Generic Gram-Schmidt Orthogonalization by Exact Division

Úlfar Erlingsson Erich Kaltofen David Musser

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. [GVL89], computes an orthogonal basis $B^* = \{b_1^*, \ldots, b_n^*\}$ for $\mathbb{R}^n$ such that $(b_i^*, b_j) = 0$ for $1 \leq j < i \leq n$ 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 [LLL82], 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 [Bar68] and by Brown and Traub in the subresultant PRS algorithm [Knu81].

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 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 [STL95]. Since functions can be inlined and optimized, compile-time instantiation generally produces the fastest code but has the disadvantages of (1) requiring recompilation of the full code for any changes and (2) "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 the algorithm are invoked at different times with a copy of the source code. 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. The 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 [KL96].

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

We first define a few quantities. If we set $B_k := (b_1, \ldots, b_k) \in \mathbb{D}^{n \times k}$ we can set

$$d_k := \det(B_k^T B_k) = \det\left((b_i, b_j)_{\substack{1 \leq i \leq k \\ 1 \leq j \leq k}}\right) \in \mathbb{D} \quad \text{for} \quad 0 \leq k \leq n. \tag{1}$$

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

$$d_k = \prod_{j=1}^{k} \|b_j^*\|_2^2 \quad \text{for} \quad 0 \leq k \leq n. \tag{2}$$

It is convenient to use the last observation to define

$$\beta_i := \|b_i^*\|_2^2 = (b_i^*, b_i^*) \quad \text{and thus} \quad \beta_i = \frac{d_i}{d_{i-1}} \quad \text{for} \quad 1 \leq i \leq n. \tag{3}$$

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

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

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

We can now state the first of our lemma (proof in appendix A):

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

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

$$d_{i-1}(b_k - \sum_{j=1}^{i-1} \mu_{k,j} b_j^*) \in \mathbb{D}^n \quad \textit{for} \quad 1 \leq i \leq k \leq n, \tag{7}$$

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

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

$$\beta_i = \left(b_i - \sum_{j=1}^{i-1} \mu_{i,j} b_j^*, b_i - \sum_{j=1}^{i-1} \mu_{i,j} b_j^*\right) = (b_i, b_i) - \sum_{j=1}^{i-1} \mu_{i,j}^2 \beta_j, \tag{9}$$

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

$$\mu_{i,j} = \frac{\left(b_i, b_j - \sum_{l=1}^{j-1} \mu_{j,l} b_l^*\right)}{\beta_j} = \frac{(b_i, b_j) - \sum_{l=1}^{j-1} \mu_{j,l} \mu_{i,l} \beta_l}{\beta_j}, \tag{10}$$

which allows us to state our next lemma (proof in appendix A):

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

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

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

3 The Algorithm

We are now able to state our algorithm (proof in appendix A):

Algorithm 1. *Let $B = \{b_1, \ldots, b_n\}$ form a basis for $L \subset \mathbb{D}^n$. 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 $1 \leq i \leq n$, with b_i^* as in (4).*
- *The values $\tilde{\mu}_{i,j} = d_j\mu_{i,j}$ with $\mu_{i,j}$ as in (5).*

$\tilde{\mu}_{i,j} \leftarrow 0 \ \forall \ 1 \leq i, j \leq n$
$d_0 \leftarrow 1$
$d_i \leftarrow 0 \ \forall \ 1 \leq i \leq n$
For $i \leftarrow 1, \ldots, n$
  For $j \leftarrow 1, \ldots, i-1$
Loop 1: $\sigma \leftarrow 0$
    For $l \leftarrow 1, \ldots, j-1$
      $\sigma \leftarrow \dfrac{d_l\sigma + \tilde{\mu}_{i,l}\tilde{\mu}_{j,l}}{d_{l-1}}$
    $\tilde{\mu}_{i,j} \leftarrow d_{j-1}(b_i, b_j) - \sigma$
Loop 2: $\sigma \leftarrow 0$
  For $l \leftarrow 1, \ldots, i-1$
    $\sigma \leftarrow \dfrac{d_l\sigma + \tilde{\mu}_{i,l}^2}{d_{l-1}}$
  $d_i \leftarrow d_{i-1}(b_i, b_i) - \sigma$
  $\tilde{\mu}_{i,i} \leftarrow d_i$
Loop 3: $a \leftarrow d_1 b_i - \tilde{\mu}_{i,1}b_1$
  For $l \leftarrow 1, \ldots, i-2$
    $a \leftarrow \dfrac{d_{l+1}a - \tilde{\mu}_{i,l+1}\tilde{b}_{l+1}}{d_l}$
  $\tilde{b}_i \leftarrow a$
$\tilde{b}_1 \leftarrow b_1$

4 A Generic C++ Implementation

Our C++ implementation of the preceding algorithm 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 C++ assignment, addition, subtraction, multiplication and division defined on it, as well as the normal comparison operations.

