \sum_{k=1}^{l} \mu_{i,k}\mu_{j,k}\beta_k = \left\langle b_i \, , \, \sum_{k=1}^{l} d_l \kappa^{(l)}_{i,k} b_k \right\rangle,$$

hence

$$\deg_X(\sigma^{(l)}_{i,j}) \le 2l+2 \quad \text{and} \quad \|\sigma^{(l)}_{i,j}\|_1 \le n^{2l+1}.$$

From these, we obtain easily

$$\deg_X(\tilde{\mu}_{i,j}) \le 2j \quad \text{and} \quad \|\tilde{\mu}_{i,j}\|_1 \le 2n^{2j-1}.$$

Furthermore,

$$\begin{aligned} \sigma^{(l)}_i &= d_l \sum_{k=1}^{l} \mu^2_{i,k}\beta_k \\ &= d_l(\beta'_{l+1} - \langle b'_{l+1} \, , \, b'_{l+1} \rangle) \\ &= d'_{l+1} - d_l \langle b'_{l+1} \, , \, b'_{l+1} \rangle \end{aligned}$$

thus (estimating crudely)

$$\deg_X(\sigma^{(l)}_i) \le 2l+2 \quad \text{and} \quad \|\sigma^{(l)}_i\|_1 \le 2n^{2l+2}.$$

The intermediate values of a defined above can be expressed by (14) and (17) as

$$a^{(l)}_i = d_l \left(b_i - \sum_{k=1}^{l} \kappa^{(l)}_{k,j} b_k \right)$$

and therefore

$$\deg(a^{(l)}_i) \le 2l+1 \quad \text{and} \quad \|a^{(l)}_i\|_1 \le 2n^{2l}.$$

Here the degree (norm) of a vector is the maximum degree (norm) of its entries. □

B The C++ Algorithm

Following is the C++ code for our algorithm.

```
template <class IDE>
IDE dot_product( Matrix<IDE>& A, size_type i,
                 Matrix<IDE>& B, size_type j )
{
  IDE dp = IDE(0);
  for( int k = 0; k < A.rows(); k++ )
    dp += A[k][i]*B[k][j];
  return dp;
}

template <class IDE>
void edgs( Matrix<IDE>& B,
           Matrix<IDE>& Bt,
           Matrix<IDE>& Mt )
{
  int m = B.rows();
  int n = B.cols();
  Vector<IDE> d( n+1 );
  IDE s;
  Bt = Matrix<IDE>( m, n );
  Mt = Matrix<IDE>( n, n );

  d[0] = IDE(1);
  for( int i = 0; i < n; i++ )
  {
    for( int j = 0; j <= i-1; j++ )
    {
      s = IDE(0);                        // Loop 1
      for( int l = 0; l <= j-1; l++ )
        s = (d[l+1]*s + Mt[i][l]*Mt[j][l])/d[l];
      Mt[i][j] = d[j]*dot_product(B,i,B,j) - s;
    }

    s = IDE(0);                          // Loop 2
    for( int l = 0; l <= i-1; l++ )
      s = (d[l+1]*s + Mt[i][l]*Mt[i][l])/d[l];
    d[i+1] = d[i]*dot_product(B,i,B,i) - s;

    Mt[i][i] = d[i+1];

    for( j = 0; j < n; j++ )             // Loop 3
      Bt[j][i] = d[1]*B[j][i] - Mt[i][0]*B[j][0];
    for( l = 0; l <= i-2; l++ )
      for( j = 0; j < n; j++ )
        Bt[j][i] = (d[l+2]*Bt[j][i]
                    -Mt[i][l+1]*Bt[j][l+1])/d[l+1];
  }
  for( i = 0; i < n; i++ )
    Bt[i][0] = B[i][0];
}
```

C Implementation Details

In this appendix we step through some of those parts of our C++ implementation which allow us to provide compile-time, link-time and run-time genericness in one C++ program.

```
#ifdef COMPILE_TIME_GENERIC
void
gmp_add( void*& dest,
         const void* src1, const void* src2 )
{
  mpz_add( (MP_INT*) dest,
           (MP_INT*) src1, (MP_INT*) src2 );
}
  ⋮
```

```
void
lip_add( void*& dest,
         const void* src1, const void* src2 )
{
  zadd( (verylong) src1,
        (verylong) src2, (verylong*) &dest );
}
⋮
#else /* LINK or RUN_TIME */
void
gmp_add( void*& dest,
         const void* src1, const void* src2 );
void
lip_add( void*& dest,
         const void* src1, const void* src2 );
⋮
#endif /* COMPILE_TIME */
```

The above code defines two wrapper functions for addition, one using GMP and the other LIP. Both functions are made to take the same parameters, i.e., have the same call interface, by using void pointers. If compile-time genericness is desired we provide the implementation of these functions, otherwise we just give prototypes to be resolved at link-time.

```
#ifdef COMPILE_TIME_GENERIC
#ifdef GMP
#define generic_add   gmp_add
#endif
#ifdef LIP
#define generic_add   lip_add
#endif
⋮
#endif /* COMPILE_TIME */
```

If compile-time genericness is required, we above select a particular arithmetic by renaming its wrapper functions to have the *generic_* prefix.

```
#ifdef RUN_TIME_GENERIC
typedef pointer_to_ternary_void_function
         <void*&,const void*,const void*>
         Add;
⋮
#else /* COMPILE or LINK_TIME */
struct Add: public ternary_function
        <void*&,const void*,const void*,void>
{
  void
  operator()( void*& dest,
              const void* src1,
              const void* src2 ) const
  {
    generic_add( dest, src1, src2 );
  }
};
⋮
#endif /* RUN_TIME */
```

Above we wrap the *generic_* arithmetic functions into function objects, with one object per operation. If run-time genericness is desired we have these be pointer-to-function objects, which allows us to change the pointers at run time. If, however, we want compile-time or link-time genericness, these objects are just another layer of wrapping, which will be resolved into direct function calls by the compiler.

