Computational Aspects of Discrete Logarithms

by

Rob Lambert

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Electrical Engineering

Waterloo, Ontario, Canada, 1996

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Integer factorization and discrete logarithm calculation are important to public key cryptography. The most efficient known methods for these problems require the solution of large sparse linear systems, modulo two for the factoring case, and modulo large primes for the logarithm case. This thesis is concerned with solving these equations modulo large primes.

The methods typically used in this application are examined and compared, and improvements are suggested. A solution method derived from the bi-diagonalization method of Golub and Kahan is developed, and shown to require one-half the storage of the Lanczos method, one-quarter less than the conjugate gradient method, and no more computation than either of these methods. It is expected that this method will become the method of choice for the solution modulo large primes of the equations involved in discrete logarithm calculation.

The problem of breakdown for the general case of non-symmetric and possibly singular matrices is considered, and new lookahead methods for orthogonal and conjugate Lanczos algorithms are derived. A unified treatment of the Lanczos algorithms, the conjugate gradient algorithm and the Wiedemann algorithm is given using an orthogonal polynomial approach. It is shown, in particular, that incurable breakdowns can be handled by such an approach. The conjugate gradient algorithm is shown to consist of coupled conjugate and orthogonal Lanczos iterations, linking it to the development given for Lanczos methods. An efficient integrated lookahead method is developed for the conjugate gradient algorithm.

The initialization and termination conditions of each of these methods is examined in detail. Given a randomness assumption, the likelihood and expected degree of breakdown are developed.

In order to better understand the number of linear relations that might be obtained from the sieving portion of index calculus methods, the Dickman function, recently extended to the single large prime case, is extended here to the double large prime case. It is concluded that the number of relations obtained by combining large prime relations approaches a Poisson distribution. Since the variance of this distribution is identical to its mean, these estimates are intrinsically of limited value in predicting the number of relations that will be obtained.

Acknowledgements

I would like to acknowledge my supervisors Ian Blake and Scott Vanstone for their patience and support. Thanks are also due to my external examiner Arjen Lenstra, who was extremely helpful. I would like to thank Jeff Shallit for his inspirational course on algorithmic number theory. Thanks to Dariush Dabiri, Vahid Tarokh, Andrew Heunis, and Tomik Yaghoobian, for many discussions, and especially to Carolyn for her editing. To everyone again, thanks.

To Carolyn, Christine, Andrew, Catherine and Laura.

Contents

List of Figures

Chapter 1

Introduction

Recent interest in both integer factorization and in computing discrete logarithms can be attributed to the importance of these problems to public key cryptography. The Rivest, Shamir and Adleman (RSA) cryptosystem is no harder to break than is the problem of finding the factors of composite integers. Similarly, efficient algorithms for computing discrete logarithms would compromise the ElGamal cryptosystem and the Diffie-Hellman key exchange.

The development of methods to solve these problems has generated algorithms which are remarkably similar, the most recent development in both fields being the number field sieve. The number field sieve is an example of an index calculus method, the most successful family of methods for both factoring and discrete logarithms.

Index calculus techniques rely on linear system solution to calculate discrete logarithms. A set of primes is considered (the factor base), and multiplicative relations between the logarithms of these primes are gathered, until sufficient relations have been obtained to solve for the logarithms of the primes considered.

The major advances in index calculus techniques have come in the development of new methods of generating these multiplicative relations, and are reflected in the size of the numbers which the method requires to be smooth in the factor base, that is, only containing those primes as factors. Another advance has come in relaxing strict smoothness constraints and in allowing one or more large prime to appear. These large prime relations are to be combined so that the large primes are cancelled and a simple relation obtained (which is the basic function of the linear solution step, although it is possible that the cancellation of large primes is done prior to the remainder of the linear solution).

Index calculus techniques can generally be divided into two stages: the initial relation gathering stage, which is usually some form of sieving (similar to the sieve of

Eratosthenes, except keeping the composites), and the linear solution stage. (For some methods, individual logarithms are computed after this, in a third stage.) The relation gathering stage is inherently parallel; the sieving interval can be divided and searched independently. In addition, very little communication is required to coordinate this effort, since the relations are rare in the intervals.

In the second stage of the calculation, the linear system is solved yielding the logarithms of the elements in the factor base. Solving large systems of linear equations is not made parallel as easily as is relation gathering; in fact, it appears that this computation requires a single fast machine with large memory capacity, or else very closely coupled distributed processors.

In related methods for factoring, the size of this matrix can be as large as a million or more variables (primes), and as many relations. For the purpose of factoring, this matrix need only be solved modulo two. A key difference for index calculus algorithms for discrete logarithms is that this solution needs to be made over much larger primes, and therefore linear solution is much more difficult. The major focus of this thesis will be the solution of these very large systems of linear equations modulo large primes.

If the logarithms are modulo p, then the linear system of logarithms is defined modulo $p-1$. Assume that the order of the units, $p-1$, has been completely factored using a factoring technique of the same running time as the logarithm calculation to be performed. (Similarly for logarithms in other fields, assume that the order of the group has been completely factored.) Solving the logarithm modulo each prime power in the factorization of $p-1$, Pohlig and Hellman [28] observe that the solution modulo $p-1$ can be constructed via the Chinese remainder theorem.

In this way, the logarithm of each prime in the factor base is obtained, and logarithms of other elements are obtained by expressing them multiplicatively in the factor base. An alternate method of solution is taken in Schirokauer's application of the number field sieve to discrete logarithms modulo p [34]. Here relation gathering (sieving) goes on for the determination of the logarithm of each element β, and instead of solving for all of the logarithms in the factor base, the object of the linear solution is to find a relation of the form $a^{p_i^{e_i}} = \alpha^{x_i}\beta$ for each factor $p_i^{e_i}$ of $p-1$, where α is the base of logarithms. This is obtained by finding a vector in the null space of the vectors of relations, operating modulo $p_i^{e_i}$. The logarithm modulo $p-1$ is once again obtained with the Chinese remainder theorem. In this way, logarithm determination can be reduced to finding $p_i^{e_i}$ powers (of the given form). Using Schirokauer's reduction to power finding, it seems that linear solution modulo prime powers is required. Schirokauer proposes to solve e_i linear equations to lift the solution.

We suggest that using Bach's method of finding logarithms modulo prime powers, given the logarithm modulo the prime [1], it is possible to instead find logarithms modulo

p_i and lift them to logarithms modulo $p_i^{e_i}$, instead of having to lift the entire solution vector. This lifting is made via approximate p-adic logarithms. For the number field sieve, it can in either case be assumed that linear solution modulo a prime is sufficient for logarithm determination.

The linear system solution stage of a discrete logarithm algorithm can then expect to solve systems over primes almost as large as p. For factoring, the number field sieve algorithm has been employed to factor integers of up to 130 decimal digits. If this was a discrete logarithm calculation, then the linear system solver would have to deal with elements of 130 digits. If standard methods such as Gaussian elimination were employed, and the system to be solved was of size 1,000,000, then the memory required would be $\log_2(10) * 130 * (10^6)^2/8$ bytes, or roughly 50 Terabytes, if the matrix became completely filled in. Therefore, methods that use the sparsity of the linear system generated by sieving are required.

Several sparse solution methods, adapted from numerical analysis, have been proposed for solution modulo primes in the pioneering work of Odlyzko [24]. These are the well-known Lanczos and conjugate gradient algorithms. Over finite fields, these algorithms exhibit breakdown problems not experienced over the real field. These breakdowns halt the progress of the algorithm before a solution to the linear system can be obtained. When this occurs, it is necessary to restart the algorithm with different parameters [19]. The method of linear solution given by Wiedemann [36] does not experience these difficulties, but requires significantly more computation.

In chapter 2, the usual methods for linear system solution are examined for the large prime case, their performance for solution modulo large primes is evaluated, and a comparison is made. The orthogonal Lanczos algorithm, which is typical in numerical analysis, is adapted to solution over large primes.

Improvements of the Lanczos methods are given, reducing the memory required by the conjugate Lanczos algorithm (as has been presented for discrete logarithm application) by 16%. Since the memory requirements of these methods are large, reduction in memory usage is very important.

The most important result of chapter 2 is the development of a method employing bidiagonalization that requires 50% less memory than the conjugate Lanczos algorithm as previously presented, and requiring no more computation. It is expected that this method will become the method of choice for solution modulo large primes.

In chapter 3 the problem of breakdown (the emergence of a zero denominator before the algorithm has completed) is examined in detail, for the general case of nonsymmetric, possibly singular matrices. New lookahead methods are derived for Lanczos methods, and are shown to be very practical in that only constant additional memory is required to handle lookahead of any degree, and that is required only during lookahead. The conjugate gradient method is shown to consist of two coupled Lanczos algorithms. An integrated lookahead method for both of these Lanczos processes is derived. The other major point of chapter 3 is the underlying unity of the Lanczos (both varieties), the conjugate gradient and the Wiedemann algorithms. Using an orthogonal polynomial interpretation, it is shown that all these methods are operating on the same underlying system. From this it is shown that the occurrence of incurable breakdown (to be defined there) in the Lanczos and conjugate gradient algorithms can be addressed by the Wiedemann algorithm.

This implies that, at least for the application to linear solution, incurable breakdowns are not really incurable. It is shown in chapter 3 that the Lanczos or conjugate gradient method and the Wiedemann algorithm can be integrated into a practical hybrid, if polynomial recurrences are also maintained in addition to the standard vector recurrences. It is not necessary that these polynomials be computed as the algorithms progress, but merely that a few scalar parameters be stored at each iteration, only to be recalled to circumvent incurable breakdown if that occurs.

Given a randomness assumption, the likelihood of breakdown and the probability of the degree of lookahead required are derived. Practical vector versions of the lookahead methods are derived after the initial polynomial versions, and the question of initial vectors is considered in detail.

Chapter 4 is concerned with the first part of the index calculus algorithm, that is, the gathering of relations. The number of relations obtained is related asymptotically to the fraction of smooth numbers. This asymptotic ratio can be estimated with the Dickman rho function. When the large prime variation is used, extensions of the Dickman rho function are required. Recently, Bach and Peralta have extended the Dickman rho function to the case of a single large prime, as is employed in the large prime variation, and have given an effective computation method for this new quantity. Following their example, an extension of the Dickman rho function to two large primes, as is used in the double large prime variation, is derived here, and an effective computation method is described.

These extensions are the first step in estimating the number of smooth relations that can be anticipated when combining relations with large prime factors (and cancelling them). The remainder of chapter 4 concerns the number of relations that can be expected to result from this combining of large prime relations. Using methods of cell occupancy and Poisson approximation, it is shown that, given an assumption on the random generation of the large prime relations, the number of useful relations derived from combining tends to a Poisson random variable. This is disappointing, since it implies that the variance of the estimate will be large. A result on random graphs implies that this is also the case for the double large prime variation. Hence, it is unlikely that these estimates can be used effectively in the design of such algorithms.

To summarize, the next chapter presents the linear system solution methods used in discrete logarithm determination, and gives comparisons, improvements and a new algorithm requiring less memory. Chapter 3 presents a unified theory for the three major methods, Lanczos, conjugate gradient, and Wiedemann, derives efficient lookahead methods for the Lanczos and conjugate gradient methods, and shows how the methods can be combined into hybrid versions retaining the advantages of each.

Chapter 4 analyses the number of relations that can be expected from the large prime variations, and derives an extension of the Dickman rho function to include two large primes.

Chapter 2

Application of Linear Solution to Discrete Logarithm Calculation

The purpose of this chapter is to consider the linear system solution methods that have been proposed for discrete logarithm calculation, in order to compare the methods and, if possible, to improve them for this application. The sufficiency of linear solution modulo primes for the linear solution stage of a discrete logarithm algorithm was shown in the last chapter, assuming that the order of the group has been factored. The methods proposed for linear solution are then briefly listed, and a comparison of their speed and memory requirements is made. Derivations of these methods are given, and several improvements are suggested, including the orthogonal Lanczos procedure, not previously used in discrete logarithm work. The chapter concludes with an adaptation of the bidiagonalization technique of Golub and Kahan [9], which produces a linear solution method with smaller memory usage than all the other methods, including the Wiedemann, but not having to repeat the Krylov subspace calculation, as the Wiedemann method does. It is expected that this method will become the method of choice for the linear system solution stage of discrete logarithm calculation.

Let E be the matrix of exponents (relations among logarithms) of the factor base, with each row specifying a relation. When determining the logarithm of each element in the factor base, the linear solution to be solved is $Ez = b$, for some b also specified when the relations were gathered. Since more relations than primes are gathered, E has more rows than columns, and z can be determined if the randomly-gathered relation vectors have rank equal to the order of the factor base.

In contrast, when finding a p-th power, a combination of relations is required that is 0 modulo p; that is, a vector in the null space of E^T is required. The linear system problem is then to find $E^T z = 0$.

2.1 Linear Solution Techniques

When solving the extremely large systems arising in discrete logarithm calculation, time and especially memory must be of concern. The required time is also heavily dependent on the amount of memory required, since the time spent reading memory into the processor will be a large factor of the time required for solution, and also since extremely large memory requirements may require the use of slow secondary storage. The nature of the linear system solution is quite different from that encountered in numerical analysis. The major difference is that exact arithmetic must be used. This means that the stability of the methods used need not be the concern that it is in numerical work, and therefore exact but unstable (for numerical work) techniques are fully applicable. In contrast, interactive methods that do not converge in a finite number of iterations (such as Gauss-Seidel, which converges asymptotically to the solution, when it does converge) are apparently not more valuable than random guessing.

Since relations are obtained from exponents of factorizations, the relations themselves are both very sparse and comprised of small components on average. For extremely large systems over large primes, it is very important that the solution technique take advantage of the sparsity of the system. A survey of the possible methods of solution is given next.

2.1.1 Gaussian Elimination

A standard method for linear system solution is Gaussian elimination. The complexity of this method is $O(n^3 b^2)$ bit operations, where n is the size of the matrix, and b is the number of bits in the binary representation of the prime. Even if the linear system was initially sparse, application of the standard Gaussian elimination causes the initially zero portions of the matrix to become nonzero and possibly large. If a matrix of dimension 1,000,000 over a 400-bit prime is fully filled in, it can be expected to require more than $5 * 10^{13}$ bytes of storage, or around 50 Terabytes (or half that much if the system is made lower triangular), which would easily outstrip the main memory resources of the largest computers. Standard Gaussian elimination is therefore a poor choice for these problems. More intelligent application of Gaussian elimination is possible, however.

2.1.2 Structured Gaussian Elimination

Structured Gaussian elimination is a method that tries to preserve as much sparsity as possible in the matrix as the elementary reductions are made. The method is derives from the sparse matrix reduction of Pomerance and Smith [31] and in the linear system solving work of Odlyzko and LaMacchia [19]. It is unclear how much initial structured Gaussian elimination should be performed before handing the system over to a sparse solver, but certainly 'singletons', or relations involving only one prime, should be removed beforehand. It is very possible that, if carried too far, this technique might actually make the resulting matrix dense enough to slow the solution step down. The answer to this question would likely involve an empirical study, and is not attempted here.

2.1.3 Wiedemann's Method

Krylov subspace methods solve linear systems $Az = b$ by building a Krylov subspace $\{b, Ab, A^2b, \cdots\}$, determining inner products, and also evaluating some scalar quantities. They therefore have the very appealing advantage of leaving the original sparsity of the matrix A completely intact. Since the matrix itself is left intact, the focus of memory usage shifts from the matrix to the intermediate vectors used in the method. For the large linear systems derived from sieving, the storage for the sparse matrix itself is smaller than the storage required for a single vector in the Krylov subspace, so a rough accounting of storage requirements can be made by determining the number of vectors required by an algorithm.

All sequences of the form $b, Ab, A^2b, \cdots$ are eventually recurrent, which is evident from the Cayley-Hamilton formula, which says that a matrix satisfies its characteristic polynomial. Let the minimal polynomial for this sequence be $f(x)$. Wiedemann's method [36] for the solution of $Az = b$ over finite fields starts with computing the scalar sequence $u^Tb, u^TAb, u^TA^2b, \cdots$, where u is a random vector.

This vector is fed into the Berlekamp-Massey algorithm [22], which determines the minimal polynomial of the sequence. Since an understanding of the Berlekamp-Massey algorithm is also central to the connection drawn in the next chapter between the Lanczos and Wiedemann algorithms, a partial description is provided here.

The motivation Massey gave to Berlekamp's original algorithm was to show how it designed the shortest linear feedback shift register that would reproduce a certain sequence. For the purposes of the next chapter, the development here is slightly different from the usual one [22]. If the sequence to be matched is $\mu_0, \mu_1, \cdots, \mu_{2i-1}$, we want to find a polynomial $\phi_i(x) = \phi_{i,0} + \phi_{i,1}x^1 + \cdots + \phi_{i,i}x^i$ annihilating this sequence, that is,

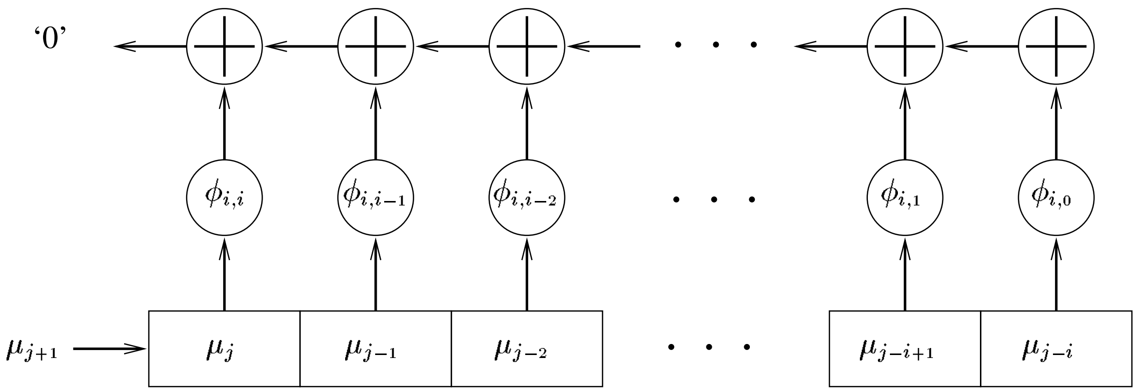

Figure 2.1: Feed-forward Filter

we want

$$\begin{bmatrix} \mu_0 & \mu_1 & \cdots & \mu_i \\ \mu_1 & \mu_2 & \cdots & \mu_{i+1} \\ \vdots & \vdots & \ddots & \vdots \\ \mu_{i-1} & \mu_i & \cdots & \mu_{2i-1} \end{bmatrix} \begin{bmatrix} \phi_{i,0} \\ \phi_{i,1} \\ \vdots \\ \phi_{i,i} \end{bmatrix} = 0.$$

If the sequence is fed through a shift register, then the coefficients $\phi_{i,j}$ form the taps (scaled outputs of register elements) of a feed-forward filter, which is to have output always zero (see figure 2.1). Any solution ϕ_i will remain a solution when scaled (with nonzero scale); therefore, if $\phi_{i,i}$ is nonzero, then ϕ_i can be made monic. This means that the input coming from the adders into the last adder must be $-\mu_j$; therefore, if the other taps are negated, they will produce μ_j. Then we can remove the first stage of the shift register, feeding back the output of the adders into the remaining registers, and dispense with having to feed the sequence into the shift register (although it still must be initialized with the first terms of the sequence). If the labelling of the taps is reversed, then the usual linear feedback shift register considered by Massey is obtained (see figure 2.2, however the labels are not reversed).

This linear feedback shift register is extended recursively in the Berlekamp-Massey algorithm, as more and more terms of the sequence $\mu_0, \mu_1, \cdots$ are matched with polynomials minimal in degree. The difference between the j-th term and the j-th output of the linear feedback shift register that matches $j-1$ terms is called the j-th discrepancy, and a discrepancy of 0 means that the present linear feedback shift register does not need to be revised at that step. The previous solution can be used to develop the solution for one more term (see [22]), but the previous polynomial having discrepancy nonzero is also

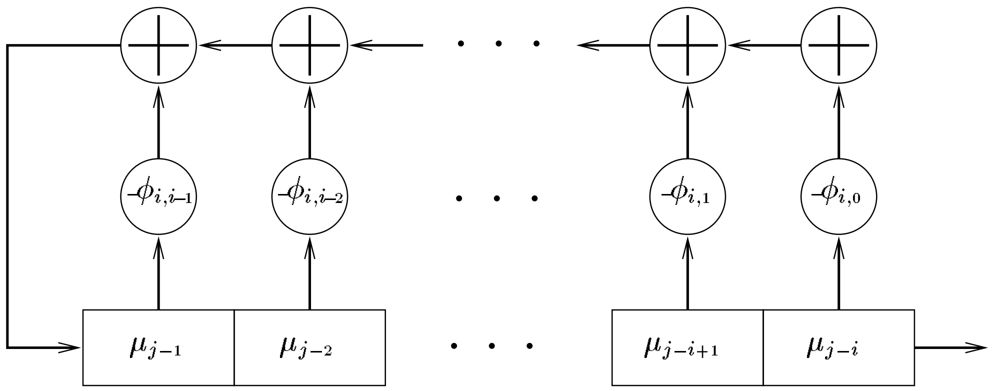

Figure 2.2: Feedback Filter

required.

The Berlekamp-Massey algorithm provides the minimal polynomial for the sequence, but requires $2n$ terms to find a minimal polynomial of degree n (see [36] and references there). The Berlekamp-Massey algorithm then requires essentially two vectors of length n, although these vectors are not fully occupied until the end of the algorithm.

When applied to the sequence $u^T b, u^T Ab, u^T A^2 b, \cdots$, the polynomial resulting from the Berlekamp-Massey algorithm must divide the minimal polynomial of the sequence $b, Ab, A^2b, \cdots$ (and also the minimal polynomial of A, which divides the characteristic polynomial of A), and so by computing the minimal polynomials of sequences with various u's, some information on the factors of the minimal polynomial of the sequence $b, Ab, A^2b, \cdots$ is discovered. Assuming random u, it is quite likely (Wiedemann shows at least 70% after three trials) that after a certain number of factors $f_i(x)$ of the minimal polynomial $f(x)$ will be found, $f(x)$ itself can be reconstructed as the least common multiple of the $f_i(x)$'s [36].

In his Algorithm 1, Wiedemann proposes an efficient procedure for accumulating the $f(x)$. After finding the minimal polynomial, $f_0(x)$, of the sequence $\{u_0 A^i b_0\}_{i=0}^{\infty}$ (where $b_0 = b$), if $f_0(A)b_0$ is not zero, then define $b_1 = f_0(A)b_0$, and selecting another vector u_1, compute the minimal polynomial of the sequence $\{u_1 A^i b_1\}_{i=0}^{\infty}$. Repeating this, factors of the minimal polynomial of the sequence $b, Ab, A^2b, \cdots$ are obtained and $f(x) = f_k(x) \cdots f_1(x) f_0(x)$.

Then the vector z can be re-constructed from the minimal polynomial (or a null vector of A found), as follows. If the constant term of this minimal polynomial is zero, then factoring out an A, a null vector of A is produced. This null vector is not trivial by the minimality of $f(x)$. Assume then that the constant term is nonzero. Let the trailing ($f(0)$) term be normalized to 1. Then the minimal polynomial in matrix form can be rearranged $I = A\left[-\sum_{i=1}^{n} f_i A^{i-1}\right]$, from which $-\sum_{i=1}^{n} f_i A^{i-1}$ is A^{-1}. Multiplying by b gives $z = -\sum_{i=1}^{n} f_i A^{i-1} b$.

At best, Wiedemann's method will complete in a single pass, requiring $2n$ multiplications by A, where n is the dimension of A, before the Berlekamp-Massey algorithm can determine the factor of the minimum polynomial. The time required by the Berlekamp-Massey algorithm is much less than the time required to form the Krylov subspace. The storage required by the vectors $b, Ab, A^2b, \cdots$ precludes storing them, and therefore it is necessary to recompute them in order to construct the solution as given by the sum above. If the minimum polynomial was found on the first occasion, this means $3n$ multiplications by A are required.

The storage required by the Wiedemann method is small: two vectors (previous and current) to calculate the new Krylov vector, and two vectors to hold the minimum polynomial and the previous minimal polynomial in the Berlekamp-Massey algorithm (besides the storage for the matrix itself and scalar quantities). Therefore, four vectors are needed in the initial pass of the Wiedemann algorithm. A vector for the solution is needed during the second pass after the minimal polynomial has been determined, but the minimal polynomial may be put on secondary storage, since it is only needed one element at a time; the previous minimal polynomial is not required to compose the solution.

At a maximum then, the Wiedemann algorithm needs four vectors in storage.

2.1.4 Exact Iterative Methods for Symmetric Systems

Odlyzko and Coppersmith first applied these iterative methods to linear solution in finite fields, following the suggestion of Karmarkar [24]. Both the conjugate gradient and the Lanczos methods have found application in the solution of sparse symmetric linear systems over the reals, and in the determination of the extremal eigenvalues of these matrices. These algorithms were proposed for such purposes by pioneering numerical analysts (the conjugate gradient method by Hestenes and Stiefel [13], and the Lanczos process by Lanczos [20]). These methods were prone to loss of orthogonality, and were often rejected in favor of more numerically-stable methods, but they since have been revived with the use of pre-conditioning techniques.

With exact arithmetic, these algorithms have the excellent property of terminating at or before the theoretical iteration limit, though in numerical analysis they are typically used as iterative methods, halted long before this limit.

The first difficulty that must be overcome when applying either of these methods is that the matrix B is not typically either symmetric or square (usually B is over-determined, but in a consistent way). One possible solution is to employ generalizations of these methods to the non-symmetric case [10], and work in this direction will be presented in the next chapter.

Another possibility, explored by Odlyzko, for solving the non-symmetric system $Bz = b$, is to form the symmetric matrix $A = B^T B$, and to solve the normal equation $B^T Bz = B^T b$. This is not an attractive alternative in numerical analysis because of the squaring of the condition number. For solution in exact arithmetic, such concerns of stability do not occur. Randomization of the normal equations with a diagonal matrix D has also been proposed as a means of side-stepping certain difficulties that occur in solution modulo p. When solving the normal equation, the solution obtained will actually be a solution $Bz = b + t$, where t is a vector in the null space of B^T.

When using linear system solution to find the logarithms of a set of elements, then typically the number of elements (usually primes), or columns, is less than the number of relations, or rows. The system is not over-determined because the relations will all be consistent (unless there has been an error in the relation generation). With this system, B^T has more columns than rows, and therefore has a non-trivial nullspace. This suggests that when applying these methods to solving for logarithms, the number of relations should be restricted to the number of primes. If B is singular, relations can be replaced, and the linear system solution repeated.

In applying these linear solution methods to finding a vector in the null space, as they might be used in the number field sieve discrete logarithm calculation, the matrix B will have more columns (relations) than rows (primes). If B has full row rank, then the solution of the normal equation form ($A = B^T B$) will be a solution to the original linear system. As the row rank of B decreases, this becomes less likely, so in practice, more relations can be added to B in an attempt to increase the row rank.

The Lanczos and conjugate gradient algorithms, as they have appeared in the discrete logarithm literature, are presented in the next subsections.

2.1.5 Lanczos's Method

Setting

$$\begin{aligned} w_0 &= y, \\ v_1 &= Aw_0, \\ w_1 &= v_1 - \frac{(v_1, v_1)}{(w_0, v_1)} w_0, \end{aligned}$$

the Lanczos algorithm, as used by discrete logarithm and factoring workers, iterates [19]:

$$\begin{aligned} v_{i+1} &= Aw_i, \\ w_{i+1} &= v_{i+1} - \frac{(v_{i+1}, v_{i+1})}{(w_i, v_{i+1})} w_i - \frac{(v_{i+1}, v_1)}{(w_{i-1}, v_i)} w_{i-1}. \end{aligned}$$

The iteration stops when w_j is found to be self-conjugate; that is, when $(w_j, Aw_j) = 0$, which appears in the denominator of the Lanczos algorithm. If $w_j = 0$, then a solution is $z = \sum_{i=0}^{j-1} w_i(w_i, y)/(w_i, v_{i+1})$, which is accumulated as the algorithm progresses. Over the reals, if A is positive definite, then $(w_j, Aw_j) = 0$ implies that $w_j = 0$, but this is not the case modulo p, where (w_j, Aw_j) can be zero without having $w_j = 0$. If $w_j \neq 0$, then the method has failed. When applying the Lanczos algorithm to solution modulo 2, Montgomery [23] proposed keeping subspaces instead of vectors in the methods, deriving a block version of the Lanczos algorithm. This has the dual advantage of greatly reducing the risk of breakdown (zero denominators in the iteration above, which is very likely for $p = 2$) and of speeding up the algorithm by making use of the fact that most processors can operate on a block of elements modulo 2. The blocking factor is then typically 32 or 64.

When p is large, breakdown is much less likely, and since other methods of logarithm determination can be used for p small, the breakdown problem can be dealt with simply by restarting the algorithm with a different system (which is usually possible to do by deleting or adding relations). Breakdown probability is considered in the next chapter. For determining a vector in the null space, the right-hand side is usually the random quantity Ar for r a random vector, and so the algorithm can simply be restarted with different r.

The Lanczos algorithm uses seven vector variables, $w_{i-1}, w_i, w_{i+1}, v_i, v_{i+1}, z, y$. If the dependencies between these values are expressed as a directed graph (a dependency graph), then several ways to reduce the number of vectors to six are apparent; for example, w_{i+1} and v_i can share the same storage, but no further reduction is possible by this means. Each iteration of the Lanczos algorithm requires one matrix-vector product (which in the case of $A = B^T B$ is two matrix-vector products of the type used in Wiedemann's method), and in addition three inner product calculations (if (w_i, v_{i+1}) is preserved for the next iteration), plus scalar operations. Since the Lanczos method will terminate in at most n iterations where n is the dimension of A, this will be at most $2n$ matrix-vector products (of the matrix B or B^T) and $3n$ inner products. Compared with the Wiedemann method, the Lanczos method requires n less matrix-vector products, $3n$ inner products where the Wiedemann algorithm required only one, and roughly 50% more storage. Since matrix-vector products will likely dominate the computation, if the storage is available, then the conclusion is that Lanczos is to be preferred over Wiedemann's method for solution modulo large primes.

2.1.6 The Conjugate Gradient Method

The proof of the conjugate gradient method given by Hestenes and Stiefel [13] is algebraic, although the motivation is the minimization of a quadratic function. The proof is not repeated here (although a compact proof of the more general bi-conjugate gradient method is included in the next chapter). Over finite fields it is possible that the algorithm may break down prematurely (that is, it may have a zero denominator in the expression for a_i or b_i, with nonzero residual r_i). The breakdown of the conjugate gradient algorithm will be studied in the next chapter.

The conjugate gradient algorithm finds the solution to the equation $Az = b$ by updating z along conjugate direction vectors p_i, or gradients. The residuals r_i are orthogonal [13].