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.

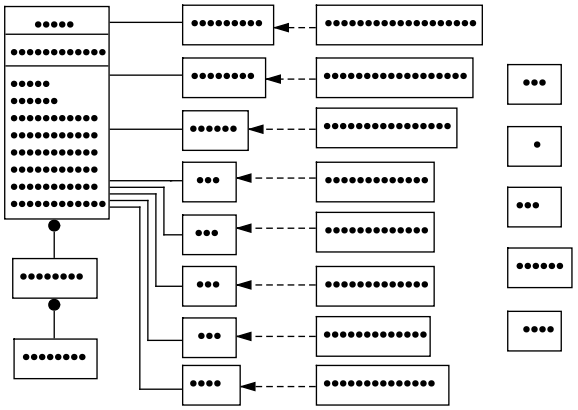

Figure 1: An overview of the C++ implementation.

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 [STL95]) 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*, for which the source code is provided 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 which 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 and normalization for comparisons. 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++, which should be clear from context.

5 Timing Results

We instantiated the C++ 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 are: Lenstra's LIP package [Len89], GNU's GMP package [GNU93], and the I arithmetic of the LiDIA project [LiD95]. 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 optimizations. We compiled the simple arithmetics other code on SUN SparcStations using the Apogee 3.0 C++ compiler, and on the IBM RS/6000 Power2 platform using the IBM xlC C++ compiler. The C++ code was compiled with full optimization and inlining turned on with each compiler. The Apogee compiler proved better at removing unnecessary temporaries, with only half as many IDE variables being generated as with the xlC compiler.

We timed the generated code on three platforms: SUN SparcStation 2, SUN SparcStation 20, and the IBM PowerStation 320H. The two SUN platforms have different architectures for normal integer multiplication. The inputs to the long-integer executables were lattices from the factoring algorithm of [LLL82]. The input to the floating-point executable was a random matrix of single decimal digits.

The results of the timed runs can be seen in the tables of figure 2. It should be noted that direct comparisons between architectures are not significant since the platforms varied in both clock-speed and system load. Due to inaccurate timing methods available to us, the times of the longer runs are only accurate to about 10%.

The difference between instantiation strategies can be clearly seen with both of the simple arithmetics, double and null. With the Apogee compiler and null

Sparc2	I	GMP	LIP	double	null
Compile	6429.7s	2096.0s	7583.1s	59.9s	1.0s
Link	6427.1s	2096.7s	7151.3s	63.6s	7.7s
Run	6412.3s	2097.8s	6748.5s	65.1s	9.7s

Sparc20	I	GMP	LIP	double	null
Compile	403.3s	781.7s	2005.0s	19.8s	0.2s
Link	425.6s	649.8s	2235.1s	20.5s	2.8s
Run	417.7s	774.7s	2146.5s	23.5s	4.2s

RS/6000	I	GMP	LIP	double	null
Compile	3979.7s	2573.3s	4558.7s	174.6s	4.1s
Link	4006.6s	2571.6s	4556.3s	172.1s	11.5s
Run	3982.9s	2574.3s	4557.3s	206.6s	20.2s

Figure 2: The execution times in seconds for three architectures

arithmetic compile-time instantiation is much faster than link-time instantiation, with run-time instantiation being the slowest as expected. However, this difference disappears when using non-trivial arithmetic. 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 greatly between the three platforms. GNU's GMP package runs fastest on two of the platforms, with LiDIA's I arithmetic being the fastest on the third platform. Lenstra's LIP arithmetic, which is likely the most portable of the three, is not surprisingly the slowest. The fluctuations between the running times of the packages on the three platforms suggests that their users should in general benchmark them on target platforms before deciding which one to use.