Below we finally define the IDE class which gets instantiated with the function objects defined above. These objects are static members of the IDE class, and therefore neither consume any extra space per IDE variable, nor do they slow down copying of IDE objects. Each IDE instance has a void pointer to its data, and uses the function objects to perform operations on them.

```
template<...,class Add,...>
class IDE
{
  protected:
    void* _data;
    static Add _add;
    ⋮
  public:
    ⋮
    ide_type
    operator+( const ide_type& h ) const
    {
      ide_type t;
      _add( t._data, _data, h._data );
      return t;
    }
    ⋮
};
```

Acknowledgment: Imin Chen, then an undergraduate summer research student at Rensselaer Polytechnic Institute visiting from Queen's University at Kingston, Canada and now a Ph.D. student at Oxford University, provided an initial version of the exact division algorithm.

References

[1] Bareiss, E. H. Sylvester's identity and multistep integers preserving Gaussian elimination. *Math. Comp. 22* (1968), 565–578.

[2] Goldstein, A. J., and Graham, R. L. A Hadamard-type bound on the coefficients of a determinant of polynomials. *SIAM Rev. 16* (1974), 394–395.

[3] Golub, G. H., and Loan, C. F. V. *Matrix Computations*, 2 ed. The Johns Hopkins University Press, 1989.

[4] Granlund, T. *GMP 1.3.2, The GNU Multiple Precision Arithmetic Library.* FTP: `prep.ai.mit.edu:/pub/gnu/`, 1993.

[5] Kaltofen, E. Effective Noether irreducibility forms and applications. *J. Comput. System Sci. 50*, 2 (1995), 274–295.

[6] Kaltofen, E., and Lobo, A. Distributed matrix-free solution of large sparse linear systems over finite fields. In *Proc. High Performance Computing '96* (1996). To appear.

[7] Knuth, D. E. *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms*, 2 ed. Addison Wesley, 1981.

[8] Lenstra, A. K. *FreeLIP 1.0, A Free Long Integer Package.* FTP: `ftp.ox.ac.uk:/pub/math/freelip/`, 1989.

[9] Lenstra, A. K., Lenstra, H. W., and Lovász, L. Factoring polynomials with rational coefficients. *Math. Ann. 261* (1982).

[10] LiDIA-Group. *LiDIA 1.1, A Library for Computational Number Theory.* Universität des Saarlandes, FTP: `crypt1.cs.uni-sb.de:/pub/systems/LiDIA`, 1995.

[11] Musser, D. R., and Saini, A. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison-Wesley, 1996.

[12] Stepanov, A., and Lee, M. The standard template library. Technical Report HPL-94-34, Hewlett-Packard Laboratories, 1994.

[13] Stobbe, R. Factory: A C++ class library for multivariate polynomial arithmetic. Report on Computer Algebra 3, Centre for Computer Algebra, University of Kaiserslautern, 1996.

Úlfar Erlingsson is a Ph.D. student of Computer Science at Rensselaer Polytechnic Institute on a Fulbright scholarship. He received a B.S. in Computer Science from the University of Iceland in 1994, and a M.S. in Computer Science from Rensselaer Polytechnic Institute in 1996. His academic interests include many aspects of programming, such as languages, methodology, systems and environments, as well as compilers and symbolic computation.

Erich Kaltofen received both his M.S. degree in Computer Science in 1979 and his Ph.D. degree in Computer Science in 1982 from Rensselaer Polytechnic Institute. He was an Assistant Professor of Computer Science at the University of Toronto and an Assistant, Associate, and full Professor at Rensselaer Polytechnic Institute. Since 1996 he is a Professor of Mathematics at North Carolina State University. His current interests are in computational algebra and number theory, design and analysis of sequential and parallel algorithms, and symbolic manipulation systems and languages. Kaltofen was the Chair of ACM's Special Interest Group on Symbolic & Algebraic Manipulation 1993 - 95. He serves as associate editor on several journals on symbolic computation. From 1985 - 87 he held an IBM Faculty Development Award. From 1990 - 91 he was an ACM National Lecturer. He has contributed 20 papers to ISSAC and its predecessor conferences.

David Musser is Professor of Computer Science at Rensselaer Polytechnic Institute. He obtained his Ph.D in Computer Science from the University of Wisconsin. His research interests are design of generic software components, formal specification and verification of software, and automated proof of equational and inductive properties of software systems. The coauthor of two books and more than 50 papers, he was also a major contributor to the design and implementation of the Affirm verification system, the Ada Generic Library, and the C++ Standard Template Library.