Choosing an arbitrary vector z_0 and setting $p_0 = r_0 = y - Az_0$, the conjugate gradient iteration is:

$$\begin{aligned}
s_i &= Ap_i; \\
a_i &= \frac{(r_i, r_i)}{(p_i, s_i)}, \\
z_{i+1} &= z_i + a_i p_i, \\
r_{i+1} &= r_i - a_i s_i, \\
b_i &= \frac{(r_{i+1}, r_{i+1})}{(r_i, r_i)}, \\
p_{i+1} &= r_{i+1} + b_i p_i.
\end{aligned}$$

Here the r_i's are the residual vectors, the p_i's are the conjugate direction vectors, the s_i's are the matrix-scaled direction vectors, and the z_i's are the solution estimate vectors. It is shown in the original exposition [13] that the solution will be found at some iteration number less than n, the dimension of A. Termination occurs when $r_{i+1} = 0$, but as with Lanczos, breakdowns can occur if $(r_i, r_i) = 0$ without $r_i = 0$.

The block version of Lanczos given by Montgomery can often avoid the breakdown problem, but a formulation of such a technique for the conjugate gradient method has not been proposed. The development in the next chapter may be useful in this regard, since it shows that the conjugate gradient is built from two coupled Lanczos recursions, and therefore Montgomery's blocking may find application there as well.

For large primes, it it not clear whether there is any advantage in a block method, since it requires more memory and computation, and since the risk of breakdown is extremely small for large primes, but for small primes a block conjugate gradient algorithm would be welcome.

Examining the algorithm, it is apparent that the conjugate gradient method needs only to keep the vector variables $s_i, z_{i+1}, r_{i+1}, p_{i+1}$ in memory, in addition to scalar quantities. This is a reduction from six to four when compared with the Lanczos method, and in addition only two inner products are required.

2.1.7 Comparison of the Methods, for Discrete Logarithm Application

If the memory is available, it was concluded above that the Lanczos method is to be preferred to Wiedemann's method for linear system solution over large prime moduli. The conjugate gradient method is in its basic building blocks (matrix-vector products and inner products) similar to the Lanczos method, but since it requires one-third less vector memory than Lanczos and half the number of inner products, it is to be preferred for linear system solution for this problem domain, needing less memory and being faster.

For comparison, the memory requirement for a 1,000,000-relation 400-bit system when employing Lanczos, is 286 megabytes of memory storage, considering only the memory required to store the vectors (the dominant storage requirement). For the conjugate gradient method this is reduced to 190 megabytes. The superiority of conjugate gradient over Lanczos for large prime moduli does not seem to have been noticed before. The largest linear system solved modulo a large prime was solved with the Lanczos algorithm [19] by Odlyzko and LaMacchia in their work showing the insecurity of a Sun crypto-system. They solved a system of 288,017 equations in 96,321 unknowns modulo a prime of 191 bits, on a workstation. In a related problem (logarithms in GF(2^n)), Gordon and McCurley have solved a system of 117,164 equations in 58,636 unknowns modulo $2^{401} - 1$. This system was solved on a large parallel machine. The largest prime in $2^{401} - 1$ is 182 bits. This solution was made with the conjugate gradient method. The special number field sieve factorization of a 162-digit number used 833,017 relations, and for a general number field sieve factorization of a 105-digit number, 1,294,861 relations were used [23]. Since the logarithm method is essentially similar, it is expected that matrices of about 1,000,000 relations (and slightly fewer primes) will be required for logarithm determination for primes of 400 bits, at least when using the special number field sieve. These numbers will be used as a rough guide when evaluating linear solution methods.

Combinations of methods are possible, and structured Gaussian elimination before Lanczos or conjugate gradient has been advocated [19]. It has also been suggested that using structured Gaussian elimination before other methods may not be wise, since it greatly reduces the sparsity of the system.

For solution modulo 2, most processors are able to compute with a block of modulo 2 components at once, using the exclusive or instruction. This makes more effective use of the arithmetic unit, and makes the algorithms run faster. As mentioned previously, Montgomery's block Lanczos also has the advantage of being able to avoid breakdowns. For large primes, the arithmetic unit is unlikely to be able to deal even with the whole element at once, and so there appears to be no speed advantage with blocking. In addition, the problem of breakdown is much less severe (the probability is considered in the next chapter), and the algorithms can easily be restarted at a new point, hopefully avoiding breakdown. Besides this, the storage cost of keeping subspaces instead of vectors is much more for the large prime modulus case than in the modulo 2 case. For these reasons, the use of block iterative methods for large primes has no obvious advantage.

2.2 Lanczos Methods in Prime Fields

In a previous section, the Lanczos and conjugate gradient algorithms were given without any derivation. The derivation of the Lanczos methods is given next, and improvements for the case of solution modulo large primes will be uncovered.

First, the Lanczos method as used by workers in discrete logarithms and factoring, will be derived. It will be shown to be a matrix-orthogonal (or conjugate) method, instead of the orthogonal Lanczos method used by numerical analysts. The derivation of the orthogonal Lanczos method will then be presented. Since the matrix B is not expected to be symmetric in discrete logarithm application (or factoring), these methods work with the normalized matrix $A = B^T B$ which, as described above, finds the solution to $Az = b$ only modulo the null space of B^T. A generalization of the Lanczos method (given in the original exposition [20]) works directly on the non-symmetric matrix B. The non-symmetric Lanczos algorithm is considered in the next chapter.

Following this, an adaptation of the bi-diagonalization method of Golub and Kahan [9], adapted to linear solution by Paige [26], is shown to be the basis of a new method for linear solution modulo large primes. This method, which again works with the normalized matrix $A = B^T B$, is shown to be more attractive than any such method previously known.

2.2.1 The Conjugate Lanczos Method

The use of the Lanczos method for linear system solution modulo large primes was first proposed by Odlyzko [24]. For symmetric A, the method seeks a matrix W such that

$W^TAW = D$, where D is a diagonal matrix. Such a matrix effectively gives the solution (if D is full rank), since if $Az = b$, and defining $z = Wy$ for some intermediate y, then $AWy = b$. Scaling by W^T gives $W^TAWy = W^Tb$, or $Dy = W^Tb$, which is easily solved for y, and thus z is determined.

In the Lanczos method, the columns of W are determined one by one, and hence the elements of y can be determined one by one as well. Therefore, $z = Wy$ can be accumulated as the columns of W are determined, and so only the current column of W need be retained, so far as the updating of the solution is concerned. The derivation of this Lanczos method is expanded from Montgomery [23].

Let the columns of W be $W = [w_1, w_2, \cdots,]$, and set $w_1 = b$; then the A-orthogonal, or conjugate, version of the Gram-Schmidt orthogonalization procedure, applied to the Krylov subspace $\{Aw_1, Aw_2, \cdots\}$, iterates

$$w_i = Aw_{i-1} - \sum_{j=1}^{i-1} c_{i,j} w_j,$$

where

$$c_{i,j} = \frac{(Aw_j)^T(Aw_{i-1})}{w_j^T Aw_j},$$

until w_i is the zero vector. Although this method in the literature [23, 35, 24] is always started with the initial vector $w_1 = b$, it is seen from the above discussion that this is not required, and so it is possible to restart the method merely by choosing a new initial vector instead of b. Choice of initial vectors is discussed in the next chapter.

If W is the partially-computed matrix given by columns $W = [w_1, w_2, \cdots, w_k]$, then $W^TAW = \mathrm{diag}[\delta_1, \delta_2, \cdots, \delta_k]$. This is true for $k = 1$, and for induction let $W_+ = [w_1, w_2, \cdots, w_k, w_{k+1}]$. Since $(w_r^T Aw_c)^T = w_c^T Aw_r$, assume that $r < c = k + 1$. Then $w_r^T Aw_{k+1} = w_r^T A(Aw_k - \sum_{j=1}^{k} c_{k+1,j} w_j) = w_r^T A(Aw_k - c_{k+1,r} w_r) = w_r^T A^2 w_k - (w_r^T A^2 w_k)(w_r^T Aw_r)/(w_r^T Aw_r) = 0$. Therefore, $W_+^T AW_+$ is diagonal when the algorithm has completed (that is, when no zero denominators have been encountered).

This Lanczos procedure is efficient because the $c_{i,j}$ vanish for $j < i - 1$ since

$$(Aw_j)^T(Aw_{i-1}) = (w_{j+1} + \sum_{k=1}^{j} c_{j+1,k} w_k)^T (Aw_{i-1}),$$

which, by the fact just established, is zero for $j + 1 < i - 1$, or $j < i - 2$. Therefore, $(AW)^T(AW)$ is symmetric tridiagonal.

The iteration thus simplifies to

$$w_i = Aw_{i-1} - c_{i,i-1} w_{i-1} - c_{i,i-2} w_{i-2},$$

for $i > 2$. The rank of $W^T AW$ limits the Lanczos algorithm so that it must terminate (successfully or unsuccessfully) in no more than $\mathtt{rank}(A)$ iterations.

The computations of

$$c_{i,i-1} = \frac{(Aw_{i-1})^T(Aw_{i-1})}{w_{i-1}^T(Aw_{i-1})},$$

which is abbreviated as α, and

$$c_{i,i-2} = \frac{(Aw_{i-2})^T(Aw_{i-1})}{w_{i-2}^T(Aw_{i-2})},$$

which is abbreviated as β, are needed to compute w_i for an iteration. Then w_{i-1}, w_{i-2}, Aw_{i-1} and Aw_{i-2}, as well as the scalars $w_{i-1}^T Aw_{i-1}$, and $w_{i-2}^T Aw_{i-2}$, need to be available at each iteration, in addition to b and the partially accumulated solution vector z. The vector w_i can share storage with w_{i-2}, so this method requires six vectors, which as previously mentioned, are the dominant factor in the memory requirements of a method over large prime fields. In algorithm form:

◯ ALGORITHM 2.2.1 *(Symmetric) Conjugate Lanczos*

$\langle w_{i-2}\rangle$:= *random vector*

$\langle Aw_{i-2}\rangle := Aw_{i-2}$

$\langle w_{i-2}^T Aw_{i-2}\rangle := w_{i-2}^T\langle w_{i-2}^T Aw_{i-2}\rangle$

$z := \frac{w_{i-2}^T b}{\langle w_{i-2}^T Aw_{i-2}\rangle} w_{i-2}$

$\alpha := (\langle Aw_{i-2}\rangle)^T(\langle Aw_{i-2}\rangle)/\langle w_{i-2}^T Aw_{i-2}\rangle$

$w_{i-1} := \langle Aw_{i-2}\rangle - \alpha\langle w_{i-2}\rangle$

$\langle Aw_{i-1}\rangle := Aw_{i-1}$

$\langle w_{i-1}^T Aw_{i-1}\rangle := w_{i-1}^T\langle Aw_{i-1}\rangle$

$z := z + \frac{w_{i-1}^T b}{\langle w_{i-1}^T Aw_{i-1}\rangle} w_{i-1}$

while $\langle w_{i-1}^T Aw_{i-1}^T\rangle \neq 0$ *do* {

  $\beta := \frac{(\langle Aw_{i-2}\rangle)^T(\langle Aw_{i-1}\rangle)}{\langle w_{i-2}^T Aw_{i-2}\rangle}$

  $\alpha := \frac{(\langle Aw_{i-1}\rangle)^T(\langle Aw_{i-1}\rangle)}{\langle w_{i-1}^T Aw_{i-1}\rangle}$

  $w_i := -\beta w_{i-2}$ *(in the same storage as* w_{i-2}*)*

  $w_i := w_i - \alpha w_{i-1}$

$$w_i \ := \ w_i + \langle Aw_{i-1}\rangle$$

$$w_{i-2} \ := \ w_{i-1}$$
$$w_{i-1} \ := \ w_i$$
$$\langle Aw_{i-2}\rangle \ := \ \langle Aw_{i-1}\rangle$$
$$\langle w_{i-2}^T Aw_{i-2}\rangle \ := \ \langle w_{i-1}^T Aw_{i-1}\rangle$$

$$\langle Aw_{i-1}\rangle \ := \ Aw_{i-1}$$
$$\langle w_{i-1}^T Aw_{i-1}\rangle \ := \ w_{i-1}^T\langle Aw_{i-1}\rangle$$
$$z \ := \ z + \frac{w_{i-1}^T b}{\langle w_{i-1}^T Aw_{i-1}\rangle} w_{i-1}$$

}

○

The $\langle\rangle$ encase storage variables used by the algorithm, to distinguish them from operations. By sharing storage, the number of vectors required can only be reduced to six. There is, however, an algebraic change to the algorithm that allows it to use only five (this may also apply to Montgomery's block method modulo two [23]).

The idea is to make fuller use of the orthogonality condition in order to reduce storage requirements. For this algorithm, notice that orthogonality permits the simplification:

$$\beta = \frac{(Aw_{i-2})^T(Aw_{i-1})}{w_{i-2}^T(Aw_{i-2})} = \frac{(w_{i-1})^T(Aw_{i-1})}{w_{i-2}^T(Aw_{i-2})} = \frac{\delta_{i-1}}{\delta_{i-2}},$$

where $\delta_i = w_i^T Aw_i$. Since exact arithmetic is used, this orthogonality can be taken advantage of in places where one might hesitate to do so in finite-precision real arithmetic. Inserting this into the algorithm:

○ ALGORITHM 2.2.2 *(Symmetric) Conjugate Lanczos, modified*

$$j \ := \ 1$$
w_{j-1} := *zero vector*
w_j := *random vector*

$$
\begin{aligned}
&\langle Av_j \rangle \ := \ Aw_j \\
&s_j \ := \ v_j \\
&\delta_j \ := \ w_j^T \langle Av_j \rangle \\
&\textit{if } \delta_j = 0 \textit{ then RESTART} \\
&\alpha \ := \ (\langle Av_j \rangle)^T (\langle Av_j \rangle)/\delta_j \\
&\beta \ := \ 0 \\
&z \ := \ (w_j^T b/\delta_j) w_j \\
&\textit{while } \delta_j \neq 0 \textit{ do } \{ \\
&\qquad w_{j+1} \ := \ \langle Av_j \rangle - \alpha w_j - \beta w_{j-1} \\
&\qquad j \ := \ j+1 \\
&\qquad \delta_j \ := \ w_j^T \langle Av_j \rangle \\
&\qquad \textit{if } \delta_j \neq 0 \textit{ then } \{ \\
&\qquad\qquad \alpha \ := \ (\langle Av_j \rangle)^T (\langle Av_j \rangle)/\delta_j \\
&\qquad\qquad \beta \ := \ \delta_j/\delta_{j-1} \\
&\qquad\qquad z \ := \ z + (w_j^T b/\delta_j) w_j \\
&\qquad \} \\
&\}
\end{aligned}
$$

○

Here notice that w_{j+1} and w_{j-1} can share storage, so only w_j, w_{j-1}, Aw_j, z and b need be retained as the algorithm progresses. The same number of matrix multiplications and one less inner product are needed, as compared to the previous version.

2.2.2 Orthogonal Lanczos Methods

Now consider the orthogonal Lanczos method as proposed by Lanczos and used by numerical analysts. The Lanczos method for symmetric A calculates W directly such that $W^TAW = T$ is (symmetric) tridiagonal and $W^TW = I$, the identity matrix. This is in contrast to the conjugate Lanczos method which diagonalizes A. Let this tridiagonal matrix be

$$
T = \begin{bmatrix}
\alpha_1 & \beta_2 & & \\
\beta_2 & \alpha_2 & \beta_3 & \\
& \beta_3 & \alpha_3 & \ddots \\
& & \ddots & \ddots
\end{bmatrix}.
$$

Equating columns in the expression $AW = WT$, where W is column partitioned $W = [w_1, w_2, \cdots]$, gives the iteration

$$Aw_j = \beta_j w_{j-1} + \alpha_j w_j + \beta_{j+1} w_{j+1},$$

and setting $\beta_1 w_0 = 0$ generalizes this iteration.

This equation, along with $\alpha_j = w_j^T A w_j$ (from $W^T A W = T$), is the basis of the standard Lanczos algorithm [11]:

○ ALGORITHM 2.2.3 *(Symmetric) Orthogonal Lanczos*

$j := 0$
$w_0 := 0$ *vector*
$w_1 :=$ *random vector*
$r_0 := w_j$
$\beta_1 := 1$
while $\beta_{j+1} \neq 0$ *do* {
  $w_{j+1} := r_j / \beta_{j+1}$
  $j := j + 1$
  $\alpha_j := w_j^T A w_j$
  $r_j := A w_j - \alpha_j w_j - \beta_j w_{j-1}$
  $\beta_{j+1} := |r_j|$
}

○

The tri-diagonalization of A can be used to solve linear systems [25]: setting $z = Wy$ in $Az = b$ gives $AWy = b$, which after multiplying by W^T gives $W^T AWy = Ty = W^T b$. If T has the Cholesky factorization $T = LDL^T$, then $LDL^T y = W^T b$. Let r be the intermediate vector $r = L^T y$ and let $C = W(L^T)^{-1}$. Then $Cr = W(L^T)^{-1} L^T y = Wy = z$.

Then components of r and columns of C can be progressively calculated and z can be accumulated, directly during the Lanczos procedure. A difficulty with this procedure lies in the fact that it requires square roots (to calculate $|r_j|$). This operation is problematic in prime fields, where it is not assured that every element has a square root (in fact, only one-half of the nonzero elements have one).

The orthogonal Lanczos method can be retained, and the square root removed, by relaxing the condition $W^T W = I$ to instead $W^T W = D$, a diagonal matrix. In addition, the idea of introducing the intermediate

$$v_j = Aw_j - \beta_j w_{j-1}$$

can be used to reduce storage. Now if $D = [\delta_1, \delta_2, \cdots]$, then the relaxed condition implies $\alpha_j = w_j^T Aw_j / \delta_j$. Also, $W^T AW = DT$ is symmetric, tridiagonal. We are free to choose

$$T = \begin{bmatrix} \alpha_1 & \beta_2 & & \\ 1 & \alpha_2 & \beta_3 & \\ & 1 & \alpha_3 & \ddots \\ & & \ddots & \ddots \end{bmatrix},$$

and then $\delta_k = \delta_{k-1}\beta_k$, or $\beta_k = \delta_k / \delta_{k-1}$. For this choice,

$$DT = \begin{bmatrix} \alpha_1\delta_1 & \delta_2 & & \\ \delta_2 & \alpha_2\delta_2 & \delta_3 & \\ & \delta_3 & \alpha_3\delta_3 & \ddots \\ & & \ddots & \ddots \end{bmatrix}.$$

In order to use orthogonality more fully to reduce the storage in this algorithm, define $v_j = Aw_j - \beta w_{j-1}$, which can be used to calculate α instead of Aw_j which, as will be seen, saves one vector of storage. An orthogonal Lanczos procedure without square roots is then:

○ ALGORITHM 2.2.4 *(Symmetric) Square-root-free Orthogonal Lanczos*

$j := 1$
$w_j :=$ *random vector*
$\delta_j := w_j^T w_j$
$v_j := Aw_j$

$\beta_1 := 1$
while $\delta_j \neq 0$ *do* {
    $\alpha_j := w_j^T v_j$
    $w_{j+1} := v_j - \alpha_j w_j$
    $j := j+1$
    $\delta_j := w_j^T w_j$
    $\beta_j := \delta_j / \delta_{j-1}$
    $v_j := Aw_j - \beta_j w_{j-1}$
}

○

If this algorithm is successful (δ_j is not zero prematurely), then A is tri-diagonalized. As mentioned previously, this tridiagonalization can be used to solve the linear system [25]. Setting $z = Wy$ in $Az = b$ gives $AWy = b$, which after multiplying by W^T gives $W^T AWy = DTy = W^T b$.

Let DT have the Cholesky factorization $DT = L\Delta L^T$, where Δ is the diagonal matrix $\Delta = [\Delta_1, \Delta_2, \cdots]$, and L is the lower-bidiagonal matrix

$$L = \begin{bmatrix} 1 & & & \\ \mu_1 & 1 & & \\ & \mu_2 & 1 & \\ & & \ddots & \ddots \end{bmatrix}.$$

Then $L\Delta L^T y = W^T b$. If the intermediate variable r is defined to be $L^T y$, and if C is defined by $C = W(L^T)^{-1}$, then $Cr = Wy = z$. The components of $r = [\rho_1, \rho_2, \cdots]^T$ and the columns of $C = [c_1, c_2, \cdots]$ can be calculated progressively as the algorithm proceeds, and updates to $z = Cr$ can be accumulated.

First the Cholesky factorization of DT is needed. This can be calculated directly:

○ ALGORITHM 2.2.5 *Cholesky*

$\Delta_1 := \alpha_1 \delta_1$

for $j > 1$ *do* {

$\mu_{j-1} := \delta_j / \Delta_{j-1}$

$\Delta_j := \delta_j(\alpha_j - \mu_{j-1})$

}

○

Since $CL^T = W$,

$$[c_1, \mu_1 c_1 + c_2, \cdots, \mu_{j-1} c_{j-1} + c_j, \cdots] = [w_1, w_2, \cdots],$$

then $c_j = w_j - \mu_{j-1} c_{j-1}$, for $j > 1$.

Also $L\Delta r = W^T b$, so then the components of r are $\rho_j = (w_j^T b - \mu_{j-1} \Delta_{j-1} \rho_{j-1}) / \Delta_j$, and z can be updated with $z := z + \rho_j c_j$.

If w_1 is chosen to be b, then since $W^T W = D$, a diagonal, all products $w_j^T b$ will vanish for $j > 1$, and so b will not have to be retained, saving one vector.

Assembling this into a new algorithm:

○ ALGORITHM 2.2.6 *(Symmetric) Orthogonal Lanczos Linear Solver*

$j := 1$

$w_1 := b$

$\delta_1 := w_1^T w_1$

$v_1 := A w_1$

while $\delta_j \neq 0$ *do* {

$\alpha_j := w_j^T v_j / \delta_j$

if $j = 1$ *then* {

$\Delta_j := \alpha_j \delta_j$

$c_j := w_j$

$\rho_j := w_j^T w_j / \Delta_j$

$z_j := \rho_j c_j$

} *else* {

$$
\begin{aligned}
&\qquad\quad \mu_{j-1} := \delta_j/\Delta_{j-1} \\
&\qquad\quad \Delta_j := \delta_j(\alpha_j - \mu_{j-1}) \\
&\qquad\quad c_j := w_j - \mu_{j-1}c_{j-1} \\
&\qquad\quad \rho_j := -\mu_{j-1}\Delta_{j-1}\rho_{j-1}/\Delta_j \\
&\qquad\quad z_j := z_{j-1} + \rho_j c_j \\
&\quad \} \\
&\quad w_{j+1} := v_j - \alpha_j w_j \\
&\quad j := j+1 \\
&\quad \delta_j := w_j^T w_j \\
&\quad \beta_j := \delta_j/\delta_{j-1} \\
&\quad v_j := Aw_j - \beta_j w_{j-1} \\
&\}
\end{aligned}
$$

○

This algorithm requires the vectors $w_{j+1}, w_j, w_{j-1}, v_j, c_j$ and z. This can be reduced to five, since w_{j+1} and v_j can share storage in $w_{j+1} := v_j - \alpha_j w_j$, and v_j and w_{j-1} can share storage in $v_j := Aw_j - \beta_j w_{j-1}$.

2.3 Bidiagonalization and Linear System Solution

As has been shown above, both the Lanczos and conjugate gradient methods depend upon the Lanczos tridiagonalization of a symmetric matrix. For large prime systems, the size of the vectors involved in these algorithms dominates the storage requirements: the least number of vectors required for implementing the Lanczos methods is five, and for the conjugate gradient method, four. In addition, the methods work with symmetric matrices, so to solve the non-symmetric system $Az = b$, the system $A^TAz = A^Tb$ is solved.

In contrast to these methods, which are based on Lanczos tridiagonalization, we will develop methods based on bidiagonalization, in order to try to reduce the number of vectors. Bidiagonalization is sometimes used as the first stage of the singular value decomposition, and can be obtained by Householder transformations [14, 11] (which have good stability properties). For sparse matrices, the Householder transformations will cause the matrix to become dense, so as an alternative, an algorithm resembling the Lanczos algorithm can be devised. A version of the method of Golub and Kahan [9] finds orthonormal matrices U and V ($U^TU = I$, $V^TV = I$) such that $V^TAU = L$, where L is lower bidiagonal of the form

$$L = \begin{bmatrix} \alpha_1 & & & & \\ \beta_2 & \alpha_2 & & & \\ & \beta_3 & \alpha_3 & & \\ & & \ddots & \ddots & \end{bmatrix}. \tag{2.1}$$

Taking the equation $AU = VL$, and multiplying on the left by V^T and on the right by U^T, $V^TA = LU^T$ is obtained. Transpose this equation to get $A^TV = UL^T$, which along with $AU = VL$ is used to obtain a recurrence to determine the vectors u_i and v_i, where $U = [u_1, u_2, \cdots]$ and $V = [v_1, v_2, \cdots]$ are column decompositions. Choose v_1 an arbitrary unit vector; then $A^TV = UL^T$ implies that $\alpha_1 u_1 = A^Tv_1$. The scalar α_1 is determined from the normality of u_1; then $\alpha_1 = |A^Tv_1|$, where $|\cdot|$ implies the normal Euclidean norm.

Next, $AU = VL$ implies that $Au_1 = \alpha_1 v_1 + \beta_2 v_2$ (where β_2 is again determined by normality), and in general that $Au_i = \alpha_i v_i + \beta_{i+1} v_{i+1}$. For $i > 1$, $A^TV = UL^T$ gives $A^Tv_i = \alpha_i u_i + \beta_i u_{i-1}$. Combining these gives an algorithm essentially the same as the Golub-Kahan bidiagonalization:

○ ALGORITHM 2.3.1 *Golub-Kahan Bidiagonalization*

$\beta_1 := 0$
$u_0 := 0$ *vector*
v_1 *arbitrary unit vector*
$i := 1$
do {
  $u_i := A^Tv_i - \beta_i u_{i-1}$
  $\alpha_i := |u_i|$
  if $\alpha_i = 0$ *then break*
  $u_i := \frac{1}{\alpha_i} u_i$
  $v_{i+1} := Au_i - \alpha_i v_i$

if $i > n$ *then done*
$i \;:=\; i + 1$
$\beta_i \;:=\; |v_i|$
if $\beta_i = 0$ *then break*
$v_i \;:=\; \frac{1}{\beta_i} v_i$;

}

○

Here A is assumed to be an $m \times n$ matrix, with $m > n$. If either α_i or β_i becomes 0 prematurely (the break conditions), assume that the algorithm is restarted with a new v_1; termination conditions will be dealt with later. Application of the Golub and Kahan bidiagonalization to least squares, and also therefore to linear system solution, was given by Paige [26].

To solve $Az = b$, since we are free to choose v_1, set $v_1 = b/|b|$. Also set as intermediate $z = Uy$. Then $Az = b$ becomes $AUy = b$, or equivalently (from the form of the tridiagonalization) $VLy = b$. Since V is orthonormal, $Ly = V^T b$, which by orthogonality and choice of v_1 simplifies to $Ly = [1, 0, 0, \cdots]^T$. This system is easily solved by forward substitution for y: $y_1 = 1/\alpha_1$ and $y_i = -(\beta_i/\alpha_i)y_{i-1}$.

The components of y can then be determined as Algorithm 2.3.1 proceeds, and in addition, $z = Uy$ can be updated by each new column of U with y_i as computation progresses. In this way, a total of three vectors (one for u_i, v_i and z) are required to find the linear solution, making this method extremely attractive for linear solution modulo large primes. The iterative equations have the property that scaling in place and addition of matrix vector products in place are both possible, so no temporary vectors are required.

When considered modulo a large prime, this method suffers from the same possibility in Lanczos or conjugate gradients, that the magnitude of a vector is zero when the vector itself is not. These magnitudes are used for normalizing the vectors u_i and v_i. For large prime moduli, the occurrence of zero magnitude vectors is expected to be extremely rare, as it is for the Lanczos and conjugate gradient methods.

There is, however, a much more damaging difficulty when this method is used modulo a prime. The magnitude calculations require a square root, which is no difficulty in the real field, but in a prime field, there is the difficulty that one-half of the nonzero elements do not have a square root; that is, they are quadratic non-residues.

From this standpoint, the bidiagonalization linear system solution in a prime field can only be considered as a probabilistic method, and one with a very small likelihood of success, since only norms that are quadratic residues may occur in a successful solution.

This method is, then, not immediately useful for discrete logarithm application, but since it has such small vector storage usage, it is important to see if it can be adapted to work reliably in a prime field.

2.4 A Bidiagonalization Method for Prime Fields

The bidiagonalization method of the previous section suffers from non-existent square roots. In an attempt to remove these square roots from the method, we propose using non-normalized matrices for U and V. Let these matrices instead have the property that $U\bar{U}^T = I$ and $\bar{V}^T V = I$, where $\bar{U}$ is a matrix having column directions identical to U, except with possibly different (nonzero) column scaling, and $\bar{V}$ being defined similarly. If U and V are invertible (as is the case during the procedure of bidiagonalization), then $\bar{U}^T U = I$ and $V\bar{V}^T = I$ as well. Such a method would not usually be considered in numerical work, where transformation by unitary matrices is one step toward eigenvalue determination.

Beginning again with $AU = VL$, multiply on the left by $\bar{V}^T$ and on the right with $\bar{U}^T$ to obtain $\bar{V}^T A = L\bar{U}^T$. Transposing, this becomes $A^T\bar{V} = \bar{U}L^T$, which along with the equation $AU = VL$, defines the iterations $A^T\bar{v}_i = \alpha_i\bar{u}_i + \beta_i\bar{u}_{i-1}$ and $Au_i = \alpha_i v_i + \beta_{i+1}v_{i+1}$, where $\bar{U} = [\bar{u}_1, \bar{u}_2, \cdots]$ and $\bar{V} = [\bar{v}_1, \bar{v}_2, \cdots]$ are again column decompositions. Conversions between u_i and $\bar{u}_i$ are found by $u_i^T\bar{u}_i = 1$, and v_i similarly. Combining, an algorithm free from square roots is obtained:

○ ALGORITHM 2.4.1 *Square-root-free Bidiagonalization*

$\beta_1 := 0$

$\bar{u}_0 := 0$ *vector*

$\bar{v}_1$ *arbitrary (nonzero) vector*

$v_1 := \frac{1}{\bar{v}_1^T\bar{v}_1}\bar{v}_1$

$i := 1$

do {

$\bar{u}_i := A^T\bar{v}_i - \beta_i\bar{u}_{i-1}$
choose α_i *arbitrarily (nonzero)*
$\bar{u}_i := \frac{1}{\alpha_i}\bar{u}_i$
if $\bar{u}_i^T\bar{u}_i = 0$ *then break*
$u_i := \frac{1}{\bar{u}_i^T\bar{u}_i}\bar{u}_i$
if $i > n$ *then done*
$v_{i+1} := Au_i - \alpha_i v_i$
$i := i+1$
choose β_i *arbitrarily (nonzero)*
$v_i := \frac{1}{\beta_i}v_i$
if $v_i^T v_i = 0$ *then break*
$\bar{v}_i := \frac{1}{u_i^T u_i}u_i$

}

○

Lemma 2.4.1 *The pairs of matrices* U, $\bar{U}$, *and* V, $\bar{V}$, *as generated by algorithm 2.4.1, are orthogonal:* $U^T\bar{U} = I$, $V^T\bar{V} = I$.

PROOF. Algorithm 2.4.1 iterates $A^T\bar{v}_i = \alpha_i\bar{u}_i + \beta_i\bar{u}_{i-1}$ and $Au_i = \alpha_i v_i + \beta_{i+1}v_{i+1}$. Let U, $\bar{U}$, V, $\bar{V}$ be the partially-completed matrices at the k-th iteration (partitioned by column $U = [u_1, u_2, \cdots u_k]$ and the others likewise).