The inputs used in these tests, along with the complete source code, are available on the Internet at URL `http://www.cs.rpi.edu/~ulfar/edgs/`.

6 Additional Results

If accepted, the final submission to the proceedings will contain the following additional information.

We will provide a complexity analysis for the generic algorithm in terms of bit operations. Again, some assumptions on the bit cost of the individual operations of the domain $\mathbb{D}$ must be made. First, the cost of the arithmetic operations must be polynomial (quadratic, say) in the bit-sizes of the operands. Second, the bit-size of the result of an arithmetic operation must only grow as

a linear function in the combined bit-sizes of the operands. Note that the latter assumption is not satisfied for multivariate polynomials. Indeed, the Gram-Schmidt orthogonalization of a generic basis has exponentially sized entries, as it contains fractions of the generic determinants. Third and last, the actual element representation must be canonical with respect to the arithmetic operations. Otherwise, one may encounter the famous phenomenon of obtaining exponentially sized elements by keeping the occurring fractions unreduced, as in polynomial GCD computations [Knu81], p. 414, eq. (27). The bit complexity can then be estimated within polynomial bounds because of the lemmas stated in section 2. A similar, but more difficult analysis has been carried out for a generic factorization algorithm [Kal95].

We will also provide some test cases where the input basis contain polynomial entries.

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

[Bar68] E. H. Bareiss. Sylvester's identity and multistep integers preserving Gaussian elimination. *Math. Comp.*, 22:565–578, 1968.

[GVL89] Golub, G. H., Van Loan, C. F. *Matrix Computations*, 2nd Edition, The Johns Hopkins University Press, Baltimore, Maryland, 1989.

[GNU93] *GMP 1.3.2, The GNU Multiple Precision Arithmetic Library*, Torbjörn Granlund, FTP: `prep.ai.mit.edu:/pub/gnu/`, 1993.

[Kal95] E. Kaltofen. Effective Noether irreducibility forms and applications. *J. Comput. System Sci.*, 50(2):274–295, 1995.

[KL96] E. Kaltofen and A. Lobo. Distributed matrix-free solution of large sparse linear systems over finite fields. In *Proc. High Performance Computing '96*, to appear.

[Knu81] D. E. Knuth. *The Art of Computer Programming, Vol. 2, Seminumerical Algorithms, Ed. 2.* Addison Wesley, Reading, MA, 1981.

[Len89] *FreeLIP 1.0, A Free Long Integer Package*, Lenstra, A. K. FTP: `ftp.ox.ac.uk:/pub/math/freelip/`, 1989.

[LiD95] *LiDIA 1.1, A library for computational number theory*, LiDIA-Group, Universität des Saarlandes, FTP: `crypt1.cs.uni-sb.de:/pub/systems/LiDIA`, 1995.

[LLL82] Lenstra, A. K., Lenstra Jr., H. W., Lovász, L. *Factoring Polynomials with Rational Coefficients*, Mathematische Annalen, vol 261, 1982.

[STL95] Stepanov, A., Lee, M.. *The Standard Template Library*, Hewlett-Packard Laboratories Technical Report HPL-94-34, Palo Alto, California, April 1994, revised October 31, 1995.

Appendix

A Proofs

Proof of Lemma 1. To prove (6) 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_l where $1 \leq l < i \leq n$. Since $(b_i^*, b_l) = 0$ we have

$$(b_i, b_l) = \sum_{j=1}^{i-1} \lambda_{i,j} (b_j, b_l),$$

which is a linear system of $i-1$ equations with $\lambda_{i,1}, \ldots, \lambda_{i,i-1}$ as the unknowns. Solving this system gives us $\det(B_j^T B_j)\lambda_{i,j} \in \mathbb{D}$ for $1 \leq j < i$ or by (1) and (2) $d_{i-1}\lambda_{i,j} \in \mathbb{D}$, thus (6).

The proof of (7) follows from the proof of (6). We have that

$$b_k - \sum_{j=1}^{i-1} \mu_{k,j} b_j^* = b_k - \sum_{j=1}^{i-1} \lambda_{k,j} b_j$$

and since $d_j \lambda_{k,j} \in \mathbb{D}$ for $1 \leq j < k$, as detailed above, and (6), we can use (2) and get (7).