When $k = 1$, orthogonality holds, so proceed by induction. Assuming the matrices are orthogonal at the k-th iteration, then the next v vector satisfies $AU = VL + \beta_{k+1}v_{k+1}e_k^T$, where e_k is the last column of the $k \times k$ identity matrix. Multiplying by $\bar{V}^T$, then $\bar{V}^T AU = \bar{V}^T VL + \beta_{k+1}\bar{V}^T v_{k+1}e_k^T$, but since $A^T\bar{V} = \bar{U}L^T$, then $\bar{V}^T A = L\bar{U}^T$. Therefore $\bar{V}^T VL + \beta_{k+1}\bar{V}^T v_{k+1}e_k^T = \bar{V}^T AU = L\bar{U}^T U$. The induction assumption reduces $\bar{V}^T V$ and $\bar{U}^T U$ to identity, so then assuming that β_{k+1} is not zero (i.e. that the algorithm does not fail at this point), then $\bar{V}^T v_{k+1} = 0$, so the next v vector is orthogonal to its predecessors.

Next the $\bar{u}$ vector is determined from $A^T\bar{v}_{k+1} = \alpha_{k+1}\bar{u}_{k+1} + \beta_{k+1}\bar{u}_k$, which, when taken with the previous $\bar{u}$ vectors, is $A^T[\bar{V}, \bar{v}_{k+1}] = [\bar{U}, \bar{u}_{k+1}]L_+^T$, where L_+ is the bidiagonal matrix for iteration $k+1$. Equivalently, $A^T[\bar{V}, \bar{v}_{k+1}] = \bar{U}[L^T, \beta_{k+1}e_k] + \alpha_{k+1}\bar{u}_{k+1}e_{k+1}$. This will be multiplied by U^T, but from $AU = VL$, then $[U, u_{k+1}]^T A^T =$

$L_+^T[V, v_{k+1}]^T$, and therefore (deleting the last row from the first matrix on both left-hand and right-hand sides), $U_T A^T = [L^T, \beta_{k+1}e_k][V, v_{k+1}]^T$. Thus

$$\begin{aligned} U^T A^T[\bar{V}, \bar{v}_{k+1}] &= U^T \bar{U}[L^T, \beta_{k+1}e_k] + \alpha_{k+1}U^T \bar{u}_{k+1}e_{k+1} \\ &= [L^T, \beta_{k+1}e_k][V, v_{k+1}]^T[\bar{V}, \bar{v}_{k+1}], \end{aligned}$$

so again, if α_{k+1} is not zero, then $U^T\bar{u}_{k+1} = 0$. (Transformation from $\bar{U}$ to U is just a column scaling chosen so that $u_i\bar{u}_i^T = 1$, and similarly for V.)

Therefore, both u and v vectors are orthogonal to their predecessors. □

Theorem 2.4.1 *When successful, algorithm 2.4.1 completes in at most* $\min(m, n)$ *iterations, where A is of dimension $m \times n$.*

PROOF. Since V and U are orthogonal, algorithm 2.4.1 must complete in at most $\min(m, n)$ iterations, since rank$(A) \leq \min(m, n)$. □

If this algorithm completes, then it too can be used to obtain a solution to $Az = b$: choose $\bar{v}_1 = b$ and set as intermediate $z = Uy$. Then $Az = b$ becomes $AUy = b$, which is $VLy = b$. Thus $Ly = \bar{V}^T b$, which by orthogonality and choice of $\bar{v}_1$ simplifies to $Ly = [b^T b, 0, 0, \cdots]^T$. Solving for y: $y_1 = (b^T b)/\alpha_1$ and $y_i = -(\beta_i/\alpha_i)y_{i-1}$.

The solution $z = Uy$ can again be accumulated during the procedure. Only one of $\bar{u}$ or u and $\bar{v}$ or v need be stored at any time, so this new algorithm needs only three vectors to complete the linear solution.

An interesting feature of this method is the freedom available in choosing α_i and β_i, which represent the diagonal and sub-diagonal elements of the matrix L. It might appear that this freedom in choosing different α_i and β_i could provide some benefit in bypassing breakdowns in the method, especially since these iterations are completely reversible, but it will be seen that, unfortunately, this is not the case.

First, if a vector v has the property that $v^T v = 0$ modulo some prime, then so will any scaling of that vector δv. Therefore, it is the direction of a vector that makes it dangerous for the linear solution algorithm.

Lemma 2.4.2 *The forms of the vectors v_i and $\bar{u}_i$ are*

$$v_i = \left(\frac{\prod_{j=1}^{i-1} \alpha_j}{\prod_{j=2}^{i} \beta_j} \right) d_i,$$

where $d_i = Ab_{i-1}/(b_{i-1}^T b_{i-1}) - d_{i-1}$, *for* $i > 1$ *and* $d_1 = v_1$, *and*

$$\bar{u}_i = \left(\frac{\prod_{j=2}^{i} \beta_j}{\prod_{j=1}^{i} \alpha_j} \right) b_i,$$

where $b_i = A^T d_i/(d_i^T d_i) - b_{i-1}$, *for* $i > 0$, *and* $b_0 = u_0$.

PROOF. To prove this, let b_1 be$A^T \bar{v}_1$ and let d_1 be v_1, which is determined by the freely chosen $\bar{v}_1$, then use induction. Then $\alpha_i \bar{u}_i = A^T \bar{v}_i - \beta_i \bar{u}_{i-1}$, and

$$\bar{v}_i = v_i/(v_i^T v_i) = \left(\frac{\prod_{j=2}^{i} \beta_j}{\prod_{j=1}^{i-1} \alpha_j} \right) \frac{d_i}{(d_i^T d_i)}.$$

Then

$$\alpha_i \bar{u}_i = A^T \left(\frac{\prod_{j=2}^{i} \beta_j}{\prod_{j=1}^{i-1} \alpha_j} \right) \frac{d_i}{(d_i^T d_i)} - \beta_i \left(\frac{\prod_{j=2}^{i-1} \beta_j}{\prod_{j=1}^{i-1} \alpha_j} \right) b_{i-1},$$

and

$$\bar{u}_i = \left(\frac{\prod_{j=2}^{i} \beta_j}{\prod_{j=1}^{i} \alpha_j} \right) b_i.$$

Similarly, v_i has the form given above. □

This means that the direction vectors are independent of the choice of α_i and β_i, so they cannot be chosen to avoid breakdown conditions. Instead, they may be chosen to make the algorithm as efficient as possible. To determine the solution, U must be determined, but the iteration gives only $\bar{U}$ directly. It is then appropriate to allocate all scaling to the iteration defining $\bar{U}$, since it cannot be avoided there. Taking this into account, and also trying to minimize the number of negations required, an attractive choice is to take $\alpha_i = -1$, $\beta_i = 1$, for $i > 0$.

The iteration then becomes $A^T \bar{v}_i - \bar{u}_{i-1} = -\bar{u}_i$ and $Au_i + v_i = v_{i+1}$. Scaling by $v_i^T v_i$ changes $\bar{v}_i$ to v_i in the first equation, which becomes

$$A^T v_i - (v_i^T v_i)/(u_{i-1}^T u_{i-1}) u_{i-1} = (v_i^T v_i)(-\bar{u}_i).$$

Choose $\bar{v}_1 = -b$, $z = Uy$, then $y_1 = b^T b$, $y_i = y_1$. Combining this, our final algorithm based on bidiagonalization for linear system solution is obtained:

○ ALGORITHM 2.4.2 *Bidiagonal Linear Solver*

$z := u := 0$ *vector*
$\lambda := b^T b$
$\rho := 1$
$\delta := -1/\lambda$
$v = \delta b$
$i := 1$
do {
    $u := (\delta/\rho)u + A^T v$
    $\theta := (u^T u)/\delta$
    if $\theta = 0$ *then break*
    $\rho := \delta/\theta$
    $u := u/\theta$
    $z := z + u$
    $i := i + 1$
    if $i > n$ *then break*
    $v := v + Au$
    $\delta := -(v^T v)$
    if $\delta = 0$ *then break*
}
$z := \lambda z$

○

Here all the lowercase Greek characters are scalars, lowercase Roman characters are vectors and uppercase are matrices. The main loop is free of negations, except for one scalar negation. Three vectors are sufficient to accumulate the solution. Two matrix-vector multiplications, two magnitude calculations, two vector scalings and three vector additions are required for an iteration, plus three scalar divisions.

2.5 Summary

At the beginning of the chapter, the linear system solution methods that have been applied to discrete logarithm calculation were reviewed, and criteria for evaluating them were given. A comparison of the methods was offered. Derivations of the exact iterative methods were then presented, including the orthogonal Lanczos algorithm, which has not been applied to discrete logarithm calculation previously. After an improvement to the conjugate version, both it and the orthogonal Lanczos algorithm were seen to require five vectors. The conjugate gradient algorithm was examined and found to require only four vectors.

Following this, an adaptation of the bidiagonalization method of Golub and Kahan [9] was shown to be the basis of a linear system solution method requiring only three vectors, and no more computational effort than the other methods. It is expected that this method will become the method of choice for solving the normal equation systems of equations in large primes.

In this chapter, only methods for symmetric systems (obtained via $A = B^T B$) have been considered, and the question of breakdown has been largely ignored. In the next chapter a consideration of the more general non-symmetric case is explored, and the question of breakdown, its causes and cures, are explored.

Chapter 3

Orthogonal Polynomials and Krylov Subspace Methods

This chapter will unify the Lanczos, conjugate gradient and Wiedemann algorithms using a polynomial interpretation of these methods. The relevance of orthogonal polynomials to the Lanczos method was recognized at the outset [20]. Three-term polynomial recurrences similar to the Lanczos method are known in system theory and Padé approximation. The polynomial interpretation facilitates a study of the breakdown phenomenon, and so it is developed here, after the statement of the bi-orthogonal Lanczos algorithm.

From this development, a Hankel matrix is introduced naturally, and an immediate tie with the Wiedemann algorithm is drawn. The breakdown phenomenon is explained in the polynomial setting and lookahead procedures are derived with simple motivation. An analogue of Taylor's mismatch theorem [27] is displayed, and another connection between Lanczos and Wiedemann is drawn.

The lookahead method is then developed using vectors. Up to this point, the development covers both the orthogonal and conjugate Lanczos procedures in a unified way. At the point of using the Lanczos procedure for linear solution, the conjugate and orthogonal methods diverge, and the exposition separates.

For the orthogonal method, a means of dealing with the more complicated system is developed. The relation of the conjugate gradient method to a combined orthogonal-conjugate Lanczos procedure is derived, and the phenomenon of breakdowns is examined for that case. Given an assumption concerning the random distribution of certain components, expressions for the likelihood of breakdown and the expected degree are then given.

The background literature on the Lanczos and related methods over the reals is quite large, and so it is important to note what is new here. The polynomial interpretation of the Lanczos algorithm dates back to Lanczos, and the moment matrix and inner product in polynomial terms are often defined [33]. The form of the matrices used to develop the lookahead methods is from the system theory work of Rissanen [32]. The connection of the Lanczos method to the Wiedemann algorithm is new. Lookahead methods have been given in a numerical analysis context, first by Parlett et al., and then also by others [7, 37, 15], but the algorithm given here is simpler yet efficient, although no concern is taken for the stability of this method. The lookahead method given here becomes the standard Lanczos iteration at its minimum, unlike some other methods. Teitelbaum [35] has recently given a development of the lookahead Lanczos algorithm for the non-singular symmetric case over finite fields. We have developed the general non-symmetric, possibly singular case (techniques for non-square matrices are also implicit in this).

Both orthogonal (as used in numerical analysis literature) and conjugate linear system solvers are developed, exploiting the connection to the Wiedemann algorithm in the case of incurable breakdown and in other cases where the solution is not directly available. In this direction, we prove an analogue of Taylor's mismatch theorem for general fields in a simple way. The orthogonal system solver requires the solution of matrices more complicated than in the conjugate case, and for this a method is developed. Guidelines for the choice of initial vectors are given.

The relation of the conjugate gradient method to the Lanczos process also dates back to the first presentation of the conjugate gradient method, and more explicit demonstrations have been given [14, 5], but the relationship given here is very simple and direct, and leads to a development of lookahead methods for the conjugate gradient method which, to our knowledge, have not been presented previously.

In this way, all of the sparse solution methods proposed and developed (Lanczos, conjugate gradient, and Wiedemann) for the linear solution stage of sieving methods for discrete logarithms (and by relation, for factoring algorithms) have been unified into a simple theory.

3.1 The Bi-Orthogonal Lanczos Algorithm

For the purpose of solving the linear system $Az = b$, where A is non-symmetric, the more general bi-orthogonal version of the Lanczos process is required. The bi-orthogonal Lanczos algorithm [33] begins with a possibly non-symmetric square matrix A of dimension n and two vectors (v_0, u_0), with which it successively builds vector sequences

$\{u_0, u_1, \cdots, u_{i-1}\}$ and $\{v_0, v_1, \cdots, v_{i-1}\}$, such that

$$\operatorname{span}(\{u_0, u_1, \cdots, u_{i-1}\}) = K_i(A, u_0) = \{u_0, Au_0, \cdots A^{i-1}u_0\}$$

and

$$\operatorname{span}(\{v_0, v_1, \cdots, v_{i-1}\}) = K_i(A^T, v_0) = \{v_0, A^T v_0, \cdots (A^T)^{i-1} v_0\},$$

with the added condition that these sequences are bi-orthogonal; that is

$$v_i^T u_j = \begin{cases} \delta_i \neq 0 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases},$$

for all $0 \leq i, j < n$.

The algorithm uses the recurrences

$$u_{j+1} = Au_j - \alpha_j u_j - \beta_j u_{j-1}, \tag{3.1}$$
$$v_{j+1} = A^T v_j - \alpha_j v_j - \beta_j v_{j-1}, \tag{3.2}$$

where $\alpha_j = \frac{v_j^T A u_j}{\delta_j}$ and $\beta_j = \frac{v_{j-1}^T A u_j}{\delta_{j-1}} = \frac{\delta_j}{\delta_{j-1}}$ are projections bi-orthogonalizing the new vectors.

Now take a polynomial view of the process. The recurrence formulas imply that $u_j = \phi_j(A)u_0$ and $v_j = \phi_j(A^T)v_0$, where ϕ_j is a monic polynomial of degree j.

Defining the product $(\phi, \psi) = v_0^T \phi(A)\psi(A)u_0$, we wish to re-phrase the bi-orthogonality condition in terms of polynomials. First,

$$(\phi_i, \phi_j) = \begin{cases} \delta_i \neq 0 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases},$$

but since $\{u_0, u_1, \cdots, u_{i-1}\}$ spans $K_i(A, u_0)$, then $(\phi_i, \psi) = 0$ for every polynomial ψ of degree $i-1$ or less. Similarly, $(\psi, \phi_i) = 0$ for ψ having degree $i-1$ or less.

Let ϕ_i have components $\phi_i(x) = \phi_{i,0} + \phi_{i,1}x^1 + \cdots + \phi_{i,i}x^i$. Choosing for ψ now the monomial basis, the bi-orthogonality condition becomes

$$\begin{bmatrix} \mu_0 & \mu_1 & \cdots & \mu_i \\ \mu_1 & \mu_2 & \cdots & \mu_{i+1} \\ \vdots & \vdots & \ddots & \vdots \\ \mu_{i-1} & \mu_i & \cdots & \mu_{2i-1} \end{bmatrix} \begin{bmatrix} \phi_{i,0} \\ \phi_{i,1} \\ \vdots \\ \phi_{i,i} \end{bmatrix} = 0, \tag{3.3}$$

where $\mu_i = v_0^T A^i u_0$. Here each increasing row comes from the orthogonality condition with an increasing power of the monomial ψ. Throughout the remainder of this chapter, the orthogonality will be only with respect to the inner product defined above.

If we assume that ϕ_i is monic, then this equation is familiar as the Berlekamp-Massey equation given in the last chapter. It is then clear that the Lanczos process is actually updating the minimal polynomial of the sequence $\{v_0^T A^0 u_0, v_0^T A^1 u_0, v_0^T A^2 u_0, \cdots\}$, which is the same sequence observed in the Wiedemann algorithm.

Therefore, if polynomials are updated during the Lanczos process:

$$p_{j+1}(x) = (x - \alpha_j)p_j(x) - \beta_j p_{j-1}(x),$$

with $p_0(x) = 1$; then p_j is the solution of the Berlekamp-Massey algorithm presented with the sequence $\{\mu_0, \mu_1, \cdots, \mu_{2j-1}\}$, assuming that no breakdowns (zero denominators in the expressions for α and β) occur in the Lanczos process. We have shown:

Theorem 3.1.1 *Up to the point of breakdown, the Lanczos algorithm solves at each step for the same (minimal) polynomial as the Berlekamp-Massey algorithm.*

A similar development can be carried through for the conjugate Lanczos process and an appropriate inner product $(\phi, \psi)' = v_0^T \phi(A) A \psi(A) u_0$. The moments are then $\mu_i' = v_0^T A^{i+1} u_0$. This corresponds to applying the Berlekamp-Massey algorithm to the sequence $\{\mu_0', \mu_1', \cdots\} = \{\mu_1, \mu_2, \cdots\}$, or equivalently to a shift of the previous sequence, $\{\mu_0, \mu_1, \cdots\}$, which is immaterial.

We have then established an immediate, but novel, connection between the Lanczos and Wiedemann algorithms.

A difficultly involved with the non-symmetric Lanczos algorithm is the possibility that some δ_i is zero, without u_i and v_i being zero themselves (which is a problem over the reals as well with non-symmetric matrices). This problem is examined in the next section.

3.2 Breakdowns

In this section, we will examine the breakdowns of the Lanczos algorithm, and the means of circumventing them, namely lookahead methods. A polynomial version of the biorthogonal Lanczos algorithm will be developed and shown to be equivalent to the Lanczos method given above, when no breakdowns occur. The polynomial algorithm will be developed further in this section to include the case of breakdown.

Let the $j \times (j+1)$ Hankel matrix given in (3.3) be extended out to n, the dimension of A, and denoted by $H_{(n,n+1)}$. We will endeavour to use the factorization form of

Rissanen [32]:

$$H_{(n,n+1)} \begin{bmatrix} 1 & \phi_{1,0} & \phi_{2,0} & \cdots & \phi_{n,0} \\ & 1 & \phi_{2,1} & \cdots & \phi_{n,1} \\ & & 1 & & \vdots \\ & & & \ddots & \phi_{n,n-1} \\ & & & & 1 \end{bmatrix} = \begin{bmatrix} r_{0,0} & & & & 0 \\ r_{0,1} & r_{1,1} & & & \\ \vdots & \vdots & \ddots & & \vdots \\ r_{0,n-1} & r_{1,n-1} & \cdots & r_{n-1,n-1} & 0 \end{bmatrix},$$

or $H_{(n,n+1)}T_{(n+1,n+1)} = L_{(n,n+1)}$, where $T_{(n+1,n+1)}$ is upper triangular and $L_{(n,n+1)}$ is lower triangular with an added zero column. The last column of T is then the solution to the linear system solved by the Berlekamp-Massey algorithm.

The factorization into this form (via the Lanczos process) is recursive; the decreasing number of lower zeros in the columns of T reflect the increasing size of H as the algorithm progresses, and the increasing number of upper zeros in the columns of L reflect the orthogonalization of the vectors against past vectors.

The columns of T are polynomials, and elements are indexed first with the polynomial and then by coefficient, instead of in the usual row-column order.

When seen in the context of the Berlekamp-Massey algorithm, each new polynomial ϕ_i matches more and more terms of the moment sequence, so the zeros in L are zeros in the discrepancies defined in the Berlekamp-Massey algorithm. The discrepancy matrix L is also indexed first by the polynomial of the discrepancy.

Also note that if a column of T is shifted down by one, the corresponding column of L is shifted up by one (because of the zeros in the lower part of the vector and the Hankel structure of H). More precisely, if

$$H_{(n,n+1)} \begin{bmatrix} f_0 \\ f_1 \\ \vdots \\ 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ r_k \\ r_{k+1} \\ \vdots \\ r_{n-1} \end{bmatrix}, \tag{3.4}$$

then

$$H_{(n,n+1)} \begin{bmatrix} 0 \\ f_0 \\ f_1 \\ \vdots \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ r_k \\ r_{k+1} \\ \vdots \\ r_{n-1} \\ r_n \end{bmatrix}.$$

Since a column of T is interpreted as a polynomial (for example, the column in (3.4) is $f_0 + f_1 x + f_2 x^2 + \cdots + f_k x^k + x^{k+1}$), then shifting down is also scaling by x. This suggests the Lanczos algorithm as the means to produce matrices of this form, and scaling by x becomes scaling by A or A^T in the vectors.

Avoiding for the moment the initialization stage of the Lanczos algorithm, let the last two columns of the partially-completed T be given as polynomials:

$$H_{(n,n+1)}[\phi_{k-1}(x), \phi_k(x)] = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \vdots & \vdots \\ 0 & 0 \\ r_{k-1,k-1} & 0 \\ r_{k-1,k} & r_{k,k} \\ r_{k-1,k+1} & r_{k,k+1} \\ \vdots & \vdots \end{bmatrix},$$

where the columns of T have been given in polynomial representation.

The next polynomial ϕ_{k+1} is to have degree k. As a first version of ϕ_{k+1}, scale ϕ_k to $x\phi_k$. Then

$$H_{(n,n+1)}[\phi_{k-1}(x), \phi_k(x), x\phi_k(x)] = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \vdots & \vdots & \vdots \\ 0 & 0 & 0 \\ r_{k-1,k-1} & 0 & r_{k,k} \\ r_{k-1,k} & r_{k,k} & r_{k,k+1} \\ r_{k-1,k+1} & r_{k,k+1} & r_{k-1,k+1} \\ \vdots & \vdots & \vdots \end{bmatrix}, \tag{3.5}$$

and now using $r_{k-1,k-1}$ and $r_{k,k}$ as pivot elements to clear the extra two nonzero rows of the third column,

$$H_{(n,n+1)}[\phi_{k-1}(x), \phi_k(x), \phi_{k+1}(x) = x\phi_k(x) - \alpha_k\phi_k(x) - \beta_k\phi_{k-1}(x)] = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \vdots & \vdots & \vdots \\ 0 & 0 & 0 \\ r_{k-1,k-1} & 0 & 0 \\ r_{k-1,k} & r_{k,k} & 0 \\ \vdots & r_{k,k+1} & r_{k+1,k+1} \\ & \vdots & r_{k+1,k+2} \\ & & \vdots \end{bmatrix},$$

where $\beta_k = \frac{r_{k,k}}{r_{k-1,k-1}}$ and $\alpha_k = \frac{(r_{k,k+1} - \beta_k r_{k-1,k})}{r_{k,k}}$ are chosen to force two zeros into the third column.

The algorithm defined in this way is exactly the bi-orthogonal Lanczos algorithm. This is proven by showing that the constants α and β are the same as in the bi-orthogonal Lanczos algorithm. Before showing this, some notation will be introduced that will allow the development to also be applied to the conjugate Lanczos algorithm.

For the conjugate Lanczos process,

$$\begin{bmatrix} \mu_0' & \mu_1' & \mu_2' & \cdots \\ \mu_1' & \mu_2' & \cdots & \\ \mu_2' & \vdots & \ddots & \\ \vdots & & & \end{bmatrix} \begin{bmatrix} 1 & \phi_{1,0} & \phi_{2,0} & \cdots \\ & 1 & \phi_{2,1} & \cdots \\ & & 1 & \cdots \\ & & & \ddots \end{bmatrix} = \begin{bmatrix} r_{0,0} & & & \\ r_{0,1} & r_{1,1} & & \\ r_{0,2} & r_{1,2} & r_{2,2} & \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix},$$

or $H'_{(n,n+1)}T_{(n+1,n+1)} = L_{(n,n+1)}$. Then $r_{i,j} = v_0^T(x^{j+1}\phi_i(x))(A)u_0$. Therefore, ϕ_j is orthogonal to all polynomials with zero constant terms, of degree up to and including j.

To unify the development, define the brackets $\{f(x)\}$ to be $f(x)$ for the orthogonal case and $xf(x)$ for the conjugate case. Also let $[f(x)] = v_0^T\{f(A)\}u_0$. With this understanding, and letting $(\phi, \psi) = v_0^T\{(\phi\psi)(A)\}u_0$, then ϕ_j is orthogonal to all polynomials of degree less than j, for both the orthogonal and the conjugate case. Similarly for vectors, we can define (v, u) to be the vector cross-product $v^T u$ for the orthogonal case and $(v, u) = v^T Au$ for the conjugate case.

For the orthogonal Lanczos algorithm which was presented at the beginning of the chapter, $\beta_k = (v_k, u_k)/(v_{k-1}, u_{k-1}) = \delta_k/\delta_{k-1}$, where $\delta_k = (v_k, u_k)$ was defined there. For the polynomial algorithm, $\beta_k = \frac{[x^k\phi_k]}{[x^{k-1}\phi_{k-1}]}$, from the form of the matrices.

Using orthogonality, x^k can be replaced by $\phi_k(x)$ and x^{k-1} by $\phi_{k-1}(x)$ in these equations: $\beta_k = \frac{[\phi_k\phi_k]}{[\phi_{k-1}\phi_{k-1}]} = \frac{(v_k,u_k)}{(v_{k-1},u_{k-1})} = \frac{\delta_k}{\delta_{k-1}}$, as required. Now remove the quantity $\beta_k\phi_{k-1}$ from $x\phi_k$, yielding $\alpha_k = \frac{[x^k(x\phi_k-\beta_k\phi_{k-1})]}{[x^k\phi_k]}$. Then $\alpha_k = \frac{[x^k(x\phi_k-\beta_k\phi_{k-1})]}{\delta_k}$, by the same argument as for β_k.

Polynomials $x\phi_k(x)$ and $\phi_{k-1}(x)$ are orthogonal to x^j for $j < k-1$, so x^k can be replaced by $\phi_k(x) - \phi_{k,k-1}x^{k-1}$ without changing the value, where $\phi_{k,k-1}$ is the second highest coefficient of $\phi_k(x)$. Then,

$$\begin{aligned}\alpha_k &= [(\phi_k(x) - \phi_{k,k-1}x^{k-1})(x\phi_k(x) - \beta_k\phi_{k-1}(x))]/\delta_k \\ &= [\phi_k(x)x\phi_k(x) - \beta_k\phi_k(x)\phi_{k-1}(x) - \phi_{k,k-1}x^{k-1}x\phi_k(x) + \beta_k\phi_{k,k-1}x^{k-1}\phi_{k-1}(x)]/\delta_k.\end{aligned}$$

The second term in this last expression is zero by orthogonality, which also allows for the replacement of x^k and x^{k-1} by ϕ_k and ϕ_{k-1}, respectively. With this substitution,

$$\begin{aligned}\alpha_k &= [\phi_k(x)x\phi_k(x) - \phi_{k,k-1}(\beta_k\phi_{k-1}(x)\phi_{k-1}(x) - \phi_k(x)\phi_k(x))]/\delta_k \\ &= ([\phi_k(x)x\phi_k(x)] + \phi_{k,k-1}(\frac{\delta_k}{\delta_{k-1}}\delta_{k-1} - \delta_k))/\delta_k \\ &= (v_k, Au_k)/\delta_k.\end{aligned}$$

Therefore, the polynomial procedure is precisely the bi-orthogonal Lanczos algorithm:

Theorem 3.2.1 *The polynomial iteration $\phi_{k+1}(x) = x\phi_k(x) - \alpha_k\phi_k(x) - \beta_k\phi_{k-1}(x)$, where $\alpha_k = \frac{r_{k,k+1}-\beta_k r_{k-1,k}}{r_{k,k}}$ and $\beta_k = \frac{r_{k,k}}{r_{k-1,k-1}}$ is exactly the standard Lanczos procedure.*

For the conjugate case, this implies that $\beta_k = \frac{v_k^T Au_k}{v_{k-1}^T Au_{k-1}}$ and $\alpha_k = \frac{(Av_k)^T(Au_k)}{v_k^T Au_k}$ which, in the symmetric case, gives the same quantities as the conjugate Lanczos algorithm described in the previous chapter.

Note that by re-associating the polynomial product, one portion to the v_0 and one to the u_0 side, other expressions of α and β are obtained, for both conjugate and orthogonal versions. For example, $\alpha_k = (A^T v_k, u_k)/\delta_k$ in the orthogonal case.

We are now in a position to consider the Lanczos and Wiedemann algorithms as special cases of the polynomial algorithm described above. The Wiedemann algorithm is derived by ignoring the vector portion of the process until a factor of the minimal polynomial is obtained (these polynomials are obtained via the Berlekamp-Massey algorithm). The Lanczos procedure is derived by ignoring the polynomials and dealing only with the vector recurrences.

It is apparent that a hybrid is possible: an algorithm that maintains vectors giving the efficiency of the Lanczos algorithm, but retains polynomials to emulate the Wiedemann algorithm in the case of breakdown in the Lanczos algorithm, as will be described later.

The bi-orthogonal Lanczos algorithm is said to break down when $\delta_k = (v_k, u_k) = 0$ without v_k or u_k being zero themselves. Breakdown is possible for arithmetic in a finite field even when A is symmetric and non-singular. This possibility is the reason that the Lanczos and related algorithms may have been considered to be heuristic methods over finite fields (even though this is not the case).

If v_k or u_k are themselves 0, then the Lanczos algorithm has terminated successfully. In the application of the Lanczos method to linear system solution, more will be said about successful termination and the solution of linear systems.

In the course of the polynomial Lanczos algorithm, in defining ϕ_k, all discrepancies above $r_{k,k}$ are forced to zero, from which an initial ϕ_{k+1} is formed by shifting ϕ_k to $x\phi_k$. If $r_{k,k}$ also happens to be zero, then it cannot be used to produce ϕ_{k+1} directly, since the vector obtained cannot be used to force the required zeros (consider the first three columns of equation (3.6), and compare to equation (3.5)):

$$H_{(n,n+1)}[\phi_{k-1}(x), \phi_k(x), x\phi_k(x), x^2\phi_k(x)] = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 \\ r_{k-1,k-1} & 0 & 0 & r_{k,k+1} \\ r_{k-1,k} & 0 & r_{k,k+1} & r_{k,k+2} \\ \vdots & r_{k,k+1} & r_{k,k+2} & \vdots \\ & r_{k,k+2} & \vdots & \\ & \vdots & & \end{bmatrix}. \tag{3.6}$$

Now, in equation (3.6), ϕ_k must be shifted by x and x^2 in order to obtain a vector that can be reduced. In other words, it is necessary to lookahead one more step in order to get a vector that can be reduced.

The discrepancy vector corresponding to $x^2\phi_k$ has three zeros injected into it by adding scaled vectors corresponding to the discrepancies of ϕ_{k-1}, ϕ_k, and $x\phi_k$. This is a generalization of the polynomial Lanczos procedure, and is a type of lookahead algorithm, several of which have been considered in a numerical analysis context [27].

The extra zero occurring in the vector of ϕ_k means that ϕ_k has a zero discrepancy for the next term in the sequence $\{\mu_0, \mu_1, \cdots\}$, when interpreted via the Berlekamp-Massey algorithm. This seemingly fortunate occurrence is the cause of breakdowns in the Lanczos procedure, and is related to double cancellation in the Euclidean algorithm for polynomials. (The Berlekamp-Massey algorithm simply retains the same polynomial when it already has the required zero discrepancy, but does its own version of 'lookahead' by retaining the last polynomial that required updating, so it might be said that Berlekamp-Massey looks back.)

If even more discrepancies are zero, then yet more shifting of ϕ_k is necessary before the procedure can go back to its normal operation. If ℓ extra zeros occur, then:

$$H_{(n,n+1)}[\phi_{k-1}, \phi_k, x\phi_k, \cdots, x^{\ell-1}\phi, x^{\ell}\phi_k, x^{\ell+1}\phi_k] \quad =$$

$$\left[\begin{array}{ccccccc}
0 & 0 & \cdots & 0 & 0 & 0 & 0 \\
0 & 0 & \cdots & 0 & 0 & 0 & 0 \\
\vdots & \vdots & & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & \cdots & 0 & 0 & 0 & 0 \\
r_{k-1,k-1} & 0 & \cdots & 0 & 0 & 0 & r_{k,k+\ell} \\
r_{k-1,k} & 0 & \cdots & 0 & 0 & r_{k,k+\ell} & \vdots \\
\vdots & \vdots & & \vdots & \iddots & \vdots & \\
 & 0 & \cdots & r_{k,k+\ell} & & & \\
 & r_{k,k+\ell} & \iddots & \vdots & & & \\
 & r_{k,k+\ell+1} & & & & & \\
 & \vdots & & & & &
\end{array}\right].$$

A replacement for $\phi_{k+\ell+1}$ is determined by zeroing $\ell+2$ places of the discrepancies of $\phi_{k+\ell+1}$ with the preceding vectors, obtaining $\phi'_{k+\ell+1}$. Alternatively, the column corresponding to ϕ_{k-1} can be zeroed and the polynomial rescaled to be monic. Then $\phi'_{k+\ell+1} = (1/\gamma_{\ell+1})(\phi_{k-1} - \sum_{j=0}^{l+1} \gamma_j x^j \phi_k)$, where

$$\left[\begin{array}{cccccc}
 & & & & & r_{k,k+\ell} \\
 & & & & r_{k,k+\ell} & r_{k,k+\ell+1} \\
 & & & \iddots & r_{k,k+\ell+1} & \vdots \\
 & & r_{k,k+\ell} & \cdots & \vdots & \\
 & r_{k,k+\ell} & r_{k,k+\ell+1} & \cdots & & \\
r_{k,k+\ell} & r_{k,k+\ell+1} & r_{k,k+\ell+1} & \cdots & r_{k,k+2\ell} & r_{k,k+2\ell+1}
\end{array}\right]
\left[\begin{array}{c} \gamma_0 \\ \gamma_1 \\ \gamma_2 \\ \vdots \\ \gamma_\ell \\ \gamma_{\ell+1} \end{array}\right]
=
\left[\begin{array}{c} r_{k-1,k-1} \\ r_{k-1,k} \\ r_{k-1,k+1} \\ \vdots \\ r_{k-1,k+\ell-1} \\ r_{k-1,k+\ell} \end{array}\right],$$

or

$$\begin{bmatrix} r_{k,k+\ell} & & & \\ r_{k,k+\ell+1} & r_{k,k+\ell} & & \\ \vdots & r_{k,k+\ell+1} & & \\ & \vdots & & \\ r_{k,k+2\ell+1} & r_{k,k+2\ell} & \cdots & r_{k,k+\ell} \end{bmatrix} \begin{bmatrix} \gamma_{\ell+1} \\ \gamma_\ell \\ \vdots \\ \gamma_1 \\ \gamma_0 \end{bmatrix} = \begin{bmatrix} r_{k-1,k-1} \\ r_{k-1,k} \\ \vdots \\ r_{k-1,k+\ell-1} \\ r_{k-1,k+\ell} \end{bmatrix},$$

which is easily solved by forward substitution.

From this development, the Lanczos process can be augmented by lookahead, so long as some nonzero element occurs in the vector of discrepancies corresponding to ϕ_k. It is possible, of course, that all of the discrepancies in the current column are zero, in which case the lookahead cannot proceed. Such an occurrence has been called an incurable breakdown.

First extend the matrix $H_{(n,n+1)}$ by adding rows, preserving the Hankel structure, and extending the remainder with $*$'s. Then

$$H_{(2n,n+1)} = \begin{bmatrix} \mu_0 & \mu_1 & \mu_2 & \cdots & \mu_{n-1} & \mu_n \\ \mu_1 & & \cdots & & & \mu_{n+1} \\ \vdots & & & & & \vdots \\ \mu_{n-1} & & \cdots & & \mu_{2n-2} & \mu_{2n-1} \\ \mu_n & & \cdots & & \mu_{2n-1} & * \\ \vdots & & & \iddots & \iddots & \vdots \\ \mu_{2n-1} & * & & & \cdots & * \\ * & & & & \cdots & * \end{bmatrix}.$$

If a nonzero valid discrepancy (that is, a discrepancy from the extended matrix not involving $*$) can be found here, then proceed with lookahead as before. Otherwise, it is apparent that a solution to the Berlekamp-Massey equation for the entire sequence has been found. Therefore, lookahead can halt after calculating $r_{i,j}$ for $i + j = 2n - 1$. We have shown:

Theorem 3.2.2 *Incurable breakdown of the Lanczos process implies that a factor of the minimal polynomial has been found.*

This is a very simple demonstration related to the major result of Taylor's thesis (see [27]), the so-called Mismatch Theorem, which he proves using the Jordan form, which is not guaranteed to exist for finite fields (although the rational canonical form exists [12]).

We have shown that when an incurable breakdown occurs, a factor of the minimal polynomial of the sequence $\{u_0, Au_0, A^2u_0, \cdots\}$ is obtained. Since a factor of this polynomial is the goal of one pass of the Wiedemann algorithm, then as a corollary, if u_0 is selected to be b, the Lanczos procedure has the option of reverting to the Wiedemann algorithm on the occurrence of incurable breakdown (so the breakdown is not really incurable). Such a procedure is examined in more detail when linear solution with the Lanczos method is examined later in this chapter.

Note that it is not necessary to preserve the polynomials during the algorithm, but rather merely to store the recursion parameters (α, β, and γ_i), perhaps on secondary store, and to construct polynomials only in the case of incurable breakdown; otherwise, it is possible to proceed simply with vectors (the lookahead can also be made without using the polynomials, as will be shown). The vector process itself will be examined later.

In this section, we have developed a polynomial lookahead version of the Lanczos algorithm, and have shown that when incurable breakdowns occur a factor of the minimal polynomial is obtained. In the next section, the method will be modified so that it does not require the powers of x (and therefore A and A^T) that are implicit in the matrix $H_{(n,n+1)}$ occurring above.

3.3 The Lookahead Lanczos Algorithm

Until now, the quantities $[x^i\phi_j] = v_0^T x^i \phi_j(x)(A)u_0$ have been used in developing the polynomial Lanczos algorithm and a lookahead version.

It is possible to formulate the lookahead Lanczos process in terms of the monomials x^j taken with ϕ_i in $[x^j\phi^i]$ without having to maintain the powers A^j and $(A^T)^j$ associated with x^j (and we have developed them). This uses coefficients of the polynomials ϕ_i, which is unattractive if we wish avoid computing this polynomial, so the method is not included here.

Instead, motivated by the bi-orthogonal process, we will endeavour to use only the already available (to the algorithm) quantities ϕ_k instead of x^k. Previously we have used the lower triangular matrix of discrepancies $L_{i,j} = [x^i\phi_j]$; now we will employ the matrix with elements $[\phi_i\phi_j]$. When lookahead is not required, since ϕ_j is orthogonal to all polynomials having lower degree, then this is equivalent. The case of lookahead is examined in this section.

Once the $(k-1)$-st iteration is complete, the lookahead degree is calculated by determining the first nonzero $r_{k,k+\ell} = [x^{k+\ell}\phi_k]$. By orthogonality, $[x^k\phi_k] = [\phi_k\phi_k]$. Additionally, if $[\phi_k\phi_k] = 0$, then ϕ_k is orthogonal to polynomials up to and including k; thus $[x^{k+1}\phi_k] = [\phi_k x\phi_k]$. Preconditioned on $[\phi_k x^j\phi_k] = 0$, then $[x^{k+j+1}\phi_k] = [\phi_k x^{j+1}\phi]$. Therefore, the lookahead degree can be found by calculating $[\phi_k x^j\phi_k]$ for $j = 0, 1, \cdots, \ell$. Since these values are equivalent to $[x^{k+j+1}\phi_k]$, the result from the mismatch theorem still holds, and incurable breakdown can be redressed by the Wiedemann algorithm.

Motivated by this calculation, $[\phi_k, x\phi_k, x^2\phi_k, \cdots, x^{k+\ell+1}\phi_k]$ will be (skew) orthogonalized with respect to $[\phi_{k-1}, \phi_k, x\phi_k, \cdots, x^\ell\phi_k]$.

Initially the lookahead block (indexed on the edges by the relevant polynomials) is:

	ϕ_{k-1}	ϕ_k	$x\phi_k$	$\cdots$	$x^\ell\phi_k$	$x^{\ell+1}\phi_k$
ϕ_{k-1}	$\times$	0	0	$\cdots$	0	$\times$
ϕ_k	$\otimes$	0	$\cdots$	0	$\times$	$\otimes$
$\vdots$	$\otimes$	$\vdots$	$\iddots$	$\iddots$	$\iddots$	$\vdots$
$x^{\ell-1}\phi_k$	$\otimes$	0	$\times$	$\otimes$	$\cdots$	$\otimes$
$x^\ell\phi_k$	$\otimes$	$\times$	$\otimes$	$\cdots$	$\otimes$	$\otimes$
$x^{\ell+1}\phi_k$	$\times$	$\otimes$	$\cdots$	$\otimes$	$\otimes$	$\otimes$

.

Here $\times$ is a nonzero, $\otimes$ is a possible nonzero, and the portion of the matrix in question is found inside the double lines. Since the polynomials ϕ_{k-1} and below, along with $\phi_k, x\phi_k, \cdots, x^{\ell-1}\phi_k$, form a basis for all polynomials of degree less than ℓ, then $[\phi_k x^j\phi_{k-1}] = 0$ for $j = 0, 1, \cdots, \ell-1$. This matrix is of Hankel form, having constant cross-diagonals.

If we define $\phi_{k+1} = x\phi_k - \frac{[\phi_k x^{\ell+1}\phi_k]}{[\phi_k x^\ell\phi_k]}\phi_k$, then replacing $x\phi_k$ with ϕ_{k+1}, the column corresponding to $x\phi_k$ becomes zero in the row corresponding to $x^\ell\phi_k$. Also, $x\phi_{k+1}$ will create a column shifted one up from the column corresponding to ϕ_{k+1}. This allows a recursive definition of ϕ_{k+j+1}, which creates a skew-diagonal block: $\phi_{k+j+1} = x\phi_{k+j} - \frac{[\phi_k x^{\ell+1}\phi_{k+j}]}{[\phi_k x^\ell\phi_k]}\phi_k$, where $j = 0, 1, \cdots, \ell$. Here the shifting property of this block has been used to replace $[\phi_k x^\ell\phi_j]$ with $[\phi_k x^{\ell+1}\phi_{j-1}]$. One technicality is the $[\phi_k x^{\ell+1}\phi_{k-1}]$ element in the last column, but since ϕ_k is orthogonal to all polynomials of degree less than $k+\ell$, then $[\phi_k x^{\ell+1}\phi_{k-1}] = [\phi_k x^\ell\phi_k]$, since ϕ_k and ϕ_{k-1} are monic.

Also note that the elements along the skew-diagonal are not disturbed in the skew-diagonalization, so they are all $[\phi_k x^\ell\phi_k] = [x^{k+\ell}\phi_k]$, including the lower-left element, $[\phi_k x^{\ell+1}\phi_{k-1}]$.

Now the row corresponding to $x^{\ell+1}\phi_k$ is to be skew-diagonalized. Let the polynomials updated for this row be labelled with ψ, and begin with $\psi_k = x^{\ell+1}\phi_k$. To zero the element under ϕ_k, $\psi_{k+1} = \psi_k - \frac{[\phi_k x^{\ell+1}\phi_k]}{[\phi_k x^{\ell}\phi_k]} x^{\ell}\phi_k$. Similarly, $\psi_{k+j+1} = \psi_{k+j} - \frac{[\phi_k x^{\ell+1}\phi_{k+j}]}{[\phi_k x^{\ell}\phi_k]} x^{\ell-j}\phi_k$.

But these constants are identical to those used to define ϕ_{k+j}, and examining the components of $\phi_{k+\ell+1}$ and $\psi_{k+\ell+1}$, we observe that they are identical (although, as a polynomial, ϕ_{k+j} has been accumulated in Horner form; that is, alternatively scaled by x and an element added).

To obtain the next vector, $\phi'_{k+\ell+1}$, a replacement for $\phi_{k+\ell+1}$, the element $[\phi_{k-1}\phi_{k+\ell+1}]$ must be zeroed, and similarly for the next vector, $\psi'_{k+\ell+1}$. Preserving monic polynomials, $\phi'_{k+\ell+1} = \phi_{k+\ell+1} - \frac{[\phi_k x^{\ell}\phi_k]}{[\phi_{k-1}\phi_{k-1}]}\phi_{k-1}$. Using the fact that the diagonal is constant, $\psi'_{k+\ell+1} = \psi_{k+\ell+1} - \frac{[\phi_k x^{\ell}\phi_k]}{[\phi_{k-1}\phi_{k-1}]}\psi_{k-1}$ as well.

Therefore, even though different polynomials are used inside the lookahead block in this method (different polynomials on the horizontal and vertical axis), they are unified into the same polynomial at the end of the block. It is also possible to orthogonalize first vertically, then horizontally, but this makes no difference.

To continue with the algorithm after this block is complete, compute $[\phi'_{k+\ell+1}\phi'_{k+\ell+1}]$. If this is not zero, then $\phi_{k+\ell+2}$ is built from $x\phi'_{k+\ell+1}$, $\phi'_{k+\ell+1}$ and ϕ_k. If $[\phi'_{k+\ell+1}\phi'_{k+\ell+1}]$ is zero, then lookahead proceeds once again, with the places of ϕ_{k-1} and ϕ_k taken by ϕ_k and $\phi'_{k+\ell+1}$, respectively.

The method for dealing with one lookahead block has been described. Now the normal course of this method from the beginning is examined. First the initialization stage is examined.

Let the transformed matrix just described be denoted D. If the first element of D, $[\phi_0\phi_0]$, is nonzero, then scale ϕ_0 to $\phi_1 = x\phi_0$ as a preliminary ϕ_1; then defining $\phi'_1 = x\phi_0 - \frac{[\phi_0 x\phi_0]}{[\phi_0\phi_0]}\phi_0$ zeros the off-diagonal:

$$\begin{array}{c|cc|}
 & \phi_0 & x\phi_0 \\ \hline
\phi_0 & \times & \otimes \\
x\phi_0 & \otimes & \otimes
\end{array}
\quad \Rightarrow \quad
\begin{array}{c|cc|}
 & \phi_0 & \phi'_1 \\ \hline
\phi_0 & \times & \\
\phi'_1 & & \otimes
\end{array}.$$

Here (and in the remainder of the chapter) unmarked locations are assumed to be zero.

If the first element of D, $[\phi_0\phi_0]$, is zero, then it is possible to shift the first nonzero up, as described in the lookahead block above, until the nonzero is in the first location, then to shift once more to produce a vector to be cancelled (as in the initial stage of the non-lookahead method):

$$\begin{array}{c|cccc|}
 & \phi_0 & \cdots & \phi_\ell & x\phi_\ell \\
\hline
\phi_0 & & & \times & \otimes \\
\vdots & & \iddots & & \\
x^\ell\phi_0 & \times & & &
\end{array}.$$

If a nonzero in the first column does not exist, then only a constant factor (unity) of the minimal polynomial is produced, and for this reason it is desired that the first element of D, $[\phi_0\phi_0]$, is nonzero (or that at least an initial lookahead block can be produced). If the second block is incurable, then at least a linear factor is generated.

If v_0 and u_0 can be chosen freely, it is simple to arrange that the first element, $[\phi_0\phi_0]$, be nonzero. For the orthogonal case $[\phi_0\phi_0] = v_0^T u_0$, therefore suitable u_0 and v_0 can be selected with nonzero matching coordinates. For the conjugate case $[\phi_0\phi_0] = v_0^T A u_0$, and then whenever A is not zero at every coordinate (which case can be excluded) suitable u_0 and v_0 can be selected to target exactly such a nonzero location, or can be selected from combinations of such vectors not adding to zero.

For the orthogonal Lanczos (and later conjugate gradient) algorithm, complete freedom in initial vectors is not desirable (since certain choices can reduce memory usage), and in order to connect these algorithms to the Wiedemann algorithm in the case of incurable breakdown, u_0 is set to b. Therefore, initial lookahead blocks may be required. A more detailed examination of initialization is presented after the orthogonal and conjugate Lanczos algorithms and the conjugate gradient algorithm have been presented in vector form.

Continuing on to the next polynomial, in normal operation $[\phi_1\phi_1] \neq 0$. Beginning the orthogonalization with ϕ_1,

$$\begin{array}{c|ccc|}
 & \phi_0 & \phi_1 & x\phi_1 \\
\hline
\phi_0 & \times & 0 & \times \\
\phi_1 & 0 & \times & \otimes \\
\hline
\end{array}.$$

Setting $\phi_2 = x\phi_1 - \frac{[\phi_1 x\phi_1]}{[\phi_1\phi_1]}\phi_1$, we can produce $\phi_2' = \phi_2 - \frac{[\phi_0\phi_2]}{[\phi_0\phi_0]}\phi_0$ (note $[\phi_0\phi_2] = [\phi_1\phi_1]$). From this,

$$\begin{array}{c|ccc|}
 & \phi_0 & \phi_1 & \phi_2' \\
\hline
\phi_0 & \times & 0 & 0 \\
\phi_1 & 0 & \times & 0 \\
\phi_2' & 0 & 0 & \times \\
\hline
\end{array} .$$

The normal procedure progresses in this manner. When unexpected zeros in ϕ' occur, the lookahead procedure described previously is employed.

In general then, D has a block-diagonal structure, where each block is skew-diagonal, these skew-diagonal elements being identical inside a block; thus D resembles:

$$\begin{bmatrix}
\times \\
 & & & \times \\
 & & \times \\
 & \times \\
 & & & & & & & & & \times \\
 & & & & & & & & \times \\
 & & & & & & & \times \\
 & & & & & & \times \\
 & & & & & \times \\
 & & & & \times \\
 & & & & & & & & & & & \times \\
 & & & & & & & & & & \times \\
 & & & & & & & & & & & & \ddots
\end{bmatrix} .$$

When finding the number of unforced zeros from ϕ', an incurable breakdown is possible. In this case, it has been shown above that a factor of the minimal polynomial is obtained, which is equivalent to one pass of the Wiedemann algorithm. If the first lookahead block is not incurable (where the normal case $\ell = 0$ is defined to be a lookahead block of degree 0), then this polynomial is not 1, the trivial monic factor.

In this section, a lookahead Lanczos method has been developed that does not require powers x^j to be used. A method to skew-diagonalize these lookahead blocks has been developed. This produces a matrix of block-diagonal form, where the blocks are skew-diagonal and each skew-diagonal is constant (and nonzero). The initialization of vectors u_0 and v_0 has also been briefly considered, and will be considered again after the lookahead conjugate gradient method has been developed. In the next section, the lookahead Lanczos algorithm will be described in its vector form.

3.4 The Lookahead Lanczos Algorithm in Terms of Vectors

A practical Lanczos algorithm will not keep polynomials alone, since the inner product $(\phi, \psi) = v_0^T \phi(A)\psi(A)u_0$ would require many multiplications of A to evaluate. Instead, vectors $u_j = \phi_j(A)u_0$ and $v_j = \phi_j(A^T)v_0$ are maintained.

If a breakdown occurs, it can be assumed that vectors v_{k-1}, u_{k-1} and v_k, u_k have been calculated, and further that $\delta_{k-1} \neq 0$ and $\delta_k = 0$. The vector calculations required for the lookahead will now be described.

To calculate the degree of the lookahead, $[\phi_k x^j \phi_k]$ for $j = 1, 2, \cdots, \ell$ is required. Let $v_{k+j} = v_0^T \{x^j \phi_k\}(A)$ for $j = 1, 2, \cdots, \ell, \ell+1$, then $v_{k+j+1} = A^T v_{k+j}$. The vector inner product version of $[\phi_k x^j \phi] = (v_{k+j}, u_k)$ is used to determine the lookahead degree (or to discover an incurable breakdown).

Next, to skew-diagonalize the block, the quantity $[\phi_k x^{\ell+1} \phi_{k+j}] = (v_{k+\ell+1}, u_{k+j})$ is required, along with $[\phi_k x^\ell \phi_k] = (v_{k+\ell}, u_k)$, already determined.

Let $\alpha_{k+j} = \frac{[\phi_k x^{\ell+1} \phi_{k+j}]}{[\phi_k x^\ell \phi_k]}$, then

$$\begin{aligned} \phi_{k+j+1} &= x\phi_{k+j} - \alpha_{k+j}\phi_k \\ u_{k+j+1} &= Au_{k+j} - \alpha_{k+j}u_k \end{aligned}$$

for $j = 0, 1, \cdots, \ell$.

Next, the vector $v_{k+\ell+1}$ must be orthogonalized (there is no need to maintain ψ, since it will equal ϕ at the end of the block). This requires v_{k+j} for $j = 0, 1, \cdots, \ell$, which (unless retained) must be recalculated. We will see that linear solution inside lookahead blocks requires a similar second pass (or storage), and that the two can be combined. Let w be used as the successively orthogonalized vector. Then, setting $w_k = v_{k+\ell+1}$,

$$w_{k+j+1} = w_{k+j} - \alpha_{k+\ell-j} v_{k+j},$$

for $j = 0, 1, \cdots, \ell$.

Now the lookahead polynomial ϕ' and its associated vectors must be found. Let $\beta = \frac{[\phi_k x^\ell \phi_k]}{[\phi_{k-1} \phi_{k-1}]}$, then

$$\begin{aligned} \phi'_{k+\ell+1} &= \phi_{k+\ell+1} - \beta\phi_{k-1} \\ u'_{k+\ell+1} &= u_{k+\ell+1} - \beta u_{k-1} \\ v'_{k+\ell+1} &= w_{k+\ell+1} - \beta v_{k-1}. \end{aligned}$$

The algorithm can be cast in terms of ϕ_{prev}, ϕ_{curr} and ϕ_{next} corresponding to ϕ_{k-1}, ϕ_k and ϕ_{k+j}, along with similar versions of the associated vectors u and v, and scalars. During lookahead only, an additional vector w and storage for α_j (a vector as long as the lookahead block) are required.

In algorithm form:

○ ALGORITHM 3.4.1 *Polynomial Lookahead*

$\phi_{\text{prev}} := \phi_{k-1}$
$u_{\text{prev}} := u_{k-1}$
$v_{\text{prev}} := v_{k-1}$
$\phi_{\text{curr}} := \phi_k$
$u_{\text{curr}} := u_k$
$v_{\text{curr}} := v_k$
$\delta_{\text{prev}} = [\phi_{\text{prev}}\phi_{\text{prev}}] \neq 0$ *known, and* $\delta_{\text{curr}} = [\phi_{\text{curr}}\phi_{\text{curr}}] = 0$
$\ell := 0$
$v_{\text{next}} := v_{\text{curr}}$
do {
    $v_{\text{next}} := A^T v_{\text{next}}$
    $\ell := \ell + 1$
    if $2k + \ell \geq 2n$ *then STOP* factor
} *while* $(v_{\text{next}}, u_{\text{curr}}) = 0$

$\delta_{\text{curr}} := [\phi_{\text{curr}} x^\ell \phi_{\text{curr}}] := (v_{\text{next}}, u_{\text{curr}})$
$v_{\text{next}} := A^T v_{\text{next}}$

$\phi_{\text{next}} := \phi_{\text{curr}}$
$u_{\text{next}} := u_{\text{curr}}$

for j $:= 0$ *to* ℓ *do* {
    $\alpha_j := (v_{\text{next}}, u_{\text{next}})/\delta_{\text{curr}}$
    $\phi_{\text{next}} := x\phi_{\text{next}} - \alpha_j \phi_{\text{curr}}$

$$u_{\text{next}} := Au_{\text{next}} - \alpha_j u_{\text{curr}}$$

}

$w := v_{\text{next}}$

$v_{\text{next}} := v_{\text{curr}}$

for $j := 0$ *to* ℓ *do* {

$$w := w - \alpha_{\ell-j}\, v_{\text{next}}$$

$$v_{\text{next}} := A^T v_{\text{next}}$$

}

$\beta := \delta_{\text{curr}}/\delta_{\text{prev}}$

$\phi_{\text{next}} := \phi_{\text{next}} - \beta\phi_{\text{prev}}$

$u_{\text{next}} := u_{\text{next}} - \beta u_{\text{prev}}$

$v_{\text{next}} := w - \beta v_{\text{prev}}$

$\phi_{\text{prev}} := \phi_{\text{curr}}$

$u_{\text{prev}} := u_{\text{curr}}$

$v_{\text{prev}} := v_{\text{curr}}$

$\phi_{\text{curr}} := \phi_{\text{next}}$

$u_{\text{curr}} := u_{\text{next}}$

$v_{\text{curr}} := v_{\text{next}}$

○

Note that it is not necessary to update the polynomial ϕ, but rather to store the coefficients α_i and β, from which ϕ can be updated if it is needed for an incurable breakdown.

Also note that the method presented calculates the degree of the lookahead without the need for a block of vectors, but instead the entire lookahead procedure can be made with only a constant number of vectors, regardless of the size of the block.

Now that a practical lookahead Lanczos method has been developed, it can be applied to linear system solution. The orthogonal and conjugate Lanczos algorithms differ in this application, and the next section contains a subsection on each.

3.5 Linear System Solution via the Lookahead Lanczos Method

The goal in applying the Lanczos procedure to the matrix A is to solve the linear system $Az = b$. The orthogonal and conjugate versions differ in linear system solution application. Both calculate matrices U,V such that V^TAU is narrow-banded, but the conjugate method immediately produces a block-diagonal matrix, which is easier to apply, and is described first.

3.5.1 Conjugate Lanczos Linear System Solution

Lanczos algorithms are useful in linear system solution because they provide a transformation of the matrix A into simple form. Assembling $U = [u_0, u_1, \cdots]$ and $V = [v_0, v_1, \cdots]$, as built by the conjugate method just described in the previous section, then $V^TAU = D$, where D is block-diagonal, the blocks are skew-diagonal, and these diagonal elements are identical within a block and nonzero. This follows from the fact that the matrix built by the conjugate method has (i, j)-th element $v_i^T A u_j$.

To solve the linear system $Az = b$, assume that z has the representation $z = Uy$, and then $AUy = b$, $V^TAUy = V^Tb$, and $Dy = V^Tb$. Solving $Dy = V^Tb$ along with $z = Uy$ determines z.

How can it be ensured that z has the representation $z = Uy$? First it is necessary to study how the Lanczos process has terminated. As the algorithm has been described, termination occurs only for incurable breakdown; that is, when a factor of the minimal polynomial has been discovered, which occurs when $[\phi_k x^\ell \phi_k] = 0$ for k and ℓ sufficiently large.

For linear system solution, the next step at this point depends upon u_k and v_k. Since the Lanczos procedure is an orthogonalization, if it completes to the rank of the Krylov subspace of A and u_0, r, then $u_{r+1} = 0$, where r is the rank. Similarly, if it completes to the rank of the Krylov subspace of A^T and v_0, then $v_{r+1} = 0$. It is also possible that the procedure has not completed to rank, in which case perhaps only one of u_k or v_k is zero, or neither are.

If $u_k = 0$, it is desired that z has the representation $z = Uy$. If the Krylov subspace of A and u_0 has full rank, and if the process completes to this rank, then U spans the whole space, and z has such a representation.

Otherwise, there is another advantage to choosing $u_0 = b$ (in addition to the connection with the Wiedemann algorithm). Since $u_k = 0$, then $\phi_k(A)u_0 = \phi_k(A)b = 0$. If ϕ_k

has a nonzero constant coefficient $\phi_{k,0}$, then re-normalizing, define $\phi'_h = \phi_k/\phi_{k,0}$. Therefore, $\sum_{i=1}^{k} \phi'_{k,i} A^i b + b = 0$ and $-A \sum_{i=1}^{k} \phi'_{k,i} A^{i-1} b = b$. Thus $z = -\sum_{i=1}^{k} \phi'_{k,i} A^{i-1} b$, as in the Wiedemann algorithm. It is not necessary to go back and compute z in this way, however, since this also implies that $z \in K_k(A, b)$.

Since U spans $K_k(A, b)$, then z can be represented as $z = Uy$. The solution z is computed from the Lanczos vectors in a manner that will be described after the various termination cases have been explored.

If $\phi_{k,0} = 0$, then x divides $\phi_k(x)$, or $A\{\frac{\phi_k}{x}\}(A)u_0 = 0$. Therefore, $\{\frac{\phi_k}{x}\}(A)u_0$ is a null vector of A, implying that a column dependency has been found. This cannot be a trivial null vector, since that would contradict the minimality of the minimal polynomial of the sequence $\{\mu_0, \mu_1, \cdots\}$, which is ϕ_k.

In the same way, if $v_k = 0$ and $\phi_{k,0} = 0$, then $A^T\{\frac{\phi_k}{x}\}(A^T)v_0 = 0$, and $\{\frac{\phi_k}{x}\}(A^T)v_0$ is a non-trivial row dependency of A.

Thus, either a row or column can be deleted from A, which is known to be singular at this point. The resulting matrix A' is no longer square. As in the Wiedemann algorithm for non-square matrices, let an excess row or column be b', transposing if necessary, and deleting this vector to obtain a square A''. Then attempt to solve for z' in $A''z' = b'$, which will result either in a replacement for A which is square, or in the discovery of still more dependencies. Apply this recursively. This will either find a solution or determine that a solution does not exist (but only probabilistically, since the Wiedemann algorithm is probabilistic).

If $v_k = 0$ and $\phi_{k,0} \neq 0$, then no row dependency is obtained. One possibility is to set v_0 to a row (or a combination of rows) from A. Using $\phi'_k = \phi_k/\phi_{k,0}$, then $A(-\sum_{i=1}^{k} \phi'_{k,i}(A^T)^{i-1}v_0) = v_0$, yielding a row dependency. It is possible that the dependency vector in this case is exactly the vector used to initialize v_0, however unlikely that may be. In that case, the only information obtained is a factor of the minimal polynomial, equivalent to one pass of the Wiedemann algorithm.

The only case not considered is when both u_k and v_k are not zero, and then an equivalent single pass of the Wiedemann algorithm is obtained again. As a practical suggestion, at this point u_k and v_k may be checked themselves for being right or left null vectors, respectively, of A. This need not be the case, but would seem to be the most likely alternative (the other alternative is that non-null vectors all produce zero discrepancies up to the incurable bound, which heuristically seems unlikely, especially early in the process).

A solution might be found without further effort (calculating ϕ) in any of these cases, and in order to avoid calculating $\phi_{k,0}$ unnecessarily, an attempt should be made to find z via $z = Uy$. If a solution is not forthcoming, then proceed as described above for each possible case.