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

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

□

Proof of Lemma 2. To prove (11) we make use of (1) and (7) and get that

$$d_k \sum_{l=1}^{k} \mu_{j,l} \mu_{i,l} \beta_l = (b_i, d_k)(b_j - \sum_{l=1}^{k} \mu_{i,l} b_l^*) - d_k (b_i, b_j) \in \mathbb{D}$$

since all of the vectors are in $\mathbb{D}^n$.

To prove (12) we consider the basis

$$B' = \{b_1, \ldots, b_k, b_i, b_{k+2}, \ldots, b_{i-1}, b_{k+1}, b_{i+1}, \ldots, b_n\} \quad \text{for} \quad 1 \leq k < i \leq n$$

which we form from the basis B by swapping the vectors b_i and b_{k+1}. We note that $d'_k = d_k$, $\mu'_{k+1,j} = \mu_{i,j}$ and $\beta'_j = \beta_j$ for $1 \leq j \leq k$. We then have that

$$d_k \sum_{j=1}^{k} \mu_{i,j}^2 \beta_j = d_k \sum_{j=1}^{k} \left(\mu_{k+1,j}^2 \beta_j\right)' = d_k(\beta'_{k+1} - \left(b'_{k+1}, b'_{k+1}\right)) \in \mathbb{D},$$

since $d_k \beta'_{k+1} = d'_k \beta'_{k+1} \in \mathbb{D}$. □

Proof of The Algorithm. Let $\sigma_{i,j}^{(l)}$ denote the value of (11) or the variable σ at the l-th iteration of loop 1. 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

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

Let $\sigma_i^{(l)}$ denote the value of (12) or the variable σ at 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

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

Let $a_i^{(l)}$ denote the value of (7) or the variable a at the $l-1$ iteration of loop 3. We note that by (7) and (4) we assign the correct value to $\tilde{b}_2$ and $a_i^{(1)}$ is correct. If we assume that $a_i^{(l-1)}$ and $\tilde{b}_{l-1}$ are correct we have that

$$a_i^{(l)} = d_l(b_i - \sum_{k=1}^{l} \mu_{i,k}\tilde{b}_k) = d_l(\frac{a_i^{(l-1)}}{d_{l-1}} - \mu_{i,l}\tilde{b}_l) = \frac{d_l a - \tilde{\mu}_{i,l}\tilde{b}_l}{d_l}$$

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

B The C++ Algorithm

```
template <class IDE>
IDE dot( 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>                                                    10
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 sigma;
  Bt = Matrix<IDE>( m, n );
  Mt = Matrix<IDE>( n, n );

  d[0] = IDE(1);                                                        20
  for( int i = 0; i < n; i++ )
  {
    for( int j = 0; j <= i-1; j++ )
    {
      sigma = IDE(0);                                   // Loop 1
      for( int l = 0; l <= j-1; l++ )
        sigma = (d[l+1]*sigma + Mt[i][l]*Mt[j][l])/d[l];
      Mt[i][j] = d[j]*dot(B,i,B,j) - sigma;
    }
                                                                        30
    sigma = IDE(0);                                     // Loop 2
    for( int l = 0; l <= i-1; l++ )
      sigma = (d[l+1]*sigma + Mt[i][l]*Mt[i][l])/d[l];
    d[i+1] = d[i]*dot(B,i,B,i) - sigma;

    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++ )                                         40
      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

```
#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                                                              10
void gmp_add( void*& dest, const void* src1, const void* src2 );
void lip_add( void*& dest, const void* src1, const void* src2 );
#endif

#ifdef GMP
#define generic_add   gmp_add
#endif
#ifdef LIP
#define generic_add   lip_add
#endif                                                             20

#ifdef RUN_TIME_GENERIC
typedef pointer_to_ternary_void_function<void*&,const void*,const void*>
        Add;
#else
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 );                               30
  }
};
#endif

template<...,class Add,...>
class IDE
{
  protected:
    void* _data;
    static Add _add;                                               40
```

```
    ⋮
  public:
    typedef IDE<...,Add,...> ide_type;
    ⋮
    ide_type operator+( const ide_type& h ) const
    {
      ide_type t;
      _add( t._data, _data, h._data );
      return t;
    }                                                          50
    ⋮
};
```