The potential solution is obtained via $z = Uy$ and $Dy = V^T b$. The vector z is accumulated as the algorithm progresses. For each u_i and v_i not from a lookahead block, $(v_i, u_i)\, y_i = v_i^T b$, determining y_i, from which z can be incremented by $y_i u_i$. It cannot be that $(v_i, u_i) = 0$, because in that case lookahead would have been required.

Now consider a lookahead block:

$$\begin{bmatrix} 0 & 0 & \cdots & 0 & \gamma \\ 0 & & & \gamma & 0 \\ \vdots & & \iddots & & \vdots \\ 0 & \gamma & & & 0 \\ \gamma & 0 & \cdots & 0 & 0 \end{bmatrix} \begin{bmatrix} y_k \\ y_{k+1} \\ \vdots \\ y_{k+\ell-1} \\ y_{k+\ell} \end{bmatrix} = \begin{bmatrix} v_k^T \\ v_{k+1}^T \\ \vdots \\ v_{k+\ell-1}^T \\ v_{k+\ell}^T \end{bmatrix} b,$$

and

$$z_{\text{incr}} = [u_k, u_{k+1}, \cdots, u_{k+\ell-1}, u_{k+\ell}] \begin{bmatrix} y_k \\ y_{k+1} \\ \vdots \\ y_{k+\ell-1} \\ y_{k+\ell} \end{bmatrix},$$

where z_{incr} is the increment to the solution z.

As the Lanczos algorithm progresses, $[y_k, y_{k+1}, \cdots, y_{k+\ell}]$ are obtained in reverse order. Thus, it is necessary to store the vectors u_j, $j = k, k+1, \cdots, k+\ell$, or to recompute them after obtaining all the y_j. If the block is very small, then storage is possible; otherwise, the calculation of u_j can easily be repeated. In fact, it can be combined with the skew-orthogonalization, where a similar choice of store or re-calculate is encountered.

Linear system solution, and the desired connection with the Wiedemann algorithm, puts some constraints on u_0 and v_0 then, and previously it has been assumed that they could be chosen freely to ensure a nonzero $[\phi_0 \phi_0]$. It is possible with random u_0 and v_0 that the algorithm may complete with u_k zero and U spanning the whole space, or that it may finish early with u_k zero and z representable as Uy. However, it seems best to try to make $[\phi_0 \phi_0]$ nonzero using the freedom of choosing v_0 to be a combination of the rows of A, and only when this is not possible to choose v_0 freely.

If such a constrained v_0 cannot be found for $u_0 = b$, then the right-hand side b can be shifted to $b' = b - As$ with random shift vector s. After successful solution, the solution is shifted back: $z = z' + s$.

For the symmetric case, an one-half saving of storage results from using $u_0 = v_0$, so the choice of different u_0 and v_0 is unattractive. Then only the freedom of shifting b is available, if $u_0 = b$ is required.

If this proves insufficient to make $[\phi_0\phi_0]$ nonzero, or to find an initial lookahead block, then choose v_0 freely as before, and if the favourable outcome of completing to rank, or of having $u_k = 0$, or $v_k = 0$ and $\phi_{k,0} = 0$, does not occur, then as ever, one pass of the Wiedemann algorithm has been completed, with a factor of the minimal polynomial.

In no case, then, can the algorithm be said to have failed completely, at least for linear system solution.

We are now ready to specify the linear system solution method in the form of an algorithm. For the conjugate method, linear system solution can be made with the following algorithm:

○ ALGORITHM 3.5.1 *Conjugate Lanczos Linear System Solution*

INPUT: A, b, u_{curr}, v_{curr}

$\delta_{\mathrm{curr}} := (v_{\mathrm{curr}}, u_{\mathrm{curr}})$
if $\delta_{\mathrm{curr}} = 0$ *then 'form initial lookahead block'*
$\phi_{\mathrm{curr}} := 1$ *;* $i := 0$ *;* $z := 0$ *;* $\beta := 0$ *; Incurable* $:=$ *false*

while $i \leq n$ *do* $\{$
    $\delta_{\mathrm{curr}} := (v_{\mathrm{curr}}, u_{\mathrm{curr}})$
    if $\delta_{\mathrm{curr}} \neq 0$ *then* $\{$ (regular)
        $y := v_{\mathrm{curr}}^T b / \delta_{\mathrm{curr}}$
        $z := z + y u_{\mathrm{curr}}$
        $\alpha := (v_{\mathrm{curr}}, A u_{\mathrm{curr}}) / \delta_{\mathrm{curr}}$

        if $i \neq 0$ *then* $\beta := \delta_{\mathrm{curr}} / \delta_{\mathrm{prev}}$
        else $\beta := 0$

        $\phi_{\mathrm{next}} := (x - \alpha)\phi_{\mathrm{curr}} - \beta \phi_{\mathrm{prev}}$
        $u_{\mathrm{next}} := A u_{\mathrm{curr}} - \alpha u_{\mathrm{curr}} - \beta u_{\mathrm{prev}}$
        $v_{\mathrm{next}} := A^T v_{\mathrm{curr}} - \alpha v_{\mathrm{curr}} - \beta v_{\mathrm{prev}}$

$i \ := \ i+1$

$\}$ *else* $\{$ (lookahead)

if $u_{\rm curr} = 0$ *then BREAK* '$u_{\rm curr} = 0$'

if $v_{\rm curr} = 0$ *then BREAK* '$v_{\rm curr} = 0$'

$\ell \ := \ 0$; $v_{\rm next} \ := \ v_{\rm curr}$; $y_\ell \ := \ v_{\rm next}^T b$

while $(v_{\rm next}, u_{\rm curr}) = 0$ *do* $\{$ (find lookahead degree)

$v_{\rm next} \ := \ A^T v_{\rm next}$

$\ell \ := \ \ell + 1$

$y_\ell := v_{\rm next}^T b$

if $(2i + \ell \geq 2n)$ *then Incurable* $:=$ *true*

$\}$

if Incurable then BREAK 'factor'

$\delta_{\rm curr} \ := \ (v_{\rm next}, u_{\rm curr})$

$v_{\rm next} \ := \ A^T v_{\rm next}$

$u_{\rm next} \ := \ u_{\rm curr}$

$\phi_{\rm next} \ := \ \phi_{\rm curr}$

for $j \ := \ 0$ *to* ℓ *do* $\{$ (determine u's)

$\alpha_j \ := \ (v_{\rm next}, u_{\rm next})/\delta_{\rm curr}$

$z \ := \ z + (y_{\ell-j}/\delta_{\rm curr}) u_{\rm next}$

$u_{\rm next} \ := \ A u_{\rm next} - \alpha_j u_{\rm curr}$

$\phi_{\rm next} \ := \ x \phi_{\rm next} - \alpha_j \phi_{\rm curr}$

$\}$

$w \ := \ v_{\rm next}$; $v_{\rm next} \ := \ v_{\rm curr}$

for $j \ := \ 0$ *to* ℓ *do* $\{$ (determine last v)

$w \ := \ w - \alpha_{\ell-j} v_{\rm next}$

$v_{\rm next} \ := \ A^T v_{\rm next}$

$\}$

$$\beta := \delta_{\text{curr}}/\delta_{\text{prev}}$$
$$\phi_{\text{next}} := \phi_{\text{next}} - \beta\phi_{\text{prev}}$$
$$u_{\text{next}} := u_{\text{next}} - \beta u_{\text{prev}}$$
$$v_{\text{next}} := w - \beta v_{\text{prev}}$$

$$i := i + \ell + 1$$

}

}

$\phi_{\text{prev}} := \phi_{\text{curr}}$; $u_{\text{prev}} := u_{\text{curr}}$; $v_{\text{prev}} := v_{\text{curr}}$; $\delta_{\text{prev}} := \delta_{\text{curr}}$

$\phi_{\text{curr}} := \phi_{\text{next}}$; $u_{\text{curr}} := u_{\text{next}}$; $v_{\text{curr}} := v_{\text{next}}$

if $Az - b = 0$ *then RETURN 'solution'* z

if $u_{\text{curr}} \neq 0$ *and* $Au_{\text{curr}} = 0$ *then RETURN 'column dependency'* u_{curr}

if $v_{\text{curr}} \neq 0$ *and* $v_{\text{curr}}^T A = 0$ *then RETURN 'row dependency'* v_{curr}

RETURN 'factor of minimal polynomial' ϕ_{curr}

○

Here, it is assumed that more than one quantity can be returned by the various RETURN statements, so it is possible that the algorithm may return a solution, a column dependency, a row dependency and a factor of the minimal polynomial, all at once.

An algorithm for linear solution via the conjugate Lanczos algorithm has been presented. This algorithm makes use of the novel and simple lookahead developed above in a polynomial context, and then expressed in terms of vectors. The transformed matrix from this lookahead matrix is particularly simple, since the lookahead blocks have been skew-diagonalized. The method uses three versions of each vector, the 'previous' the 'current' and the 'next'. If lookahead is not required, then previous and next versions of the vectors can share storage. In lookahead this is not possible, and another vector in addition (w in the algorithm) is required. As described above, the polynomial portion of the computation may be delayed until required if the recursion parameters (α, α_i and β) are stored at each iteration.

In the next section, linear solution with the orthogonal Lanczos algorithm is considered.

3.5.2 Orthogonal Lanczos Linear System Solution

The orthogonal algorithm is the Lanczos method usually presented by numerical analysts. The advantage of the orthogonal method, in this setting, is that if $u_0 = b$, then $V^T b$ reduces to a vector having only the first component nonzero, since b, as a column of U, is orthogonal to the other v's. Then b need not be retained in memory.

The difficulty with this method is that $V^T AU$ is no longer block-diagonal, as it was with the conjugate method. This complicates the method, and although the solution method will be described, it will not be presented in algorithm form, as was done above for the conjugate Lanczos algorithm.

The method of solution is the same for the orthogonal method, and all the material on termination given for the conjugate method in the last section applies equally well to the orthogonal method. The only difference lies in the solution of $V^T AU = V^T b$, or $By = V^T b$, where B is the transformed vector.

As with the conjugate method, the matrix with elements $D_{i,j} = [\phi_i \phi_j]$ has been constructed with a block-diagonal structure, having skew-diagonal blocks, and each block having nonzero elements identical within the block.

To transform A in this case, $B = [\phi_{\mathrm{row}} x \phi_{\mathrm{col}}]$ is required, since the x was not included in the orthogonalization. The element $[\phi_0 x \phi_0]$ is potentially nonzero, so the matrix B has a potential nonzero in the top-left corner.

If ϕ_k is not from a lookahead block, then $[\phi_k x \phi_k]$ is also potentially nonzero, and $[\phi_k x \phi_{k-1}] = [\phi_k \phi_k]$ is certainly nonzero, but $[\phi_k x \phi_j] = 0$ for $j < k - 1$.

Therefore, at k there is a small 'zig-zag' (nonzero skew-diagonal going down, then a possible horizontal going right) on the diagonal:

$$
\begin{array}{c|cccc}
 & & & \phi_{k-1} & \phi_k \\
\hline
 & & & & \vdots \\
 & & & & 0 \\
\phi_{k-1} & & & & \times \\
\phi_k & \cdots & 0 & \times & \otimes
\end{array}\;,
$$

where each $\times$ represents a nonzero and each $\otimes$ represents a possible nonzero. Each non-lookahead k introduces such a 2×2 zig-zag.

Suppose that $\phi_{k+\ell}$ is the last polynomial for a lookahead block having ℓ extra zeros, before this last row has been zeroed:

$$
\begin{array}{c|ccccc|}
 & \phi_{k-1} & \phi_k & \phi_{k+1} & \cdots & \phi_{k+\ell} \\
\hline
\phi_{k-1} & \times & & & & \\
\phi_k & & & & & \times \\
x\phi_k & & & & \iddots & \\
\vdots & & & & & \\
x^{\ell}\phi_k & & \times & & & \\
x^{\ell+1}\phi_k & \times & \otimes & & \cdots & \otimes \\
\hline
\end{array}.
$$

Adding an x to the inner product, $[\phi x \psi]$, simply shifts this block up, creating a longer diagonal and a row of possible nonzeros (equal to the α_j of algorithm (1), after scaling by δ_{curr}). The form of the matrix with this inner product resembles:

$$
\begin{bmatrix}
\otimes & & \times & & & & & & & & & \\
 & \times & & & & & & & & & & \\
\times & \otimes & \otimes & & & & & & & \times & & \\
 & & & & & & & & \times & & & \\
 & & & & & & & \times & & & & \\
 & & & & & & \times & & & & & \\
 & & & & & \times & & & & & & \\
 & & & & \times & & & & & & & \\
 & & & \times & & & & & & & & \\
 & & \times & \otimes & \otimes & \otimes & \otimes & \otimes & \otimes & \otimes & \times & \\
 & & & & & & & & & \times & \otimes & \\
 & & & & & & & & & & & \ddots
\end{bmatrix}.
$$

In normal (non-lookahead) operation, these stacked 2×2 zig-zags produce a tridiagonal matrix:

$$
\begin{bmatrix}
\times & \rightarrow & \times & & & & & \\
 & \swarrow & & & & & & \\
\times & \rightarrow & \otimes & \rightarrow & \times & & & \\
 & & & \swarrow & & & & \\
 & & \times & \rightarrow & \otimes & & \ddots & \\
 & & & \ddots & & \ddots & & \times \\
 & & & & & & \swarrow & \\
 & & & & & \times & \rightarrow & \otimes
\end{bmatrix}.
$$

A modification to the ordering of the calculations of algorithm 1 finds an intermediate v that removes the possible nonzero elements from the transformed matrix, making it symmetric (V^TAU becomes symmetric, although then the matrix V^TU becomes non-symmetric, but this is not needed in the linear solution). The idea is to delay zeroing $\ell+1$ components of $v_{k+\ell+1}$, but instead to zero ℓ components of $v_{k+\ell}$, then to use this vector to transform A to a symmetric form. Once this vector has been obtained, the last iteration of the loops of algorithm (1) is made outside of the loop, forcing $\ell+1$ zeros.

○ ALGORITHM 3.5.2 *Modified Orthogonal Lookahead*

$$\begin{aligned}
\phi_{\mathrm{prev}} &:= \phi_{k-1}\\
u_{\mathrm{prev}} &:= u_{k-1}\\
v_{\mathrm{prev}} &:= v_{k-1}\\
\phi_{\mathrm{curr}} &:= \phi_{k}\\
u_{\mathrm{curr}} &:= u_{k}\\
v_{\mathrm{curr}} &:= v_{k}
\end{aligned}$$

$(\delta_{\mathrm{prev}} = [\phi_{\mathrm{prev}}\phi_{\mathrm{prev}}] \neq 0$ *known, and* $\delta_{\mathrm{curr}} = [\phi_{\mathrm{curr}}\phi_{\mathrm{curr}}] = 0)$

$\ell := 0$

$v_{\mathrm{next}} := v_{\mathrm{curr}}$

do $\{$

$\quad v_{\mathrm{next}} := A^T v_{\mathrm{next}}$

$\quad \ell := \ell + 1$

$\quad$ *if* $2k+\ell \geq 2n$ *then STOP* factor

$\}$ *while* $(v_{\mathrm{next}}, u_{\mathrm{curr}}) = 0$

$$\delta_{\mathrm{curr}} := [\phi_{\mathrm{curr}} x^{\ell} \phi_{\mathrm{curr}}] := (v_{\mathrm{next}}, u_{\mathrm{curr}})$$

$$\begin{aligned}
\phi_{\mathrm{next}} &:= \phi_{\mathrm{curr}}\\
u_{\mathrm{next}} &:= u_{\mathrm{curr}}
\end{aligned}$$

for $j := 0$ *to* $\ell - 1$ *do* {
$u_{\text{next}} := Au_{\text{next}}$
$\alpha_j := (v_{\text{next}}, u_{\text{next}})/\delta_{\text{curr}}$
$\phi_{\text{next}} := x\phi_{\text{next}} - \alpha_j\phi_{\text{curr}}$
$u_{\text{next}} := u_{\text{next}} - \alpha_j u_{\text{curr}}$
}
$w := v_{\text{next}}$
$v_{\text{next}} := v_{\text{curr}}$
for $j := 0$ *to* $\ell - 1$ *do* {
$w := w - \alpha_{\ell-j-1}\, v_{\text{next}}$
$v_{\text{next}} := A^T v_{\text{next}}$
}
// Use w *instead of* v_{next} *for symmetric transformed matrix*
// Use v_{next} *to make this method identical to algorithm 3.4.1*

// force last zero, which was inside loop in algorithm 3.4.1:
$u_{\text{next}} := Au_{\text{next}}$
$\alpha := (w, u_{\text{next}})/\delta_{\text{curr}}$
$\phi_{\text{next}} := x\phi_{\text{next}} - \alpha\phi_{\text{curr}}$
$u_{\text{next}} := u_{\text{next}} - \alpha u_{\text{curr}}$
$v_{\text{next}} := A^T w - \alpha v_{\text{curr}}$

$\beta := \delta_{\text{curr}}/\delta_{\text{prev}}$

$\phi_{\text{next}} := \phi_{\text{next}} - \beta\phi_{\text{prev}}$
$u_{\text{next}} := u_{\text{next}} - \beta u_{\text{prev}}$
$v_{\text{next}} := v_{\text{next}} - \beta v_{\text{prev}}$

$\phi_{\text{prev}} := \phi_{\text{curr}}$
$u_{\text{prev}} := u_{\text{curr}}$
$v_{\text{prev}} := v_{\text{curr}}$

$\phi_{\text{curr}} := \phi_{\text{next}}$

$$\begin{aligned} u_{\mathrm{curr}} &:= u_{\mathrm{next}} \\ v_{\mathrm{curr}} &:= v_{\mathrm{next}} \end{aligned}$$

$\bigcirc$

The form of the matrix with this modification has a simpler form resembling:

$$\begin{bmatrix}
\otimes & & \times & & & & & & & & & \\
& \times & & & & & & & & & & \\
\times & & \otimes & & & & & & & \times & & \\
& & & & & & & & \times & & & \\
& & & & & & & \times & & & & \\
& & & & & & \times & & & & & \\
& & & & & \times & & & & & & \\
& & & & \times & & & & & & & \\
& & & \times & & & & & & & & \\
& & \times & & & & & & & \otimes & \times & \\
& & & & & & & & & \times & \otimes & \\
& & & & & & & & & & & \ddots
\end{bmatrix}.$$

The first diagonal element is equal to the first $\alpha\delta_{\mathrm{curr}}$ obtained, and thereafter the last diagonal corner of the block is equal to $\alpha\delta_{\mathrm{curr}}$, where α and δ_{curr} have been defined in algorithm 3.5.2.

This matrix will be referred to as a block-overlapping matrix because of the skew-diagonal blocks that overlap at the diagonal corners, at which points there are additional possible nonzero elements (whereas the previous less simple form might have been called a 'zig-zag' matrix). A method for solving a system $By = V^T b$ where B is a block-overlapping matrix is required. The symmetric tridiagonal form of the non-lookahead matrix suggests a band Cholesky, or a band LDL^T decomposition, as was used in the previous chapter. It has been discovered that elementary row and column operations are a more illuminating alternative.

First consider the narrowest case, the matrix generated with no lookahead, which is tridiagonal. If the first diagonal element is nonzero, then using it as a pivot, reduce horizontal and vertical adjacent elements with row and column operations:

$$\begin{bmatrix}
\times & \times & & \\
\times & \otimes & \times & \\
& \times & \otimes & \ddots \\
& & \ddots & \ddots
\end{bmatrix} \Rightarrow \begin{bmatrix}
\times & & & \\
& \otimes & \times & \\
& \times & \otimes & \ddots \\
& & \ddots & \ddots
\end{bmatrix}.$$

After this diagonalization, the same initial configuration appears one row and column advanced.

If the first diagonal element is zero, then using the first row and column, zero elements to decouple the first 2×2 block:

$$\begin{bmatrix} & \times & & & \\ \times & \otimes & \times & & \\ & \times & \otimes & \times & \\ & & \times & \otimes & \ddots \\ & & & \ddots & \ddots \end{bmatrix} \Rightarrow \begin{bmatrix} & \times & & & \\ \times & & & & \\ & & \otimes & \times & \\ & & \times & \otimes & \ddots \\ & & & \ddots & \ddots \end{bmatrix}.$$

Here two nonzero skew-diagonal elements are produced, and the shape is preserved (two rows and columns advanced). The termination requires examination. Applying the above procedure repeatedly, a 2×2 block remains in the lower-right corner. If the first diagonal element is nonzero, reduce the skew-diagonal:

$$\begin{bmatrix} \times & \times \\ \times & \otimes \end{bmatrix} \Rightarrow \begin{bmatrix} \times & \\ & \otimes \end{bmatrix}.$$

Otherwise, reduce the last diagonal element:

$$\begin{bmatrix} & \times \\ \times & \otimes \end{bmatrix} \Rightarrow \begin{bmatrix} & \times \\ \times & \end{bmatrix}.$$

It is apparent that B is non-singular, depending on the last element in the above reduction. The above operations generate new matrices U' and V', such that $(V')^T AU' = B'$ is diagonal and nonzero along the diagonal, except possibly the last element. Because elementary operation matrices have determinant ± 1, U' and V' retain their rank. Elementary row operations on B are column operations on V, and also column operations on B are column operations on U.

From this, $B'y = (V')^T b$ can be solved if B' is non-singular. If it is singular, one pass of the Wiedemann method has still been completed.

Now examine the possibility of lookahead blocks. If the first diagonal element of the block is nonzero, then the block is decoupled from the others by zeroing the skew-diagonal corners using the first row and column:

$$\begin{bmatrix} \times & & & & \times \\ & & & \times & \\ & & \iddots & & \\ & \times & & & \\ \times & & & & \otimes \end{bmatrix} \Rightarrow \begin{bmatrix} \times & & & & \\ & & & \times & \\ & & \iddots & & \\ & \times & & & \\ & & & & \otimes \end{bmatrix}.$$

Otherwise, if the first diagonal element of the block is zero, then decouple by zeroing the skew-diagonal corners of the next block, plus the last diagonal element of the block:

$$\begin{bmatrix} & & \times & & & & & \\ & \iddots & & & & & & \\ \times & & \otimes & & & & \times & \\ & & & & & \times & & \\ & & & & \iddots & & & \\ & & & \times & & & & \\ & & \times & & & & \otimes & \\ & & & & & & & \ddots \end{bmatrix} \Rightarrow \begin{bmatrix} & & \times & & & & & \\ & \iddots & & & & & & \\ \times & & & & & & & \\ & & & & & \times & & \\ & & & & \iddots & & & \\ & & & \times & & & & \\ & & & & & & \otimes & \\ & & & & & & & \ddots \end{bmatrix}.$$

Repeating this more generalized process, a B' of the same form as in the conjugate method is obtained, except possibly for the last diagonal element, which may be zero, all with elementary row and column operations. The system $(V')^T AU' = B'y$ can potentially be solved, depending on this last element. Linear solution is only attempted when the breakdown is incurable; therefore, if a solution cannot be completed because of the last element being zero, at least a factor of the minimal polynomial is available.

The advantage of the orthogonal method is the simplicity of the form of $V^T b$, if $u_0 = b$. The column b will now be mixed into V', but this can be kept track of, and b need not be retained.

Assume now that $u_0 = b$. Because the vectors interior to a block never experience reductions, they never have a multiple of b added to them, and thus the y's associated with them are all zero; therefore, these components add nothing to the solution update. For row reductions, only the multiple of the first row that is added need be kept, not the modified row, since this will be sufficient to determine the associated y. For column reductions, a separate vector is required. Therefore, the storage requirements of the orthogonal variation are the same as in the conjugate version (one less vector because b is not required, but one more because of the vector required to store the column reductions).

Since the conjugate method is simpler, the complete algorithm for linear system solving was given in the last section only for the conjugate method. An examination of the orthogonal case is justified because of its popularity in numerical analysis, and also because the orthogonal Lanczos procedure is included in the conjugate gradient method, which is the topic of the next section.

3.6 The Conjugate Gradient Method and the Lanczos Algorithm

To describe the conjugate gradient method in the same terms as the Lanczos algorithm, first consider coupled recursions for both orthogonal and conjugate methods. As before, define

$$H_{(n,n+1)} = \begin{bmatrix} \mu_0 & \mu_1 & \cdots & \mu_n \\ \mu_1 & \mu_2 & \cdots & \mu_{n+1} \\ \vdots & \vdots & \ddots & \vdots \\ \mu_{n-1} & \mu_n & \cdots & \mu_{2n-1} \end{bmatrix}. \tag{3.7}$$

Then, for the orthogonal Lanczos iteration:

$$H_{(n,n+1)} \begin{bmatrix} 1 & \phi_{1,0} & \phi_{2,0} & \cdots \\ & 1 & \phi_{2,1} & \cdots \\ & & 1 & \\ & & & \ddots \\ & & & \end{bmatrix} = \begin{bmatrix} r_{0,0} & & \\ r_{0,1} & r_{1,1} & \\ \vdots & \vdots & \ddots \\ r_{0,n-1} & r_{1,n-1} & \cdots \end{bmatrix}. \tag{3.8}$$

Using this same $H_{(n,n+1)}$ (instead of $H'_{(n,n+1)}$ used above), the conjugate recursion builds:

$$H_{(n,n+1)} \begin{bmatrix} 0 & 0 & 0 & \cdots \\ 1 & \psi_{1,0} & \psi_{2,0} & \cdots \\ & 1 & \psi_{2,1} & \cdots \\ & & 1 & \\ & & & \ddots \\ & & & \end{bmatrix} = \begin{bmatrix} s_{0,0} & & \\ s_{0,1} & s_{1,1} & \\ \vdots & \vdots & \ddots \\ s_{0,n-1} & s_{1,n-1} & \cdots \end{bmatrix}, \tag{3.9}$$

where the polynomials for the conjugate portion are ψ and the discrepancies are s.

The orthogonal recurrences are

$$\phi_{i+1}(x) = x\phi_i(x) - \alpha_i\phi_i(x) - \beta_i\phi_{i-1}(x),$$

and the conjugate recurrences are (with different α and β values)

$$\psi_{i+1}(x) = x\psi_i(x) - \alpha_i\psi_i(x) - \beta_i\psi_{i-1}(x).$$

It is now possible to couple the recursions, reducing them from trinomial recurrences to binomial ones, again using the shifting property of the Hankel matrix $H_{(n,n+1)}$, which is common to both equations (3.8) and (3.9). A polynomial ψ from the conjugate part may be scaled by x and installed in the orthogonal part, producing the same discrepancy vector. If a polynomial ϕ from the orthogonal part is installed in the conjugate part, then the resulting discrepancy vector is shifted up by one.

Starting with the orthogonal recurrence, ϕ_{i+1} can be built from the conjugate polynomial ψ_i

$$\phi_{i+1} = x\psi_i - \alpha_i\phi_i,$$

where now $\alpha_i = s_{i,i}/r_{i,i}$ (again, the scalars α_i and β_i are reused). Next, to build ψ_{i+1}, use ϕ_{i+1} and zero the discrepancy, yielding

$$\psi_{i+1} = \phi_{i+1} - \beta_i\psi_i,$$

where then $\beta_i = r_{i+1,i+1}/s_{i,i}$.

Therefore, if both orthogonal and conjugate recursions are required, it is possible to economize on the number of elements in the recursions (and therefore on storage) by coupling the two recursions together, using the same initial vectors.

Let vectors for the conjugate recursion be u and v; then $u_i = \phi_i(A)u_0$ and $v_i = \phi_i(A^T)v_0$. Similarly, let q and w be the orthogonal vectors, $q_i = \psi_i(A)q_0$ and $w_i = \psi_i(A^T)w_0$. Assembling these vectors,

$$U = [u_0, u_1, \cdots], V = [v_0, v_1, \cdots], Q = [q_0, q_1, \cdots], W = [w_0, w_1, \cdots].$$

Let $\delta = (w, q) = w^T q$ and $\sigma = (v, u) = v^T Au$. The binomial form allows the recursion to be made with two vectors for each of u, v, q and w:

○ ALGORITHM 3.6.1 *Coupled Orthogonal/Conjugate Lanczos*

$$\begin{aligned}
\sigma_{\text{prev}} &:= v_{\text{prev}}^T A u_{\text{prev}} \\
\alpha &:= \sigma_{\text{prev}}/\delta_{\text{prev}}
\end{aligned}$$

$$\begin{aligned}
\phi_{\text{curr}} &:= x\psi_{\text{prev}} - \alpha\phi_{\text{prev}} \\
q_{\text{curr}} &:= Au_{\text{prev}} - \alpha q_{\text{prev}} \\
w_{\text{curr}} &:= A^T v_{\text{prev}} - \alpha w_{\text{prev}}
\end{aligned}$$

$$\delta_{\rm curr} := w_{\rm curr}^T q_{\rm curr}$$
$$\beta := \delta_{\rm curr}/\sigma_{\rm prev}$$

$$\psi_{\rm curr} := \phi_{\rm curr} - \beta\psi_{\rm prev}$$
$$u_{\rm curr} := q_{\rm curr} - \beta u_{\rm prev}$$
$$v_{\rm curr} := w_{\rm curr} - \beta v_{\rm prev}$$

$$\phi_{\rm prev} := \phi_{\rm curr};\ \psi_{\rm prev} := \psi_{\rm curr}$$
$$u_{\rm prev} := u_{\rm curr};\ v_{\rm prev} := v_{\rm curr}$$
$$q_{\rm prev} := q_{\rm curr};\ w_{\rm prev} := w_{\rm curr}$$

○

It is observed that, as in the conjugate gradient algorithm, the updates occur in such a way that current and previous values can share the same storage. Therefore, this algorithm requires four vectors for the non-symmetric case and two for the symmetric case. An extra vector is required if the scaling $Au_{\rm prev}$ is not to be made twice, so five and three vectors are required for the non-symmetric and symmetric cases, respectively (not counting the solution vector, which is mentioned later, bringing the total to six and four).

If $z = Uy$, then $Az = b$ becomes $AUy = b$, and multiplying by V^T, then $Dy = V^Tb$, where $D = V^TAU$ is the special block-diagonal matrix exhibited in the section on conjugate linear system solving. It is desired that V^Tb be simple, so that b is not needed during the iteration.

The recursion $\psi_{i+1} = \phi_{i+1} - \beta_i\psi_i$ yields $v_{i+1} = w_{i+1} - \beta_i v_i$. Since W is from the orthogonal recursion, therefore $w_i^T b = 0$ for $i > 0$ if $q_0 = b$ is chosen (and therefore $u_0 = b$, as has been suggested above for other reasons). With this choice, the components of V^Tb are $[v_0^Tb, -\beta_0 v_0^Tb, \beta_1\beta_0 v_0^Tb, \cdots]^T$. From $Dy = V^Tb$,

$$y = \left[\frac{\delta_0}{\sigma_0}, \frac{-\beta_0\delta_0}{\sigma_1}, \frac{\beta_1\beta_0\delta_0}{\sigma_2}, \cdots\right],$$

but $\beta_i = \delta_{i+1}/\sigma_i$; therefore

$$y = \left[\frac{\delta_0}{\sigma_0}, \frac{-\delta_1\delta_0}{\sigma_1\sigma_0}, \frac{\delta_2\delta_1\delta_0}{\sigma_2\sigma_1\sigma_0}, \cdots\right].$$

Here again, $\delta_i = w_i^T q_i$ and $\sigma_i = v_i^T u_i$.

The solution $Dy = V^T b$ can then be found component-wise for y, and the linear system 'estimate' can also be updated component-wise from $z = Uy$, $z_{\mathrm{incr}} = z_{\mathrm{incr}} + y_i u_i$. Now if the linear system is 'shifted', as was explored above, then $b' = b - Az_0$, and if z' is the solution to $Az' = b'$, then the final solution will be $z' + z_0$. Therefore, if z_{incr} is initialized with z_0 and $b' = b - Az_0$ is used instead of b, then z_{incr} will contain the solution to $Az = b$, if the algorithm succeeds in finding a solution to the shifted problem.

Incorporating this shift results in exactly the general bi-conjugate gradient method, except for scaling (where in conjugate gradient terminology, z_0 is the initial linear solution estimate and b' is the initial residual).

The conjugate gradient method does not keep monic polynomials, if interpreted in a polynomial way, but instead scales so that the above expression for y does not contain past δ's and σ's. Monic polynomials will be preserved so that the lookahead methods developed above will apply without scaling.

Usual theorems on the orthogonality of residuals, the conjugacy of directions, various expressions for determining α and β, and so on, are immediate. In the case of symmetric A, choosing identical vectors $u_0 = v_0$ and $q_0 = w_0$, only v and w need be kept, and this becomes equivalent to the standard conjugate gradient method. Versions not setting $q_0 = b$ are also possible, but have the disadvantage of requiring b during the computation.

From the above development, a monic version of the conjugate gradient method in algorithm form is given in the following (here omitting the termination until breakdown is examined in the next section). The variable γ is equal to minus the current component of y above.

○ ALGORITHM 3.6.2 *Monic Conjugate Gradient*

INPUT: A, b, z_0, $v_0 = w_0$

$i := 0$
$z_{\mathrm{incr}} := z_0$
$\phi_i := \psi_i := 1$
$q_i := u_i := b - Az_0$
$\delta := v_i^T q_i$
$\gamma := 1$
do {

$$\sigma := v_i^T A u_i$$
$$\gamma := -(\delta/\sigma)\gamma$$
$$z_{\mathrm{incr}} := z_{\mathrm{incr}} - \gamma u_i$$

$$\alpha_i := \epsilon/\delta$$

$$\phi_{i+1} := x\psi_i - \alpha_i \phi_i$$
$$q_{i+1} := A u_i - \alpha_i q_i$$
$$w_{i+1} := A^T v_i - \alpha_i w_i$$

$$\delta := w_{i+1}^T q_{i+1}$$
$$\beta_i := \delta/\epsilon$$

$$\psi_{i+1} := \phi_{i+1} - \beta_i \psi_i$$
$$u_{i+1} := q_{i+1} - \beta_i u_i$$
$$v_{i+1} := w_{i+1} - \beta_i v_i$$
$$i := i + 1$$

}

○

3.6.1 Breakdowns, Lookahead and the Conjugate Gradient Method

Since, as shown above, the conjugate gradient method is based on both orthogonal and conjugate Lanczos processes, it too can have breakdowns occurring in exactly the same way as in the Lanczos algorithm. In the lookahead Lanczos methods, the polynomial which caused the breakdown was shifted by multiplying by x until a nonzero discrepancy appeared. A similar method can be applied to the conjugate gradient case.

Assume that a breakdown has occurred on the conjugate side of the conjugate gradient at polynomial ψ_k, with ℓ unforced zeros. It is possible to shift ψ_k and skew-diagonalize, producing $\psi_k, \psi_{k+1}, \cdots, \psi_{k+\ell}$, just as in the Lanczos case. Local variables ϕ and ψ will be used now instead of powers x^k, as in the Lanczos case. For conjugate gradient application, it is not necessary to shift until the nonzero discrepancy is at the level of ψ_{k-1}'s, but only to the level where ψ_k should have produced a nonzero:

$$
\begin{array}{c|cccccc}
 & \psi_{k-1} & \psi_k & \psi_{k+1} & \cdots & \psi_{k+\ell-1} & \psi_{k+\ell} \\
\hline
\psi_{k-1} & \times & & & & & \\
\psi_k & & & & & & \times \\
x\psi_k & & & & & \times & \\
\vdots & & & & \iddots & & \\
x^{\ell-1}\psi_k & & & \times & & & \\
x^{\ell}\psi_k & & \times & & & &
\end{array}
$$

Next, this whole lookahead block can be transferred to the orthogonal side (multiplying each by x):

$$
\begin{array}{ccc|ccccccc}
 & & & & \phi_{k+1} & \phi_{k+2} & \cdots & \phi_{k+\ell} & \phi_{k+\ell+1} & \\
 & & & & \| & \| & & \| & \| & \\
 & & & \phi_k & x\psi_k & x\psi_{k+1} & \cdots & x\psi_{k+\ell-1} & x\psi_{k+\ell} & \phi'_{k+\ell+1} \\
\hline
 & & \phi_k & \times & & & & & \times & \\
\phi_{k+1} & = & x\psi_k & & & & & \times & & \\
\vdots & & \vdots & & & & \iddots & & & \\
\phi_{k+\ell-1} & = & x^{\ell-1}\psi_k & & & \times & & & & \\
\phi_{k+\ell} & = & x^{\ell}\psi_k & & \times & & & & & \\
 & & \phi'_{k+\ell+1} & & & & & & & \otimes
\end{array}
$$

At this point, the column corresponding to $\phi_{k+\ell+1}$ can be zeroed using ϕ_k, producing $\phi'_{k+\ell+1}$.

Now suppose that lookahead occurs on the orthogonal side:

$$
\begin{array}{c|cccccc}
 & \phi_{k-1} & \phi_k & \phi_{k+1} & \cdots & \phi_{k+\ell-1} & \phi_{k+\ell} \\
\hline
\phi_{k-1} & \times & & & & & \\
\phi_k & & & & & & \times \\
x\phi_k & & & & & \times & \\
\vdots & & & & \iddots & & \\
x^{\ell-1}\phi_k & & & \times & & & \\
x^{\ell}\phi_k & & \times & & & &
\end{array}
$$

Similarly to the breakdown on the conjugate side, this whole lookahead block can be transferred to the conjugate side (this time no scaling by x is required):

$$
\begin{array}{rcl|ccccccc}
 & & & & \psi_k & \psi_{k+1} & \cdots & \psi_{k+\ell-1} & \psi_{k+\ell} & \\
 & & & & \| & \| & & \| & \| & \\
 & & & \psi_{k-1} & x\phi_k & x\phi_{k+1} & \cdots & x\phi_{k+\ell-1} & x\phi_{k+\ell} & \psi'_{k+\ell} \\
\hline
 & & \psi_{k-1} & \times & & & & & \times & \\
\psi_k & = & x\phi_k & & & & & \times & & \\
\vdots & & \vdots & & & & \iddots & & & \\
\psi_{k+\ell-2} & = & x^{\ell-1}\phi_k & & & \times & & & & \\
\psi_{k+\ell-1} & = & x^{\ell}\phi_k & & \times & & & & & \\
 & & \psi'_{k+\ell} & & & & & & & \otimes
\end{array}
$$

Thus, lookahead can be applied to the conjugate gradient method in an integrated fashion, with lookahead blocks being transferred between the conjugate and orthogonal sides, and vice versa. Notice that the size of the block is reduced by one when transferred in either direction. The column corresponding to $\psi_{k+\ell}$ has been zeroed using ψ_{k-1}, producing $\psi'_{k+\ell}$.

The lookahead algorithm is similar to the Lanczos lookahead procedures given above. In terms of polynomials only, the lookahead on the conjugate side is given by:

○ ALGORITHM 3.6.3 *Conjugate-Side Lookahead*

$\psi_i \ := \ \phi_i - ([\phi_i\phi_i]/[\psi_{i-1}\psi_{i-1}])\psi_{i-1}$
$\phi_{i+1} \ := \ x\phi_i$

$l \ := \ 0$
while $[\psi_i x^\ell \psi_i] = 0$ *do* $\{$ (find lookahead degree)
  $\ell \ := \ \ell + 1$
$\}$

for $j \ := \ 0$ *to* $\ell - 1$ *do* $\{$ (skew-diagonalize)

$$\beta_j := [\psi_i x^{\ell+1}\psi_{i+j}]/[\psi_i x^\ell \psi_i]$$
$$\psi_{i+j+1} := x\psi_{i+j} - \beta_j\psi_i$$
$$\phi_{i+j+1} := x\psi_{i+j}$$
}

$$\phi_{i+\ell+1} := x\psi_{i+\ell}$$
$$\phi'_{i+\ell+1} := \phi_{i+\ell+1} - ([\psi_i x^\ell \psi_i]/[\phi_i\phi_i])\phi_i$$

○

Here the inner products $[\cdot]$ containing ψ are understood to be conjugate (that is, containing an extra x), and those containing ϕ are understood to be orthogonal inner products, as defined previously.

Similarly, lookahead on the orthogonal side, in terms of polynomials, is:

○ ALGORITHM 3.6.4 *Orthogonal-Side Lookahead*

$$\phi_i := x\psi_{i-1} - ([\psi_{i-1}\psi_{i-1}]/[\phi_{i-1}\phi_{i-1}])\phi_{i-1}$$
$$\psi_i := \psi_i$$

$l := 0$
while $[\phi_i x^\ell \phi_i] = 0$ *do* { (find lookahead degree)
$\ell := \ell + 1$
}

for $j := 0$ *to* $\ell - 1$ *do* { (skew-diagonalize)
$$\alpha_j := [\phi_i x^{\ell+1}\phi_{i+j}]/[\phi_i x^\ell \phi_i]$$
$$\phi_{i+j+1} := x\phi_{i+j} - \alpha_j\phi_i$$
$$\psi_{i+j} := x\phi_{i+j}$$
}

$$\psi_{i+\ell} := x\phi_{i+\ell}$$
$$\psi'_{i+\ell} := \psi_{i+\ell} - ([\phi_i x^\ell \phi_i]/[\psi_{i-1}\psi_{i-1}])\psi_{i-1}$$

○

To specify the algorithm completely, it remains to cast it in terms of vectors, but first termination and incurable breakdown will be considered. Since each portion of the conjugate gradient is a Lanczos process, the development given for Lanczos and incurable breakdown applies here. More specifically, upon incurable breakdown a factor of the minimal polynomial is obtained. When computing the lookahead degree for the i-th vector on either side (where the first vector is the 0-th), it is therefore not necessary to go past ℓ such that $2i + \ell \geq 2n$, where n is the dimension of A (or an upper bound on its rank).

If an incurable breakdown occurs on the orthogonal side, then transferring, an incurable breakdown is produced on the conjugate side, with the identical polynomial. However, if an incurable breakdown occurs on the conjugate side, then in transferring the polynomial to the orthogonal side, a factor of x is introduced. Therefore, if a conjugate breakdown occurs, the matrix A is singular (and the dependency can be obtained from the polynomial, as was explained in the Lanczos case).

Linear systems are to be solved using the conjugate side of the recurrence. If $z = Uy$, then $Az = b$ becomes $AUy = b$, and multiplying by V^T, then $Dy = V^T b$. For reasons given in the non-lookahead version, $q_0 = b$ is chosen (and therefore $u_0 = b$).

During the lookahead procedure, the vectors v are scaled by A^T as the lookahead continues. Thus, it might be thought that $v_0^T A^j b$ would have to be pre-computed, limiting the size of lookahead that could be handled without recalling b.

Fortunately, this is not the case. If lookahead occurs on the conjugate side, then all but the first vector v_k, corresponding to ψ_k, will be orthogonal to b, since they are conjugate to $u_0 = b$. If a lookahead block is generated from the orthogonal side, then all the vectors transferred from the orthogonal side are orthogonal to $q_0 = b$. Thus, only the first vector of a lookahead block generated on the conjugate side (including $\ell = 0$ blocks) is possibly non-orthogonal to b, and has a potential effect upon the solution vector.

For this reason, it is possible to retain the arrangement used in the non-lookahead version, which updates γ (minus the current component of y, as above) by scaling with $-\delta/\sigma$, and incrementing the solution vector only when a new v vector has been produced. Shifting of the right-hand side is exactly similar to the non-lookahead version.

Observing the polynomial lookahead iterations exhibited above, it is apparent that an extra vector is needed for each of v, u, w and q, if lookahead is required. These extra vectors will be labelled v', u', w' and q'. In addition, polynomials ϕ' and ψ' are needed during lookahead. Since lookahead is engendered only by one side at a time, the variables v', u' and w', q' can share storage, as can ϕ' and ψ'. If no lookahead occurs, then only unprimed variables are needed, and the algorithm degenerates into the monic conjugate gradient algorithm given above.

Assembling the orthogonal and conjugate recursions together, and filling in the vectors, the full lookahead conjugate gradient algorithm is obtained:

○ ALGORITHM 3.6.5 *Lookahead Conjugate Gradient*

INPUT A, q, v, z

$\phi := 1$; $\psi := 1$
$q := b - Az$
$u := q$
$w := v$

$t := Au$
$\delta := w^T q$
$\sigma := v^T t$

$\gamma := 1$
$i := 0$; $l := 0$
ϕ-*incurable* $:=$ *false*
ψ-*incurable* $:=$ *false*
while $i \leq n$ *do* {
  $\alpha := \sigma/\delta$
  $\gamma := -(\delta/\sigma)\gamma$
  if $l \neq 0$ *then* {
    $z := z - \gamma q'$
    $\phi := x\phi' - \alpha\phi$
    $q := Aq' - \alpha q$
    $w := A^T w' - \alpha w$

```
        i := i + l + 1
}

if l = 0 then {
        z := z − γu
        φ := xψ − αφ
        q := t − αq
        w := A^T v − αw
        i := i + 1
}

/⋆ new ψ: ⋆/
l := 0
δ := w^T q
if δ = 0 then {
        /⋆ find lookahead degree: ⋆/
        v' := w
        do {
                l := l + 1
                v' := A^T v'
                δ := (v')^T q
                if (2i + l ≥ 2n) then {
                        φ-incurable := true
                        break
                }
        } while δ = 0

        if φ-incurable then break

        u' := q; ψ' := φ
        for j := 0 to l − 1 do {
                u' := Au'
                α_j := (v')^T u'/δ
```

```
            u′ := u′ − α_j q
            ψ′ := xψ′ − α_j φ
        }

        t := w
        for j := 0 to l − 1 do {
            v′ := v′ − α_{l−1−j} t
            if j < l − 1 then t := A^T t
        }
    }

    β := δ/σ
    if l ≠ 0 then {
        ψ := ψ′ − βψ
        u := u′ − βu
        v := v′ − βv
        i := i + l
    }

    if l = 0 then {
        ψ := φ − βψ
        u := q − βu
        v := w − βv
    }

    /⋆ new φ: ⋆/
    l := 0
    t := Au
    σ := v^T t
    if σ = 0 then {
        /⋆ find lookahead degree: ⋆/
        w′ := v
        do {
```

```
                l := l + 1
                w' := A^T w'
                σ := (w')^T t
                if (2i + l ≥ 2n) then {
                        ψ-incurable := true
                        break
                }
        while σ = 0 {

        if ψ-incurable then break

        q' := u
        φ' := ψ
        for j := 0 to l − 1 do {
                q' := Aq'
                β_j := (w')^T Aq'/σ
                q' := q' − β_j u
                φ' := xφ' − β_j ψ
        }

        t := v
        for j := 0 to l − 1 do {
                w' := w' − β_{l−1−j} t
                if j < l − 1 then t := A^T t
        }
    }
}

if φ-incurable then RETURN 'factor' φ
if ψ-incurable then RETURN 'factor' ψ

if w ≠ 0 and w^T A = 0 then RETURN 'row dependency' w
if q ≠ 0 and Aq = 0 then RETURN 'col dependency' q
```

if $v \neq 0$ *and* $v^T A = 0$ *then RETURN 'row dependency'* v
if $u \neq 0$ *and* $Au = 0$ *then RETURN 'col dependency'* u

if $Az - b = 0$ *then RETURN 'solution'* z

○

Here, as in the non-lookahead version given above, the vector t has been employed to remove the need to scale twice (and t is also used in the skew-diagonalization). As before, the RETURN statements are not assumed to be exclusive, so it is possible that multiple pieces of information may be returned from this algorithm. As was explained in the Lanczos algorithm, it is prudent to check for dependencies if vectors which are not zero have produced incurable breakdowns. Here each nonzero vector is checked for this occurrence.

As for the Lanczos methods, it is possible that an initial lookahead block is required, or that the choice $q_0 = b$ is not possible for any shift of the linear system. In this case, v_0 can be chosen explicitly to produce a non-incurable initial block, and the algorithm can be adjusted appropriately.

As before with the Lanczos algorithms, it is not necessary to update the polynomials, but only to store enough information (α, β, α_i and β_i) to recreate them on incurable breakdown. The algorithm is essentially the monic conjugate gradient presented above, only complicated by possible lookahead on the conjugate and orthogonal updates.

For a non-symmetric matrix that requires lookahead, the number of vectors required is eight: six for v, u, w, q, t, z, and two to provide storage for v', u', w' and q'. For symmetric matrices, this is reduced to six (not five, because two vectors are still required for lookahead). Lookahead vectors are required during lookahead alone, so it is possible that some alternate, possibly much slower, provision be made for their storage. Whether this is practical depends upon the likelihood of lookahead, which is examined later. For the non-lookahead conjugate gradient, six and four vectors were required for the non-symmetric and symmetric cases, respectively.

3.7 Initialization

Now that orthogonal and conjugate Lanczos algorithms have been presented, as well as a similar lookahead version of the conjugate gradient method, the question of choosing the initial vectors can be considered in detail.

It has been shown that there are advantages to choosing $u_0 = b$. This choice allows the algorithm to revert to the Wiedemann algorithm upon incurable breakdown, and also ensures the representation of $z = Uy$ in some cases. Initialization with this choice is considered in the following.

For the non-symmetric case, v_0 can be chosen freely. For the orthogonal case, v_0 can be chosen so that $v_0^T b \neq 0$, which is possible as long as b is not zero; but $b = 0$ has the solution $z = 0$. For the conjugate case, v_0 is chosen so that $v_0^T Ab \neq 0$, which is possible as long as b is not in the null space of A, and therefore a dependency.

For the conjugate gradient method, both orthogonal and conjugate portions must be considered. Neither b nor Ab can be zero; otherwise a solution or a dependency is obtained, as above. If b and Ab have a common nonzero location, then v_0 nonzero only in that location is suitable. Otherwise, b and Ab have no common nonzero location. A vector v_0 nonzero in two locations (one nonzero corresponding to the nonzero in b and the other to Ab) is then suitable.

For the symmetric version, freedom in choosing v_0 implies keeping all the vectors required for a non-symmetric algorithm. This is possible, but should be resorted to only when necessary. Otherwise, $v_0 = u_0 = b$, or by shifting, $v_0 = u_0 = b' = b - Az_0$. Here z_0 is the initial estimate, or equivalently, the shift vector.

Consider the orthogonal case. The first block cannot be made of dimension one if $(b - Az_0)^T(b - Az_0) = 0$ for every z_0. Thus $b^T b - z_0^T Ab - b^T Az_0 + z_0^T A^2 z_0 = 0$, but $b^T b$ must be 0, or else $z_0 = 0$ would have sufficed; therefore, $-2b^T Az_0 + z_0^T A^2 z_0 = 0$, for every z_0.

If the field has characteristic two, then $z_0^T A^2 z_0 = 0$ which, taking z_0 as elementary unit vectors, implies that every row (or column, but A is symmetric) is orthogonal to itself. If every row is orthogonal to itself, then no shift produces a nonzero initial element. To prove this, take $(\alpha z_0 + \beta z_0')A^2(\alpha z_0 + \beta z_0')$, which is $\alpha^2 z_0^T A^2 z_0 + \beta^2 (z_0')^T A^2 z_0'$, and proceed by induction over the elementary vectors by dimension. Therefore, for the characteristic two case, if every row is orthogonal to itself, then only an initial lookahead block can be produced; but if this proves incurable, then a suitable v_0 can be selected, and the non-symmetric algorithm must be applied.

If the field is not characteristic two, then $(-2b^T + (z_0 + z_0')^T A)A(z_0 + z_0') = 0$ is zero for all z_0 and z_0', which is also $-2b^T A(z_0 + z_0') + z_0^T A^2 z_0 + (z_0')^T A^2 z_0' + 2z_0^T A^2 z_0' = 0$, or $(-2b^T + z_0^T A)Az_0 + (-2b^T + (z_0')^T A)Az_0' + 2z_0^T A^2 z_0' = 0$. The first two terms must themselves be 0, so $z_0^T A^2 z_0' = 0$ for all z_0 and z_0' (if z_0's are selected to be elementary unit vectors, then for the characteristic not two case, all rows must be orthogonal to every other row).

Applying this to $-2b^T Az_0 + z_0^T A^2 z_0 = 0$ implies that $b^T Az_0 = 0$ for all z_0; therefore

$b^T A = 0$, and thus a row dependency is required. To demonstrate that $b^T A \neq 0$ is sufficient to find such a z_0, first assume that $z_0 = 0$ does not suffice, therefore $b^T b = 0$. Then a suitable z_0 must be selected with $-2b^T A z_0 + z_0^T A^2 z_0 \neq 0$. Suppose that a nonzero element of $-2b^T A$ is γ, then let z_0 select and scale this element to $\alpha\gamma$, and let the component of $z_0^T A^2 z_0$ be $\alpha^2\delta$, possibly zero. Then any $\alpha \neq 0$ not solving $(\gamma + \alpha\delta) = 0$ is suitable (which exists for characteristic greater than two), and a suitable z_0 is determined. Therefore, if $b^T A \neq 0$, then a shift z_0 can always be selected to produce a nonzero initial element, and otherwise there is a dependency.

For the conjugate symmetric case, a nonzero initial element cannot be created with any shift if $(b - Az_0)^T A(b - Az_0) = 0$ for every z_0. Using a similar argument, this implies that $z_0^T A^3 z_0 = 0$ for all z_0 in the characteristic two case. In a similar way, this implies that every row of A is conjugate to itself, and that is sufficient for $z_0^T A^3 z_0$ to be zero for all z_0. In this situation, the non-symmetric version can again be employed.

In the characteristic greater than two case, an argument similar to that above shows that $z_0^T A^3 z_0' = 0$ and $b^T A^2 z_0 = 0$ for every z_0 and z_0', Therefore, $b^T A$ is a left null vector of A, and again, a dependency is produced. Again, to show that $b^T A^2 \neq 0$ is sufficient to find such a z_0, first assume that $z_0 = 0$ does not suffice, therefore $b^T Ab = 0$. Then a suitable z_0 must be selected with $-2b^T A^2 z_0 + z_0^T A^3 z_0 \neq 0$. Suppose that a nonzero element of $-2b^T A$ is ρ; then let z_0 select and scale this element to $\alpha\rho$, and let the component of $z_0^T A^3 z_0$ be $\alpha^2\sigma$, possibly zero. Then any $\alpha \neq 0$ not solving $(\rho + \alpha\sigma) = 0$ is suitable (and such a non-solution exists for characteristic not two), and suitable z_0 is determined. Therefore, if $b^T A^2 \neq 0$, then a shift z_0 can always be selected to produce a nonzero initial element, and otherwise a dependency is obtained.

For the conjugate gradient method, both the orthogonal and conjugate initial inner products are to be nonzero. These quantities are $b^T b - 2b^T Az_0 + z_0^T A^2 z_0$ and $b^T Ab - 2b^T A^2 z_0 + z_0^T A^3 z_0$, respectively.

In the characteristic two case, $b^T b + z_0^T A^2 z_0$ and $b^T Ab + z_0^T A^3 z_0$ are to be nonzero. One or both of $b^T b$ and $b^T Ab$ must be zero; otherwise $z_0 = 0$ suffices. Let z_0 be chosen from the elementary vectors. If $b^T b = 0$ and $z_0^T A^2 z_0$ is zero for every elementary vector, then shifting cannot be effective, and a non-symmetric version (or, if possible, an initial lookahead block) is required; likewise for $b^T Ab$ and $z_0^T A^3 z_0$.

Therefore if $b^T b = 0$, then assume that there is an elementary vector z_0 such that $z_0^T A^2 z_0 \neq 0$, and similarly if $b^T Ab = 0$, then assume that there exists an elementary z_0' such that $(z_0')^T A^3 z_0' \neq 0$. If only one of $b^T b = 0$ or $b^T Ab = 0$ is zero, then let z_0 select and scale a nonzero element of $z_0^T A^2 z_0$ or $z_0^T A^3 z_0$ as $b^T b = 0$ or $b^T Ab = 0$, respectively. This produces, from $b^T b + z_0^T A^2 z_0$ and $b^T Ab + z_0^T A^3 z_0$, two second-degree polynomials $\eta + \alpha^2\delta$ and $\theta + \alpha^2\sigma$, where α is the scaling from z_0 and $\delta \neq 0$ if $\eta = 0$, and $\sigma \neq 0$

if $\theta = 0$, and one of η or θ is zero. Neither of these polynomials is to be zero for α. If the solutions to these polynomials are distinct, this excludes at most three elements, so if $p > 3$, then a suitable α is determined.

If both $b^T b = 0$ and $b^T A b = 0$, then both $z_0^T A^2 z_0 \neq 0$ and $(z_0')^T A^3 z_0' \neq 0$ for elementary z_0 and z_0'. If $z_0^T A^2 z_0$ and $z_0^T A^3 z_0$ have no common z_0 making them nonzero, then choose a z_0' nonzero in the different locations, which excludes only the scale 0. If $z_0^T A^2 z_0$ and $z_0^T A^3 z_0$ have only one common nonzero, then still only the scale of 0 is excluded, so in either case suitable z_0 is obtained.

In the characteristic not two case, $b^T b - 2b^T A z_0 + z_0^T A^2 z_0$ and $b^T A b - 2b^T A^2 z_0 + z_0^T A^3 z_0$ are to be nonzero, and it can be assumed that one of $b^T b$ or $b^T A b$ or both is zero, because otherwise $z_0 = 0$ suffices, and that each of $b^T A$ and $b^T A^2$ is nonzero, because otherwise a dependency is available. Here $b^T A$ and $b^T A^2$ being nonzero only in a common position is the most restrictive case.

This introduces two at most second-degree polynomials in α, neither of which can be solved by α if it is to produce a useful shift. At least one of the polynomials has a zero constant term, and both polynomials have nonzero linear terms. Therefore, if the number of elements exceeds four, a shift is always possible (and can be found explicitly).

We have shown:

Theorem 3.7.1 1. *For the non-symmetric orthogonal and conjugate Lanczos algorithms, and also for the conjugate gradient algorithm with initial vector $u_0 = b$, initial v_0 can be chosen producing a nonzero initial element, or otherwise a dependency is obtained.*

2. *For the symmetric orthogonal and conjugate Lanczos algorithms, and also for the symmetric conjugate gradient algorithm:*

   (a) *In characteristic two, if $z_0 = 0$ is not suitable (where suitable z produces a nonzero initial element), then if every row of A is self-orthogonal (or self-conjugate, as appropriate), no suitable shift z_0 exists (if the initial lookahead block is incurable, then the non-symmetric algorithm can be used).*

   (b) *In characteristic not two, if $z_0 = 0$ is not suitable, and if the number of elements exceeds four, then if $b^T A$ is nonzero, a suitable shift exists (and can be determined) for the orthogonal Lanczos algorithm. Similarly, if $b^T A^2$ is nonzero, then a suitable shift exists (and can be determined) for the conjugate Lanczos algorithm. Otherwise, for both orthogonal and conjugate versions, a dependency is obtained. In the case that $z_0 = 0$ is not suitable, a suitable shift for the conjugate gradient can also be determined if both $b^T A$ and $b^T A^2$ are nonzero, and otherwise, a dependency is available.*

We are concerned here with large prime fields; therefore, it can be assumed that suitable initializations exist, at least for the first trial of the algorithm (or that dependencies are found). The course of action after this first application of Lanczos or conjugate gradient is discussed briefly in the next section.

3.8 Integrating the Methods

It was concluded above that upon incurable breakdown not yielding solution, the resulting factor of the minimal polynomial could still be useful as one pass of Wiedemann's algorithm. The integration of Wiedemann's algorithm and Lanczos or conjugate gradient algorithms is considered briefly in this section.

Wiedemann's algorithm 1, as described in the last chapter, builds the minimal polynomial f of the sequence $b, Ab, A^2b, \cdots$, beginning with $b_0 = b$, obtaining the factor f_0, defining $b_1 = f_0(A)b_0$, obtaining f_1, and thereby building $f = f_k f_{k-1} \cdots f_1 f_0$, factor by factor, until $b_k = 0$. Each factor is assumed to be scaled to have trailing coefficient one, which assumes that A is not singular.

To find the solution, Wiedemann defines $\bar{f}(x) = (f(x) - f(0))/x$, and if A is nonsingular, then $z = -\bar{f}(A)b$. Instead of calculating this at the end, Wiedemann suggests that the solution z be updated for each new factor, $z := z + \bar{f}_i(A)b_i$ [36].

If a similar polynomial $\bar{\phi}(x)$ is maintained in the Lanczos and conjugate gradient methods, then the same means of solution is possible there. The polynomial itself need not be calculated, but only the vector evaluation of the polynomial, along with the constant coefficient of the polynomial, which will be required for scaling, since all of the recursions given for these methods keep the polynomials monic, and Wiedemann's method instead requires the constant term to be one.

This implies two things: first, that the Lanczos and conjugate gradient methods may be used iteratively to update the solution as factors of the minimal polynomial are obtained; and second, that if the constant term of the polynomial is maintained, then the complete polynomial will never need to be calculated, even in the case that dependencies are discovered.

The Lanczos and conjugate gradient algorithms can be used as replacements for the Berlekamp-Massey algorithm in Wiedemann's method. These methods have the advantage that $\bar{f}(A)b$ does not have to be recalculated.

3.9 Probability of Lookahead

In planning the implementation of the methods described above, the likelihood of lookahead, and the extra memory that this requires, should be estimated.

In this section, we will estimate both the size of the lookahead blocks and the probability of incurable breakdown. The Lanczos process, as has been described, forces a zero into a column of a transformed matrix. As has been seen, lookahead is required if extra cancellation occurs. Incurable breakdown occurs when $2k + \ell \geq 2r$, for k the point in the Lanczos algorithm, ℓ the number of extra cancellations, and r the rank of the Krylov subspace of A, which is not greater than the dimension of A.

Assume that the elements below the forced zero are randomly and independently distributed. (Few apologies for this assumption are required, since the sieving process that generates A and b is not well understood in its distribution, and a similar assumption is made for its analysis.)

The probability of ℓ cancellations is then $(1/p)^\ell$; that is, ℓ independent trials, when working over the numbers modulo p. At position k in the Lanczos algorithm, incurable breakdown will then occur with probability $(1/p)^{2(r-k)}$. The expected degree of lookahead is

$$\sum_{\ell=0}^{2(r-k)} \ell(1/p)^\ell = \frac{-2r + 2k + 2pr - 2pk + p}{p^{2(r-k)}(1-p)^2} + \frac{p}{(1-p)^2},$$

which takes its maximum when $k = 0$. When r becomes large, this approaches

$$(2p-1)/(p(p-1)^2),$$

which is $2/p^2 + O(1/p^3)$.

The probability that no lookahead at all is required during the course of the Lanczos process is $(1 - 1/p)^r$. Expressions for the probability of only single-degree lookahead, and other similar quantities, are easily obtained.

When p is very large, the probability of lookahead can be very small, so it is important to use a method that imposes very little overhead to allow lookahead. The algorithms above match this requirement.

3.10 Summary

This chapter has presented a unifying theory for the Wiedemann, Lanczos and conjugate gradient algorithms used to solve sparse linear systems over finite fields, including a development of lookahead methods required to circumvent the breakdowns that can occur in the Lanczos and conjugate gradient methods.

Polynomial versions were developed here first, and the connection between the Lanczos algorithm and the Wiedemann method was noted. In this direction, we demonstrated an analogue of Taylor's mismatch theorem for the Lanczos method over finite fields, which implies that a factor of the minimal polynomial is obtained when the Lanczos algorithm encounters incurable breakdown. Therefore, incurable breakdown of the Lanczos method still provides an equivalent single pass of the Wiedemann algorithm.

Simple and efficient new lookahead methods were developed, and practical vector versions were given, requiring only a constant number of vectors, regardless of the degree of lookahead. The conjugate gradient method was shown to be a coupled orthogonal and conjugate Lanczos algorithm. It was shown that lookahead methods can be applied to the conjugate gradient algorithm in an integrated manner (that is, that lookahead on the orthogonal side can be transferred to the conjugate side, and vice versa).

The question of the initialization of the vectors was considered, and it was shown that it is possible to explicitly find vectors which ensure that at least a linear factor of the minimal polynomial, or a dependency, is obtained (for the fields under consideration).

The integration of the Lanczos or conjugate gradient algorithm with the Wiedemann algorithm was briefly discussed. Finally, estimates on the probability and degree of lookahead were derived.

In summary, we have succeeded in finding a practical unity between the sparse linear solution techniques: the Wiedemann, Lanczos, and conjugate gradient algorithms.

Chapter 4

Analysis of Large Prime Variations

The large prime variation is very useful in practice for sieve methods used in factorization and discrete logarithms. While not giving any asymptotic advantage [29], it is able to speed up these methods by a constant factor, in practice by 2 to 2.5 times [21] for the quadratic sieve factoring method. This speed-up makes the large prime variation an essential component of modern factoring and discrete logarithm methods. The speed-up that is obtained on any given experiment appears to be highly variable [21], and therefore difficult to quantify beforehand.

Sieve numbers that completely factor in the factor base are relatively rare in practice, and the hope of the large prime variations is to retain not only completely smooth sieve numbers, but also those numbers which contain in addition one (or perhaps two) primes too large to be included in the factor base. Numbers which are smooth except for the possible addition of a single larger prime have been named semi-smooth [2], and those containing at most two larger primes might be called doubly-semi-smooth. Such numbers from the sieve can still be useful if the large primes occurring also appear in other such numbers, and if these numbers can be combined to eliminate completely the large primes. An extension of the Dickman rho function, expressing the asymptotic fraction of semi-smooth numbers, has been recently derived by Bach and Peralta [2].

Employing this estimate, we will consider in the next section the single large prime variation and its effect on the quadratic sieve factoring algorithm, applying methods of cell occupancy and Poisson approximation. The sieving procedure is similar to sieving procedures for discrete logarithm algorithms [4], and the quadratic sieve has the advantage that reports of the number of relations obtained from the large prime variation are available [8]. It is conceivable that similar analysis might be applied to the large prime variation as applied to number field sieve methods, but would be more difficult.

It will be shown that the number of relations obtained from the single large prime variation tends to a Poisson random variable, and therefore that the variance is equal to the mean. From this, we conclude that such predictions cannot be made very accurately.

To estimate the number of doubly-semi-smooth partial relations (partial because they contain large primes that must be cancelled) generated by the sieve, a further extension to the Dickman rho function is required, expressing the asymptotic fraction of doubly-semi-smooth numbers. This extension is developed here, and an effective method for computing this new quantity is derived.

The combining (cancelling) process for double large prime relations is more complicated, but it can be expressed as the number of fundamental cycles of a random graph. Results from the theory of random graphs suggest that the number of relations obtained from the double large prime relation also tends to a Poisson-distributed random variable, as did the single large prime case. Therefore, given the assumption that the partial relations are generated randomly, an estimate of the number of relations resulting from the cancellation cannot be expected to match experiments very closely, as in the single prime case above.

4.1 The Quadratic Sieve

The quadratic sieve attempts to factor n by sieving through numbers $S_m = (\lfloor\sqrt{n}\rfloor + m)^2 - n$, for m in an interval, saving those which are completely factorable (smooth) in the factor base B (a set consisting typically of the first b prime quadratic residues mod n, and -1). A multiplicative combination of these S_m forming a square (mod n) is found by finding a dependency, mod 2, in the vectors from V_m, consisting of the exponents of the factorization of S_m in B. Since each sieve number is itself a square (mod n), this produces congruent squares mod n, which may be useful in factoring n.

Already in the first quadratic sieve implementation by Gerver [8], single large primes were utilized, and in fact they contributed the majority of the relation used to find congruent squares. Gerver factored a 47-digit number, sieving roughly 1 billion values of m to obtain 327 relations from completely smooth sieve numbers (with factor base the first 999 quadratic residues modulo the number to be factored, plus -1), but also 690 relations were obtained from combining the 25,747 semi-smooth sieve numbers. The goal of a large prime analysis is to estimate, in advance, the quantity of relations that will be generated by combining semi-smooth (and doubly-semi-smooth) relations, and to determine how precisely these random quantities can be predicted. Experimentally, these quantities have shown large variance [21], and so it might be expected that these predictions have little advantage in practice, especially since the number of relations that will result can be found as sieving progresses, without actually doing the combining [21]. In fact, we show that the number of relations obtained from combining approaches Poisson random variable, and as such has standard deviation equal to the mean, so outcomes of experiments cannot be expected to closely match predictions.

The effect that can be expected on the quadratic sieve algorithm when using the single large prime variation is the topic of the next section.

4.2 The Single Large Prime Variation

The quadratic sieve searches an interval of m, finding those sieve numbers which are smooth in the factor base, and for the single large prime variation, finding also those numbers which contain a prime greater that the largest prime in the factor base b_s, but not greater than another prime b_l. (Often b_l is taken to be b_s^2, since these semi-smooth numbers can be obtained with almost no additional effort).

The estimation of the asymptotic fraction of smooth numbers has been studied extensively. For fixed $u > 1$, the number of integers up to x^u with all prime factors below x is $\sim \rho(u)x^u$ as $x \to \infty$, where $\rho(u)$ is a positive constant depending on u, called the Dickman ρ function [6, 30]. To estimate the number of semi-smooth sieve numbers, an extension of the Dickman function to semi-smooth numbers has been developed by Bach and Peralta [2].

The estimation of the fraction of smooth numbers will involve the Dickman ρ function. An asymptotic estimate of the fraction of numbers of size near S, having factors no greater that B, is then $\rho(\log_B(S))$. With this, the number of completely smooth numbers that will occur in the output of the sieve can be predicted. The m-th sieve number is of size $2m\sqrt{n} + O(\sqrt{n})$, so the number of smooth sieve numbers will be estimated by $\int \rho(\log_{b_s}(2m\sqrt{n}))dm$, where the integral is over the range of m sieved.

To predict the number of semi-smooth sieve numbers, Bach and Peralta's σ function is useful [2]. An asymptotic estimate of the fraction of numbers of size near S having factors no greater that b_s, except possibly for one no greater than b_l, is $\sigma(\log_{b_s}(S), \log_{b_l}(S))$. Removing the fraction which is completely smooth, the fraction having exactly one large prime is $\sigma(\log_{b_s}(S), \log_{b_l}(S)) - \rho(\log_{b_s}(S))$. The number of semi-smooth sieve numbers that appear in the output of the sieve is then roughly $\int \sigma(\log_{b_s}(2m\sqrt{n}), \log_{b_l}(2m\sqrt{n}))dm$.

The smooth sieve numbers are useful immediately for the remaining stages of the quadratic sieve, but the semi-smooth numbers need to be combined with others. If desired, the large primes occurring can be added to the factor base, but it is usually considered more efficient to eliminate them first.

Consider the range of large primes that can occur as cells, and think of each semi-smooth number as a ball that might occupy any of these cells, admittedly not with equal likelihood. Assuming that the large primes from the semi-smooth sieve numbers can be modeled as occurring randomly, the problem of predicting the number of useful and independent combinations of these semi-smooth sieve numbers is then reduced to the classical cell occupancy problem. Those cells containing a single prime will not yield any useful relation vector, but those containing two will yield one useful relation when the numbers are combined. Only independent relations (which are to be combined in the later stage of the quadratic sieve algorithm) are useful, so then those cells containing three balls will yield two useful relations, those with four will yield three, and so on. The total number of relations can then be found by obtaining the population of cells with occupancy 2, 3 and higher.

The number of cells is fixed, the number of prime quadratic residues in the large prime region being admitted. The number of balls can be found using the function σ, as above. Delay for now finding the distribution of a ball falling in cell p_i (for p_i a large prime quadratic residue). The mean and variance of the number of cells containing two or more, balls, and also of those cells containing exactly two, three, or more have been derived [17]. If I_i is an indicator for the i-th cell having the occupancy in question (2,3 or higher), then the mean number of cells with this occupancy is given by $\sum_i P(I_i = 1)$, and the first factorial moment by $\sum_{(i<j)} P(I_i = 1, I_j = 1)$, higher moments being obtained similarly. Probabilities on the indicators are obtained directly from the occupancy distribution of a single ball, which is assumed to fall independently. If μ_c is the cell population containing c balls, then let λ_c be the mean of μ_c.

The number of cells containing more than one ball (and similarly exactly two or more balls), has also been shown to approach asymptotically a Poisson-distributed random variable [17]. For uniform occupancy probability, this was first shown by von Mises. For practical use, it is more valuable to know just how fast these random variables approach a Poisson distribution, and recently a result of this type has been obtained [3], bounding the total variational distance from a Poisson distribution.

The total variation on two measures, γ and δ , is $\sup_{a\in A}\{|\gamma(a)-\delta(a)|\}$ for a any subset of the probability space A; in this case A is a subset of the non-negative integers. The distance in total variation between the distribution of the number of cells with occupancy c and the Poisson distribution with parameter λ_c is bounded by

$$\min(1, \lambda_c^{-1}) \left[6r(\sum_{k=1}^{n} q_k \pi_k)^2 + \sum_{k=1}^{n} \pi_k^2 + 6\lambda_c^2 m^2/r \right] ,$$

where n is the number of cells, q_k the occupancy probability for cell k of a single ball, π_k the Poisson rate of cell k, $\pi_k = \binom{r}{c} q_k^c (1 - q_k)^{r-c}$, r the number of balls, and λ_c the aggregate rate, $\lambda_c = \sum_{k=1}^{n} \pi_k$ (from [3]). The total variational distance also bounds the point probabilities, so the difference between the actual occupancy probability and the Poisson assumption is also bounded by the above.

The variance of a Poisson random variable is equal to its mean, λ, which implies that in any given experiment the variation in the number of cells containing two or more balls will vary widely. Therefore, intrinsically, predicting the number of useful relations obtained with the single large prime variation cannot be very precise, if it is assumed that the sieve produces semi-smooth numbers in the random way described.

It still remains to model the occupancy distribution of a single ball (which for our experiment is the likelihood that a given large prime will occur in a semi-smooth number obtained from the sieve). It is reasonable to assume that the likelihood that a certain randomly-chosen large number will be divisible by a prime p be proportional to $1/p$. Therefore, if S is a set of randomly-generated large numbers, all divisible by primes in some range R, then this assumption means the probability that a given prime from R divides a member of S, is proportional to $1/p$. This assumption was used by Lenstra and Manasse to model the distribution of large primes in the large prime output of the quadratic sieve. It was noted there that this assumption is not totally accurate, and a modification of the probability to $1/p^\alpha$ for an α fitted experimentally was proposed.

The reason the $1/p$ model does not accurately reflect the occurrence of primes in the large prime output of the sieve is that, for the single large prime variation, the remainder of the number must be completely smooth in the factor base. Numbers which have larger prime factors will have greater probability that the remainder is smooth in the factor base, simply because the remainder is smaller. The effect this has on the probability of the occurrence of large primes in the large prime output of the quadratic sieve (which is important to the analysis of the expected number of relations generated) is found by integrating the marginal distribution on the size of the sieve number. Let S_{ss} be the set of semi-smooth numbers from the sieve, let p be a specified large prime in the range considered, and let z range over the numbers considered in the sieve; then

$$
\begin{aligned}
P(p|s : s \text{ random} \in S_{\mathrm{ss}}) \quad &= \\
&\frac{\sum_z P(p|z \cap z \in S_{\mathrm{ss}})}{\sum_z P(z \in S_{\mathrm{ss}})} \\
&= C_1 \sum_z P(z \in S_{\mathrm{ss}} : p|z) P(p|z) \\
&= C_2 \sum_z P(z/p \in S_{\mathrm{ss}}) 1/p,
\end{aligned}
$$

where C_i are constants normalizing the probabilities.

This last quantity is asymptotically estimated by Dickman's rho function. Making a continuous approximation, and noting that sieving the m-th sieve number produces a quantity of size about $2m\sqrt{n}$, where n is the number to be factored, then

$$\begin{aligned} P(p|s : s \in S_{\mathrm{ss}}) &= \\ C/p \int \rho(\ln(2m\sqrt{n}/p)/\ln(b_s))dm, \end{aligned}$$

integrated over the interval of m sieved.

This probability is of the form $c(p)/p$, where there is a correction $c(p)$ for the effect of numbers with larger prime factors being more probably semi-smooth.

To discover how well the corrected model predicts practice, an experiment similar to the quadratic sieve of Gerver [8] was made, using the same number and factor base, except that the sieving range was extended to $5E9$. The ratio between the number of primes in a region and the number predicted by the uncorrected $1/p$ model was collected, and normalized to 1 for the first large prime. A graph showing the experimental data and the predicted correction, found by evaluating the integral above, is shown in Figure 4.2. A point on the graph is the number of primes in a region of $1E6$ divided by the expected number for the $1/p$ model for the same region. The horizontal axis is the size of the large prime.

The experiment matches the theoretical curve closely, except for the largest primes, where the experiment is beneath the prediction (there are also very few of these large primes occurring, so this could be the result of insufficient population for statistics). For these primes, it is extremely rare that there would be any primes having occupancy two or more, and so this error will have little effect on the prediction of these populations, and then also little effect on the prediction of the number of relations.

Applying this correction, the distribution of the single ball occupancy is slightly less skewed to small primes than suggested by the $1/p$ model, and the reduced concentration reduces the expected number of cells having double or greater occupancy.

Applying the above methods, we anticipate that this experiment would yield 950 sieve numbers that are completely smooth, but in reality there are 1,717. In fact, the number Gerver used has the feature that 3,5 and 8 are all quadratic residues, which makes it more likely that the sieve numbers will be divisible by these factors. Taking this fact into account, the prediction for smooth numbers becomes 2,083. The number of semi-smooth sieve numbers is predicted to be 194,265, and there were 161,510 found by the sieve.

The population of cells with certain occupancies is predicted by the Poisson approximation. For this the Poisson rate λ_c for the occupancy μ_c is needed. The probability

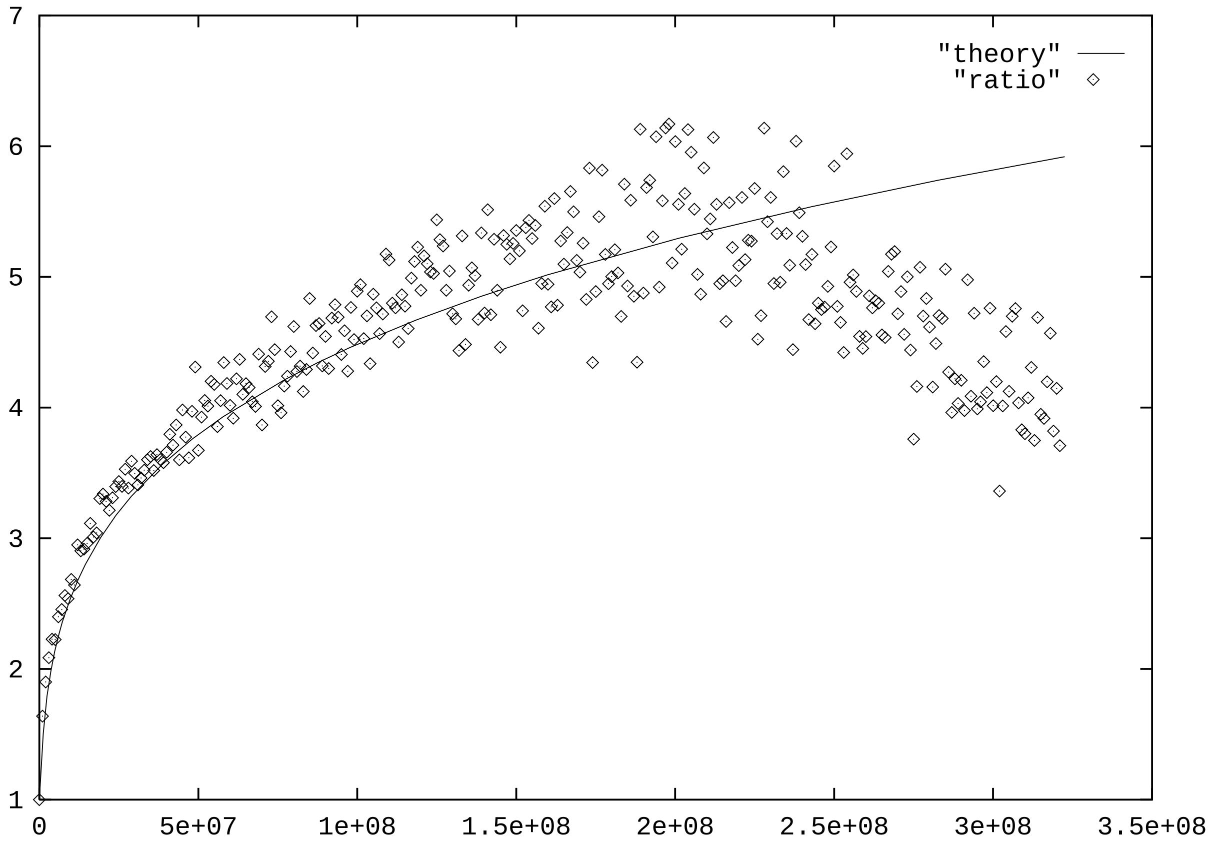

Figure 4.1: Experimental ratio and predicted ratio

occupancy	actual	$1/p$ model	corrected	TV bound
2	6,939	8,610	6,959	0.369
3	1,556	3,086	1,871	0.491
4	580	1,604	894	0.529
5	246	988	445	0.531
6	133	674	264	0.522

Table 4.1: Table of means and standard deviations of various occupancies.

that a ball falls in cell p is proportional to $c(p)/p$. Now assume that primes in the region are distributed as predicted by the prime number theorem. The constant normalizing the distribution is then roughly $\int c(p)1/(p\log(p))dp$, where the integral is over the admitted range of large primes. From this, obtain the probability q_i of a ball falling in cell i. Then the Poisson rate for double occupancy is $\lambda_2 = \sum \binom{r}{2} q_i^2(1-q_i)^{r-2}$. Approximating this by an integral, we obtain the prediction that if 161,510 semi-smooth sieve numbers are generated, then 6,959 primes will occur in pairs, and in fact there were 6,939. The $1/p$ model would predict 8,610 such primes. Table 4.2 gives the actual and predicted populations for multiple occurrences, for the $1/p$ and corrected models. (The TV column gives the bound on the distance between the actual and the Poisson distribution, given by the bound on the total variation above, for the corrected model). For these calculations, the quadratic residues have been assumed to be distributed evenly, and the sum has been approximated by an integral. The estimate for the number of doubly occupied cells is surprisingly good, given the high variance; for higher occupancies the estimate is consistently high, yet still a much better prediction than the $1/p$ model.

As a first step in predicting the number of useful relations that may be obtained with the double large prime variation, it is necessary to have in addition a prediction of the number of doubly-semi-smooth numbers that can be expected to occur in the sieve. For this purpose, it is necessary to extend the σ function of Bach and Peralta to two large primes. This extension is the topic of the next section.

4.3 Asymptotic Doubly-Semi-Smoothness

Using notation similar to Knuth and Trabb Pardo [16], let the prime factorization of a positive number n be $n = n_{(1)}n_{(2)}\cdots$, where $n_{(1)} \geq n_{(2)} \geq \cdots$; that is, $n_{(1)}$ is the largest prime factor, $n_{(2)}$ is the second largest, and so on.

Define $\Psi_2(x, x^\beta, x^\alpha)$ to be the number of positive integers up to x having exactly two prime factors between x^α and x^β, with all the other prime factors below x^α, or more precisely,

$$\begin{aligned}
\Psi_2(x, x^\beta, x^\alpha) &= \sum_{x^\alpha < p_1 \le x^\beta} \sum_{x^\alpha < p_2 \le p_1} \#\{n \le x : n_{(1)} = p_1, n_{(2)} = p_2, n_{(3)} \le x^\alpha\} \\
&= \sum_{p_1, p_2} \# \left\{ m \le \frac{x}{p_1 p_2} : m_{(1)} \le x^\alpha \right\} \\
&= \sum_{p_1, p_2} \# \left\{ m \le \frac{x}{p_1 p_2} : m_{(1)} \le \left(\frac{x}{p_1 p_2}\right)^{\frac{\alpha}{1 - \log(p_1 p_2)/\log(x)}} \right\},
\end{aligned}$$

where $\sum_{p_1,p_2}$ is shorthand for $\sum_{x^\alpha < p_1 \le x^\beta} \sum_{x^\alpha < p_2 \le p_1}$.

Using results of de Bruijn, it has been shown [2] that

$$\#\{n \le t : n_{(1)} \le t^\gamma\} = t\rho(1/\gamma) + O\left(\frac{t}{\alpha \log(t)}\right),$$

for $0 < \alpha < 1$, $\alpha \le \gamma$ and $t^\alpha \ge 2$. Let $\gamma = \frac{\alpha}{1 - \log(p_1 p_2)/\log(x)}$. Then for $\beta < 1/2$, $\alpha < \gamma$, so this estimate implies that

$$\Psi_2(x, x^\beta, x^\alpha) = \sum_{p_1, p_2} \frac{x}{p_1 p_2} \rho\left(\frac{1 - \log(p_1 p_2)/\log(x)}{\alpha}\right) + O\left(\frac{1}{\alpha} \sum_{p_1, p_2} \frac{\frac{x}{p_1 p_2}}{\log(\frac{x}{p_1 p_2})}\right).$$

Examining the error term,

$$\frac{1}{\alpha} \sum_{p_1, p_2} \frac{\frac{x}{p_1 p_2}}{\log(\frac{x}{p_1 p_2})} = O\left(\frac{x}{\alpha \log x} \log(\frac{\beta}{\alpha}) \log\left(\frac{\beta(1 - 2\alpha)}{\alpha(1 - 2\beta)}\right)\right).$$

Let T be the first term of $\Psi_2(x, x^\beta, x^\alpha)$ above. Dividing by x and using Stieltjes integration,

$$\begin{aligned}
T/x &= \sum_{p_1, p_2} \frac{1}{p_1 p_2} \rho\left(\frac{1 - \log(p_1 p_2)/\log(x)}{\alpha}\right) \\
&= \int_{x^\alpha}^{x^\beta} \int_{x^\alpha}^{t_1} \rho\left(\frac{1 - \log(t_1 t_2)/\log(x)}{\alpha}\right) \frac{d\pi(t_2)}{t_2} \frac{d\pi(t_1)}{t_1}.
\end{aligned}$$

The prime number theorem implies [2] that the number of primes up to t, $\pi(t) = \mathrm{li}(t) + O(\frac{t}{\log^c t})$, for $c > 0$, and where $\mathrm{li}(t) = \int_0^t dx/\log x$. Now substitute $\pi(t_2) = \mathrm{li}(t_2) + \epsilon(t_2)$, where $\epsilon(t_2)$ is the error term, integrate by parts and collect terms to obtain

$$\begin{aligned} T/x &= \int_{x^\alpha}^{x^\beta} \left\{ \int_{x^\alpha}^{t_1} \rho\left(\frac{1-\log(t_1t_2)/\log(x)}{\alpha}\right) \frac{dt_2}{t_2 \log t_2} \right. \\ &\quad + \left[\rho\left(\frac{1-\log(t_1t_2)/\log(x)}{\alpha}\right)\frac{\epsilon(t_2)}{t_2}\right]_{x^\alpha}^{t_1} \\ &\quad \left. - \int_{x^\alpha}^{t_1} \frac{d}{dt}\left(\frac{1}{t_2}\rho\left(\frac{1-\log(t_1t_2)/\log(x)}{\alpha}\right)\right)\epsilon(t_2)dt_2 \right\} \frac{d\pi(t_1)}{t_1}. \end{aligned}$$

To bound the terms involving $\epsilon(t_2)$, use the bound given above for the prime number theorem, and also the fact that $|\rho| \le 1$, $|\rho'| \le 1$ [2]. Then

$$\left[\rho\left(\frac{1-\log(t_1t_2)/\log(x)}{\alpha}\right)\frac{\epsilon(t_2)}{t_2}\right]_{x^\alpha}^{t_1} = O\left(\frac{1}{\alpha \log x}\right),$$

and similarly, differentiating, bounding ρ and ρ' by 1, and using the prime number theorem bound with $c = 2$, gives

$$-\int_{x^\alpha}^{t_1} \frac{d}{dt_2}\left(\frac{1}{t_2}\rho\left(\frac{1-\log(t_1t_2)/\log(x)}{\alpha}\right)\right)\epsilon(t_2)dt_2 = O\left(\frac{1}{\alpha \log x}\right).$$

Combining, this yields the estimate

$$T/x = \int_{x^\alpha}^{x^\beta} \left\{ \int_{x^\alpha}^{t_1} \rho\left(\frac{1-\log(t_1t_2)/\log(x)}{\alpha}\right) \frac{dt_2}{t_2 \log t_2} + O\left(\frac{1}{\alpha \log x}\right) \right\} \frac{d\pi(t_1)}{t_1}.$$

Next, perform a similar manipulation with the variable t_1:

$$\begin{aligned} T/x &= \int_{x^\alpha}^{x^\beta} \int_{x^\alpha}^{t_1} \rho\left(\frac{1-\log(t_1t_2)/\log(x)}{\alpha}\right) \frac{dt_2}{t_2 \log t_2}\frac{dt_1}{t_1 \log t_1} \\ &+ \left[\int_{x^\alpha}^{t_1} \rho\left(\frac{1-\log(t_1t_2)/\log(x)}{\alpha}\right) \frac{dt_2}{t_2 \log t_2}\frac{\epsilon(t_1)}{t_1}\right]_{x^\alpha}^{x^\beta} \\ &- \int_{x^\alpha}^{x^\beta} \frac{d}{dt_1}\left(\frac{1}{t_1}\int_{x^\alpha}^{t_1} \rho\left(\frac{1-\log(t_1t_2)/\log(x)}{\alpha}\right) \frac{dt_2}{t_2 \log t_2}\right)\epsilon(t_1)dt_1 \\ &\quad +O\left\{ \int_{x^\alpha}^{x^\beta} \left(\frac{1}{\alpha \log x}\right)\frac{dt_1}{t_1 \log t_1} + \left[\left(\frac{1}{\alpha \log x}\right)\frac{\epsilon(t_1)}{t_1}\right]_{x^\alpha}^{x^\beta} \right. \\ &\quad \left. - \int_{x^\alpha}^{x^\beta} \frac{d}{dt_1}\left(\frac{1}{t_1 \alpha \log x}\right)\epsilon(t_1)dt_1 \right\}. \end{aligned}$$

Bounding the terms containing $\epsilon(t_1)$, similarly to the case with $\epsilon(t_2)$, we obtain

$$\Psi_2(x, x^\beta, x^\alpha)/x = \int_{x^\alpha}^{x^\beta}\int_{x^\alpha}^{x^\beta} \rho\left(\frac{1-\log(t_1 t_2)/\log(x)}{\alpha}\right)\frac{dt_2}{t_2\log t_2}\frac{dt_1}{t_1\log t_1} \quad (4.1)$$

$$+ \; O\left(\frac{\log(\beta/\alpha)}{\alpha\log x}\left(\log\left(\frac{\beta(1-2\alpha)}{\alpha(1-2\beta)}\right)+1\right)\right). \quad (4.2)$$

Substituting $\lambda_1 = \log t_1/\log x$, $\lambda_2 = \log t_2/\log x$, and using symmetry to change the limits of integration, the first term becomes

$$\frac{1}{2}\int_\alpha^\beta\int_\alpha^\beta \rho\left(\frac{1-\lambda_1-\lambda_2}{\alpha}\right)\frac{d\lambda_1}{\lambda_1}\frac{d\lambda_2}{\lambda_2}. \quad (4.3)$$

This proves an extension to a theorem of Bach and Peralta [2] to 2 large primes:

Theorem 4.3.1 *Examining equation (4.3), and the error term of equation (4.2), the limit*

$$G_2(\alpha,\beta) = \lim_{x\to\infty}\Psi_2(x, x^\beta, x^\alpha)/x,$$

for $0 < \alpha < \beta < 1/2$*, exists and equals*

$$\frac{1}{2}\int_\alpha^\beta\int_\alpha^\beta \rho\left(\frac{1-\lambda_1-\lambda_2}{\alpha}\right)\frac{d\lambda_1}{\lambda_1}\frac{d\lambda_2}{\lambda_2}.$$

Define $\sigma_2(u,v) = G_2(1/u, 1/v)$, extending the function σ to the double prime case. An effective method of computing $\sigma_2(u,v)$ is given in the next section.

4.4 Effective Calculation of $\sigma_2(u,v)$

Bach and Peralta have presented effective calculation procedures for the Dickman ρ function, based on the work of Patterson and Rumsey (reported in [2]), and also for the σ function which they have defined. Here we give a similar effective computation for σ_2.

The ρ function is piecewise analytic between the integers [2]: there is an analytic function $\rho_k(x)$ which equals $\rho(x)$ for $x \in [k-1, k]$. In Taylor series form,

$$\rho_k(k-\xi) = \sum_{i=0}^{\infty} c_i^{(k)}\xi^i.$$

Since $\rho_1(x)$ is identically 1, and since $\rho_2(x) = 1 - \log(x)$, the coefficients $c_i^{(k)}$ are determined for $k = 1$ or 2. The Taylor series coefficients, $c_i^{(k)}$ for $k > 2$, can be computed recursively [2]. For $i > 0$, $k > 2$,

$$c_i^{(k)} = \sum_{j=0}^{i-1} \frac{c_j^{(k-1)}}{ik^{i-j}};$$

for $i = 0$,

$$c_0^{(k)} = \frac{1}{k-1} \sum_{j=1}^{\infty} \frac{c_j^{(k)}}{j+1}.$$

To compute σ_2, the regions of integration are broken up into the various ρ_k, and the series is integrated term by term. The boundaries between the ρ_k regions are lines parallel to $\lambda_1 + \lambda_2 = 1$, inside the square from α to β, so the regions of integration are much simpler if the coordinates are rotated 45 degrees. Making the substitutions $\mu_1 = \lambda_1 + \lambda_2$, $\mu_2 = \lambda_2 - \lambda_1$, then $G_2(\alpha, \beta)$ separates into left and right terms:

$$\begin{aligned} G_2(\alpha,\beta) &= 2\int_{2\alpha}^{\alpha+\beta}\int_{-(\mu_1-2\alpha)}^{\mu_1-2\alpha} \rho\left(\frac{1-\mu_1}{\alpha}\right)\frac{1}{(\mu_1-\mu_2)(\mu_2+\mu_1)}\,d\mu_2\,d\mu_1 \\ &+ 2\int_{\alpha+\beta}^{2\beta}\int_{-(2\beta-\mu_1)}^{2\beta-\mu_1} \rho\left(\frac{1-\mu_1}{\alpha}\right)\frac{1}{(\mu_1-\mu_2)(\mu_2+\mu_1)}\,d\mu_2\,d\mu_1, \end{aligned}$$

the regions of ρ_k now corresponding simply to intervals of μ_1. Let $k = \lceil (1-\mu_1)/\alpha \rceil$; then the error is $\xi(\mu_1) = k - (1-\mu_1)/\alpha$. Therefore, for an interval of $\mu_1 \in [\eta, \theta]$ such that k is constant, the left term of $G_2(\alpha,\beta)/2$ becomes

$$\begin{aligned} G_2^{left}(\alpha,\beta)/2 &= \int_{\eta}^{\theta}\int_{-(\mu_1-2\alpha)}^{\mu_1-2\alpha} \rho(k-\xi(\mu_1))\frac{1}{\mu_1^2-\mu_2^2}\,d\mu_2\,d\mu_1 \\ &= \sum_{i=0}^{\infty} c_i^{(k)} \int_{\eta}^{\theta}\int_{-(\mu_1-2\alpha)}^{\mu_1-2\alpha} \xi(\mu_1)^i \frac{1}{\mu_1^2-\mu_2^2}\,d\mu_2\,d\mu_1 \\ &= \sum_{i=0}^{\infty} c_i^{(k)} \int_{\eta}^{\theta}\int_{-(\mu_1-2\alpha)}^{\mu_1-2\alpha} \left(k - \frac{1-\mu_1}{\alpha}\right)^i \frac{1}{\mu_1^2-\mu_2^2}\,d\mu_2\,d\mu_1 \\ &= \sum_{i=0}^{\infty} \frac{c_i^{(k)}}{\alpha^i} \int_{\eta}^{\theta} ((k\alpha-1)+\mu_1)^i \left(\frac{\log(\mu_1-\alpha)-\log(\alpha)}{\mu_1}\right) d\mu_1 \\ &= \sum_{i=0}^{\infty} \frac{c_i^{(k)}}{\alpha^i} \int_{\eta}^{\theta} \sum_{j=0}^{i} \binom{i}{j} (k\alpha-1)^{i-j}\mu_1^{j-1}(\log(\mu_1-\alpha)-\log(\alpha))d\mu_1 \end{aligned}$$

$$= \sum_{i=0}^{\infty} \frac{c_i^{(k)}}{\alpha^i} \Big\{ (k\alpha - 1)^i \Big[\log(\mu_1 - \alpha)\log(\frac{\mu_1}{\alpha}) + \operatorname{dilog}(\frac{\mu_1}{\alpha}) - \log(\mu_1)\log(\alpha) \Big]_\eta^\theta$$
$$+ \sum_{j=1}^{i} \binom{i}{j} (k\alpha - 1)^{i-j} \int_\eta^\theta \mu_1^{j-1} (\log(\mu_1 - \alpha) - \log(\alpha)) d\mu_1 \Big\},$$

where $\operatorname{dilog}(x) = \int_{t=1}^{x} \log(t)/(1-t)$. For the remaining integral, substitute $\nu = \mu_1 - \alpha$:

$$\int_\eta^\theta \mu_1^{j-1} (\log(\mu_1 - \alpha) - \log(\alpha)) d\mu_1$$
$$= \int_{\eta-\alpha}^{\theta-\alpha} (\alpha + \nu)^{j-1} (\log(\nu) - \log(\alpha)) d\nu$$
$$= \sum_{\ell=0}^{j-1} \binom{j-1}{\ell} \alpha^{j-1-\ell} \int_{\eta-\alpha}^{\theta-\alpha} \nu^\ell (\log(\nu) - \log(\alpha)) d\nu$$
$$= \sum_{\ell=0}^{j-1} \binom{j-1}{\ell} \alpha^{j-(\ell+1)} \left[\frac{\nu^{\ell+1}((\ell+1)\log(\nu) - (\ell+1)\log(\alpha) - 1)}{(\ell+1)^2} \right]_{\eta-\alpha}^{\theta-\alpha}$$
$$= \sum_{\ell=1}^{j} \binom{j-1}{\ell-1} \alpha^{j-\ell} \left[\frac{\nu^\ell(\ell\log(\nu) - \ell\log(\alpha) - 1)}{\ell^2} \right]_{\eta-\alpha}^{\theta-\alpha}.$$

Then finally,

$$G_2^{left}(\alpha, \beta)/2$$
$$= \sum_{i=0}^{\infty} \frac{c_i^{(k)}}{\alpha^i} \Big\{ (k\alpha - 1)^i \Big[\log(x - \alpha)\log(\frac{x}{\alpha}) + \operatorname{dilog}(\frac{x}{\alpha}) - \log(x)\log(\alpha) \Big]_{x=\eta}^{x=\theta}$$
$$+ \sum_{j=1}^{i} \binom{i}{j} (k\alpha - 1)^{i-j} \sum_{\ell=1}^{j} \binom{j-1}{\ell-1} \frac{\alpha^{j-\ell}}{\ell^2} [x^\ell(\ell\log(x) - \ell\log(\alpha) - 1)]_{x=\eta-\alpha}^{x=\theta-\alpha} \Big\}.$$

The right term of $G_2(\alpha, \beta)/2$ is of the same form:

$$G_2^{right}(\alpha, \beta)/2$$
$$= \sum_{i=0}^{\infty} \frac{c_i^{(k)}}{\alpha^i} \Big\{ (k\alpha - 1)^i \Big[\log(x - \beta)\log(\frac{x}{\beta}) + \operatorname{dilog}(\frac{x}{\beta}) - \log(x)\log(\beta) \Big]_{x=\theta}^{x=\eta}$$
$$+ \sum_{j=1}^{i} \binom{i}{j} (k\alpha - 1)^{i-j} \sum_{\ell=1}^{j} \binom{j-1}{\ell-1} \frac{\beta^{j-\ell}}{\ell^2} [x^\ell(\ell\log(x) - \ell\log(\beta) - 1)]_{x=\theta-\beta}^{x=\eta-\beta} \Big\}.$$

The terms $c_i^{(k)}$ decrease exponentially, $c_i^{(k)} \leq (1/2)^i$ for $k \geq 2$ [2]. The expressions above will be used by truncating the first summations. Using the fact that $\xi \leq 1$ to bound the tail of the left expression,

$$\begin{aligned}
G_2^{\substack{left\\tail}}(\alpha,\beta)/2 &= \sum_{i=n+1}^{\infty} c_i^{(k)} \int_\eta^\theta \int_{-(\mu_1-2\alpha)}^{\mu_1-2\alpha} \xi(\mu_1)^i \frac{1}{\mu_1^2-\mu_2^2}\, d\mu_2\, d\mu_1 \\
&\leq \sum_{i=n+1}^{\infty} c_i^{(k)} \int_\eta^\theta \int_{-(\mu_1-2\alpha)}^{\mu_1-2\alpha} \frac{1}{\mu_1^2-\mu_2^2}\, d\mu_2\, d\mu_1 \\
&= \sum_{i=n+1}^{\infty} c_i^{(k)} \int_\eta^\theta \frac{\log(\mu_1-\alpha)-\log(\alpha)}{\mu_1} d\mu_1 \\
&\leq \sum_{i=n+1}^{\infty} (1/2)^i \left[\log(x-\alpha)\log(\frac{x}{\alpha}) + \operatorname{dilog}(\frac{x}{\alpha}) - \log(x)\log(\alpha)\right]_{x=\eta}^{x=\theta} \\
&= \frac{1}{2^n}\left[\log(x-\alpha)\log(\frac{x}{\alpha}) + \operatorname{dilog}(\frac{x}{\alpha}) - \log(x)\log(\alpha)\right]_{x=\eta}^{x=\theta}.
\end{aligned}$$

Similarly,

$$G_2^{\substack{right\\tail}}(\alpha,\beta)/2 \leq \frac{1}{2^n}\left[\log(x-\beta)\log(\frac{x}{\beta}) + \operatorname{dilog}(\frac{x}{\beta}) - \log(x)\log(\beta)\right]_{x=\theta}^{x=\eta}.$$

Since there can be at most $1+\beta/\alpha$ different k values in the left region of integration, and also at most the same number of k values in the right, the total truncation error is $1+\beta/\alpha$ times the sum of the tails above, which decreases exponentially in the number of terms. The G_2 function can therefore be computed effectively by summing a finite number of truncated G_2^{left} series in each left interval where k is constant, plus a similar sum of truncated G_2^{right} series over each right interval with constant k.

The shape of G_2 is shown in figure 4.2. Instead of tables, Maple procedures for ρ, σ (called σ_1 there) and σ_2 are given in appendix 1.

4.5 The Double Large Prime Variation

For the double large prime variation, the primes fall in pairs, and this process can be considered as the process of dropping edges in to a random graph, non-uniformly, with the initial graph consisting of a certain number of non-connected vertices representing the

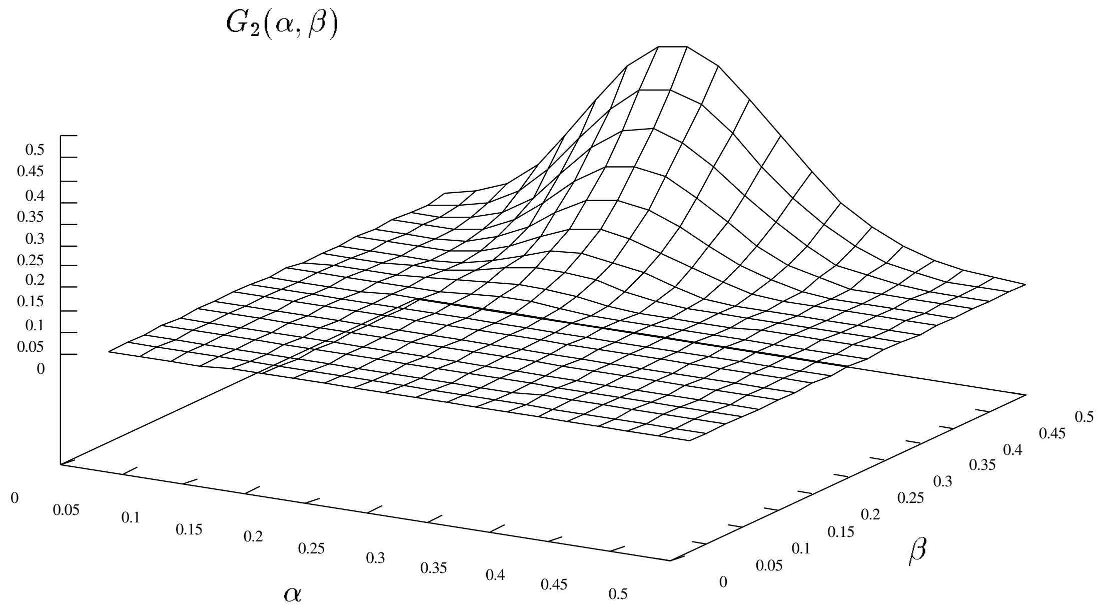

Figure 4.2: The G_2 function.

large primes under consideration. Any cycle found in this graph represents a combination of sieve numbers such that the large primes are eliminated (mod 2), and thus produce a useful relation. Useful relations must be independent of those already obtained, since these relations will be combined to find a dependency. The double large prime can also be combined with the single large primes to form useful relations. These single large prime relations are introduced into the graph by using Lenstra's idea of adding 1 to the graph as a default large prime, and then single large primes will all be connected to this vertex.

The number of components determines the number of independent relations, which is easily seen by examining the addition of a single edge: either the new edge connects two previously unconnected components, generating no new cycle and reducing the number of components by one while increasing the number of edges, or else it falls in a previously connected component, increasing the number of independent cycles by one as well as increasing the number of edges. The number of independent cycles is then $e + c - v$, for e the number of edges, c components, and v vertices, and for our case only the number of components is unknown.

The evolution of random graphs has been extensively studied, but usually under the assumption that the edge probability is uniform. Under certain assumptions on the edge probability being uniformly small, Kovalenko has proved the asymptotic Poisson distribution of the number of completely isolated vertices, and has found a relation between this and the number of components [18]. From this result, it appears that the number relations will again be roughly Poisson distributed, as in the single large prime case, and as such will have large variance.

Therefore, as in the single large prime case, it is not expected that such predictions will be very accurate predictors of actual experiments.

4.6 Summary

Large prime variations have proven to be valuable methods of increasing the number of relations obtained with sieving. Dickman's rho function gives the asymptotic fraction of numbers that are smooth to some bound, and is useful to estimate the number of completely smooth relations that might be obtained. To find the asymptotic fraction of semi-smooth numbers — that is, those numbers that are completely smooth except for one large prime factor — Bach and Peralta have extended Dickman's rho function.

In applying these functions to estimate the number of relations generated with large prime variation, we have restricted ourselves to the quadratic sieve. For the single large prime relation, we have employed methods of cell occupancy and Poisson approximation to show, given the assumption that the numbers tested for smoothness are chosen randomly, that the number of relations tends to a Poisson distribution: a distribution having variance equal to the mean.

For the double large prime variation, another extension is required, and we have defined a function that yields the asymptotic fraction of numbers that are smooth except for two large prime factors, and have given an effective computation for this quantity. For the double large prime variation, some results on random graphs suggests that the number of relations obtained also tends to a Poisson random variable.

Therefore, if these methods are to be modelled as random processes, then the estimates obtained cannot be expected to match experiments closely, and thus these estimates may not be very useful in the design of sieve methods.

Chapter 5

Conclusions

The linear system solution of the second stage of discrete logarithm calculation involves the solution of a large sparse matrix, in exact arithmetic, over large primes. In chapter 2, the usual methods for linear system solution were examined for the large prime case, and their performance for solution modulo large primes was evaluated. The orthogonal Lanczos algorithm, typical in numerical analysis, was adapted to solution over large primes. The conjugate gradient method was concluded to be the most efficient method for the linear solution problem.

Improvements of the Lanczos methods were given, reducing the memory required by the conjugate Lanczos algorithm. Using the bi-diagonalization of Golub and Kahan, a linear system solution method was presented in chapter 2. This method requires 50% less memory than the conjugate Lanczos algorithm as previously presented, 25% less memory than the conjugate gradient method, and no more computation than either of these methods. It is expected that this method will become the method of choice for solution modulo large primes.

In chapter 3, the question of breakdown was examined in detail, for the general case of non-symmetric, possibly non-singular matrices. New and efficient lookahead methods were derived for Lanczos methods, requiring only constant additional memory to handle lookahead of arbitrary degree, and this only during lookahead itself. The conjugate gradient method was shown to consist of two coupled Lanczos algorithms, and an integrated lookahead method for both halves of the coupled processes was derived.

This chapter also succeeded in unifying the theory of the Lanczos, conjugate gradient and Wiedemann algorithms, using orthogonal polynomials, and in proving an analogue of Taylor's mismatch theorem for the Lanczos algorithm over general fields. This implies that the occurrence of incurable breakdown in fact provided an equivalent single pass of the Wiedemann algorithm, and therefore can be used to advance the linear system solution. Hybrid solution methods were thereby proposed, retaining both the efficiency of the Lanczos and conjugate gradient algorithms and the breakdown-free properties of the Wiedemann algorithm. Efficient hybrid algorithms were presented, requiring only that a few scalar parameters be stored at each iteration, to be recalled in order to reconstruct the factor of the irreducible polynomial to circumvent incurable breakdown.

Given a randomness assumption, the likelihood and the degree of breakdown were derived.

Chapter 4 considered the number of relations that can be obtained via large prime variations. Using asymptotic estimates of the number of smooth, semi-smooth and doubly-semi-smooth numbers, and methods of cell occupancy and Poisson approximation, it was shown that, given an assumption on the random generation of the large prime relations, the number of useful relations derived from combining tends to a Poisson random variable. This implies that the variance of the estimate will be large.

A result on random graphs implies that this is also the case for the double large prime variation. Hence, such estimates will likely be of limited use.

5.1 Implications for Future Work

The linear system methods considered here have focussed on solution over large primes. Because there is no apparent advantage in this case, the techniques of blocking derived by Coppersmith and Montgomery have not been explored. The importance of block methods to factoring comes from the fact that, for factoring, solution modulo 2 is all that is required. The simple framework presented in chapter 3, from which we derived our lookahead methods, seems ideal for deriving block lookahead methods, if there is some advantage to be gained from this. In addition, the compact derivation of the conjugate gradient method as a coupled pair of Lanczos processes immediately implies that block methods, keeping subspaces instead of vectors for each of the coupled recursions, can be given for the conjugate gradient method as well. Since the conjugate gradient algorithm has lower memory requirements than Lanczos methods, this should be explored, and may prove profitable for linear solution in factoring algorithms.

The lookahead methods in chapter 3 are also connected to the solution of the Berlekamp-Massey algorithm, as has been shown there. These lookahead methods can therefore be employed in the same application as the Berlekamp-Massey algorithm, that is, the decoding of cyclic error-correcting codes.

Appendix A

Maple Routines for ρ, σ_1, and σ_2

```
#
# Maple routines for calculating quantities
# for the extension to 2 primes of Bach and Peralta's
# sigma function
#

Digits       := 15: # number of digits of precision
numcoeffrho  := 22: # number of coefficients in truncated rho series
numcoeffrhok := 22: # number of coefficients in truncated rho_k func
numcoeffj    := 12: # number of coefficients in truncated j series
numcoeffg    := 12: # number of coefficients in truncated g series
#
# i^th taylor series term of the k^th piece of the Dickman rho
# c^(k)_i in Bach and Peralta.
#
dickmantaylor := proc(k,i)
local j, value;
option remember;
if (k = 1) then
    if (i = 0) then
        value := 1;
    else
        value := 0;
    fi;
elif (k = 2) then
    if (i = 0) then
        value := evalf(1-log(2));
    else
        value := evalf(1/(i*2^i));
    fi;
else
```

```
    value := 0;
    if (i = 0) then
        for j from 1 to numcoeffrho do
            value := value + dickmantaylor(k,j)/(j+1);
        od;
        value := value / (k-1);
    else
        for j from 0 to i - 1 do
            value := value + dickmantaylor(k-1,j)/(i*k^(i-j));
        od;
    fi;
fi;
value;
end;

#
# truncated taylor series approximation to the k^th piece
# of the Dickman rho function, evaluated at k-xi
#
dickmanpiece := proc(k,xi)
local i, summer;
if (k = 1) then
    summer := 1;
else
    summer := 0;
    for i from numcoeffrhok by -1 to 0 do
        summer := summer * xi + dickmantaylor(k,i);
    od;
fi;
summer;
end;

#
# approximate Dickman rho function
#
dickman := proc(x)
local ceilx;
ceilx := ceil(x);
dickmanpiece(ceilx, ceilx - x);
end;

#
# Bach and Peralta’s H_i(u,v,w) (k implicit)
#
Hfunction := proc(i,u,v,w,k)
local A,B,C, j, divpoly, r;
divpoly := 0;
```

```
for j from 1 to i do
    divpoly := divpoly + r^j/j;
od;
A := w/v+k-w;
B := w/u+k-w;
C := k-w;
C^i * (ln(u/v) + subs(r=A/C, divpoly) - subs(r=B/C, divpoly));
end;

#
# truncated seried expression for Bach and Peralta's
# J(u,v,w) (k implicit) (eq. (4.6)), for xi(t) in [0,1]
#
Jtruncsum := proc(u,v,w,k)
local i, summer;
summer := 0;
for i from numcoeffj by -1 to 0 do
    summer := summer + dickmantaylor(k,i) * Hfunction(i,u,v,w,k);
od;
summer;
end;

#
# Bach and Peralta's J(u,v,w) (k implicit)
# splits J(u,v,w) and applies Jtruncsum above.
#
Jfunction := proc(u,v,w)
local k, val;
k := ceil(w-w/u);
if v < w/(w-k+1) then
    # must split the interval
    Jfunction(w/(w-k+1), v, w) + Jfunction(u, w/(w-k+1), w);
else
    Jtruncsum(u,v,w,k);
fi;
end;

#
# Bach and Peralta's generalization of Dickman's
# splits J(u,v,w) and applies Jtruncsum above.
#
sigma := proc(u,v)
local summer;
summer := dickman(u);
if (u>v) then
    summer := summer + evalf(Jfunction(u,v,u));
fi;
```

```
summer;
end;
sigma1 := proc(u,v)
local summer;
summer := 0;
if (u>v) then
    summer := summer + evalf(Jfunction(u,v,u));
fi;
summer;
end;

rho := <dickman(x)|x>;
g1 := <sigma1(1/x,1/y)|x,y>;

#
# functions which extend to 2 large primes,
# ending with definition of sigma2 function
#

term0 := proc(v,low,high)
local x, f;
f := log(x-v)*log(x/v)+dilog(x/v)-log(x)*log(v);
evalf(subs(x=high,f)-subs(x=low,f));
end;

termnot0 := proc(v,l,low,high)
local x, f;
f := x^l*(l*log(x)-l*log(v)-1);
evalf(subs(x=high,f)-subs(x=low,f));
end;

lsum := proc(v,j,low,high)
local l, summer;
summer := 0;
for l from 1 to j do
    summer := summer +
      binomial(j-1,l-1)*v^(j-l)/l^2*termnot0(v,l,low,high);
od;
evalf(summer);
end;

jsum := proc(a,v,i,k,low,high)
local j, summer;
summer := 0;
for j from 1 to i do
    summer := summer + binomial(i,j)*(k*a-1)^(i-j)*lsum(v,j,low,high);
od;
```

```
evalf(summer);
end;

g2left := proc(k,a,b,low,high)
local i, summer, term0val, sbp, x, intv;
summer := 0;
if high > low then
    term0val := term0(a,low,high);
    for i from 0 to numcoeffg do
        sbp := (k*a-1)^i * term0val + jsum(a,a,i,k,low-a,high-a);
        if (sbp < 0) then
            sbp := 0;
        fi;
        summer := summer + dickmantaylor(k,i)/a^i * sbp;
    od;
fi;
2*summer;
end;

g2right := proc(k,a,b,low,high)
local i, summer, term0val, sbp, x, intv;
summer := 0;
if high > low then
    term0val := term0(b,high,low);
    for i from 0 to numcoeffg do
        sbp := (k*a-1)^i * term0val + jsum(a,b,i,k,high-b,low-b);
        if (sbp < 0) then
            sbp := 0;
        fi;
        summer := summer + dickmantaylor(k,i)/a^i * sbp;
    od;
fi;
2*summer;
end;

#
# g2
#
g2 := proc(a, b)
local k, kl0, kr0, kl1, kl2, kr1, kr2, summer;
kl0 := ceil(1/a) - 2;
kr0 := ceil((1-b)/a) - 1;
kl1 := floor(1/a) - 2;
kl2 := ceil((1-b)/a) - 1;
kr1 := floor((1-b)/a) - 1;
kr2 := ceil((1-2*b)/a);
summer := 0;
```

```
#
# left
#
if (1 - kl1*a > a+b) then
# k doesn't change in left
    summer := summer +
    g2left(kl0,a,b,2*a,a+b);
else
# first odd bit;
    summer := summer +
    g2left(kl0,a,b,2*a,1-kl1*a);
# full pieces;
    for k from kl1 by -1 to kl2 + 1 do
#        print('leftfulls', k, 1-k*a,1-(k-1)*a);
         summer := summer + g2left(k,a,b,1-k*a,1-(k-1)*a);
    od;
# last odd bit;
    summer := summer +
    g2left(kl2,a,b,1-kl2*a,a+b);
fi;

#
# right
#
if (1 - kr1*a > 2*b) then
# k doesn't change in right
    summer := summer +
    g2right(kr0,a,b,a+b,2*b);
else
    # first odd bit;
        summer := summer +
        g2right(kr0,a,b,a+b,1-kr1*a);
    # full pieces;
    for k from kr1 by -1 to kr2 + 1 do
        summer := summer + g2right(k,a,b,1-k*a,1-(k-1)*a);
    od;
    # last odd bit;
    summer := summer +
    g2right(kr2,a,b,1-kr2*a,2*b);
fi;
end;

#
# sigma2
#
sigma2 := <g2(1/x,1/y)|x,y>;
```

Bibliography

[1] E. Bach, *Discrete logarithms and factoring*, Tech. Report UCB/CSD/84/186, Computer Science Division, University of California, Berkeley, 1984.

[2] E. Bach and R. Peralta, *Asymptotic semi-smoothness probabilities*, to appear, Math. Comp., 1996.

[3] A. D. Barbour, L. Holst, and S. Janson, *Poisson Approximation*, Clarendon Press: Oxford University Press, Oxford, New York, 1992.

[4] D. Coppersmith, A. M. Odlyzko, and R. Schroeppel, *Discrete logarithms in $GF(p)$*, Algorithmica **1** (1986), 1–15.

[5] J. K. Cullum and R. A. Willoughby, *Lanczos Algorithms for Large Symmetric Eigenvalue Computations*, vol. 1, Theory, Birkhäuser, Boston, Basel, Stuttgart, 1985.

[6] K. Dickman, *On the frequency of numbers containing prime factors of a certain relative magnitude*, Ark. Mat., Astronomi och Fysik **22A** (1930), 1–13.

[7] R. W. Freund, M. H. Gutknecht, and N. M. Nachtigal, *An implementation of the look-ahead Lanczos algorithm for non-Hermitian matrices*, SIAM J. Comput. **14** (1993), 137–158.

[8] J. L. Gerver, *Factoring large numbers with a quadratic sieve*, Math. Comp. **41** (1983), 287–294.

[9] G. Golub and W. Kahan, *Calculating the singular values and pseudo-inverse of a matrix*, SIAM J. Numer. Anal. Ser. B. **2** (1965), no. 2, 205–224.

[10] G. H. Golub and J. M. Ortega, *Scientific Computing*, Academic Press, San Diego and London, 1993.

[11] G. H. Golub and C. F. van Loan, *Matrix Computations*, second ed., Johns Hopkins University Press, Baltimore and London, 1989.

[12] I. N. Herstein, *Topics in Algebra*, Blaisdell Publishing Company, Waltham, Mass., Toronto, London, 1964.

[13] M. R. Hestenes and E. Stiefel, *Method of conjugate gradients for solving linear systems*, J. Res. Nat. Bur. Standards (1952), 409–436.

[14] A. S. Householder, *The Theory of Matrices in Numerical Analysis*, Dover Publications, New York, 1975.

[15] T. Huckle, *Low-rank modification of the unsymmetric Lanczos algorithm*, Math. Comp. **64** (1995), no. 212, 1577–1588.

[16] D. E. Knuth and L. Trabb Pardo, *Analysis of a simple factorization algorithm*, Theoretical Computer Science **3** (1976), 321–348.

[17] V. F. Kolchin, B. A. Sevast'yanov, and V. P. Chistyakov, *Random Allocations*, V. H. Winston & Sons/John Wiley & Sons, Washington D.C./ New York, 1966, Translation editor A. V. Balakrishnan.

[18] I. N. Kovalenko, *Theoy of random graphs*, Kibernetika **4** (1970), 575–579.

[19] B. A. LaMacchia and A. M. Odlyzko, *Solving large sparse linear systems over finite fields*, Advances in Cryptology–CRYPTO '90 Proceedings (S. Vanstone, ed.), Lecture Notes in Computer Science, vol. 537, Springer-Verlag, 1990, pp. 109–133.

[20] C. Lanczos, *Solution of systems of linear equations by minimized iterations*, J. Res. Nat. Bur. Standards **49** (1952), 33–53.

[21] A. K. Lenstra and M. S. Manasse, *Factoring with two large primes*, Advances in Cryptology - EUROCRYPT'90 (New York) (I. Damgard, ed.), Lecture Notes in Computer Science, Springer-Verlag, 1990, pp. 72–82.

[22] J. L. Massey, *Shift-register synthesis and BCH decoding*, IEEE Trans. Inform. Theory **IT-15** (1969), 122–127.

[23] P. L. Montgomery, *A block Lanczos algorithm for finding dependencies over GF(2)*, Advances in Cryptology - EUROCRYPT'95 (New York) (Louis C. Guillou and Jean-Jacques Quisquater, eds.), Lecture Notes in Computer Science, volume 921, Springer-Verlag, 1995, pp. 106–120.

[24] A. M. Odlyzko, *Discrete logarithms in finite fields and their cryptographic significance*, Advances in Cryptology–Proceedings of EUROCRYPT 84 (T. Beth, N. Cot, and I. Ingemarsson, eds.), Lecture Notes in Computer Science, vol. 209, Springer-Verlag, 1984, pp. 224–314.

[25] C. C. Paige, *Computational variants of the Lanczos method for the eigenproblem*, J. Inst. Math. Applics. **10** (1972), 373–381.

[26] ______, *Bidiagonalization of matrices and solution of linear equations*, SIAM J. Numer. Anal. **11** (1974), no. 1, 197–209.

[27] B. N. Parlett, D. R. Taylor, and Z. A. Liu, *A look-ahead Lanczos algorithm for unsymmetric matrices*, Math. Comp. **44** (1985), no. 169, 105–124.

[28] S. C. Pohlig and M. E. Hellman, *An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance*, IEEE Trans. Inform. Theory **IT-24** (1978), 106–110.

[29] C. Pomerance, *Fast, rigorous factorization and discrete logarithm algorithms*, Proc. of the Japan-US Joint Seminar, June 4, 1986, Kyoto, Japan, Academic Press, 1987, pp. 119–143.

[30] ______, *Factoring*, Cyptology and Computational Number Theory (C. Pomerance, ed.), Proceedings of symposia in applied mathematics, vol. 42, American mathematical society, 1990, pp. 27–47.

[31] C. Pomerance and J. W. Smith, *Reduction of huge, sparse matrices over finite fields via created catastrophes*, Experimental Mathematics **1** (1992), 89–94.

[32] J. Rissanen, *Recursive identification of linear systems*, SIAM J. Control **9** (1971), no. 3, 420–430.

[33] Y. Saad, *The Lanczos biorthogonalization algorithm and other oblique projection methods for solving large unsymmetric systems*, SIAM J. Numer. Anal. **19** (1982), no. 3, 485–506.

[34] O. Schirokauer, *Discrete logarithms and local units*, Proceedings of the Royal Society **345** (1993), 409–423.

[35] Jeremy Teitelbaum, *On the Lanczos method over finite fields*, preprint, 1996.

[36] D. H. Wiedemann, *Solving sparse linear equations over finite fields*, IEEE Trans. Inform. Theory **IT-32** (1986), no. 1, 54–62.

[37] Q. Ye, *A breakdown-free variation of the nonsymmetric Lanczos algorithms*, Math. Comp. **62** (1994), no. 205, 179–207.