**Programming Research Group**

# A NEW SPARSE GAUSSIAN ELIMINATION ALGORITHM AND THE NIEDERREITER LINEAR SYSTEM FOR TRINOMIALS OVER $F_2$

Fatima Abu Salem

PRG-RR-03-17

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD

**Abstract**

An important factorization algorithm for polynomials over finite fields was developed by Niederreiter. The factorization problem is reduced to solving a linear system over the finite field in question, and the solutions are used to produce the complete factorization of the polynomial into irreducibles. One charactersistic feature of the linear system arising in the Niederreiter algorithm is the fact that, if the polynomial to be factorized is sparse, then so is the Niederreiter matrix associated with it. In this paper, we investigate the special case of factoring trinomials over the binary field. We develop a new algorithm for solving the linear system using sparse Gaussian elmination with the Markowitz ordering strategy. Implementing the new algorithm to solve the Niederreiter linear system for trinomials over $F_2$ suggests that, the system is not only initially sparse, but also preserves its sparsity throughout the Gaussian elimination phase. When used with other methods for extracting the irreducible factors using a basis for the solution set, the resulting algorithm provides a more memory efficient and sometimes faster sequential alternative for achieving high degree trinomial factorizations over $F_2$. [1]

# 1 Introduction And Background

Sparse linear systems arise in a variety of applications in chemical engineering, linear programming, and integer factorization algorithms. A recent application extends to polynomial factorization over $F_2$ using the Niederreiter algorithm, where the system to be solved is sparse if the polynomial to be factorized is sparse. Various methods have been used to solve the Niederreiter linear system using explicit and dense linear algebra in [17] and implicit linear algebra in [6]. While those implementations have achieved factorization records for polynomials over $F_2$, confirming the argument that the Niederreiter algorithm is the most efficient linear algebra based algorithm for the factorization of such polynomials (see [2, 13, 14, 15, 16]), we attempt to investigate the sparsity feature of the algorithm which renders it more efficient for sparse polynomials, and in particular, trinomials. Trinomials are very sparse and hence provide a good model for investigating this aspect of the Niederreiter algorithm. In this paper, we prove the argument that the Niederreiter matrix is sparse in the case of a trinomial, and establish the exact sparsity pattern and density of the Niederreiter matrix. We also develop a new algorithm for solving the sparse linear system directly to produce a basis for the solution set through Gaussian elimination and using the data structure of Gustavson [10]. The new algorithm is mainly aimed at circumventing the problems that have always been associated with this data structure in terms of elbow space and compression (see Section 5). Although it can be easily modified to solve general sparse linear systems over $F_2$, the algorithm proves to be very efficient when the matrix maintains a high level of sparsity throughout the reduction phase. Our experimental results confirm that the Niederreiter matrix is initially sparse, and maintains its sparsity throughout the reduction phase. Our results can be incorporated into any algorithm for extracting the factors using a basis for the

[1]Key-words: finite fields, polynomial factorization, Niederreiter algorithm, sparse Gaussian Elimination.

solution set. In particular, they can be used in Göttfert's acceleration of the Niederreiter algorithm over $F_2$ which uses all the elements of the basis set [9].

In section 2 we give a brief summary of the Niederreiter algorithm over the binary field. In section 3, we prove the assumption that the Niederreiter matrix is sparse for trinomials, and establish that the initial number of entries in the sparse matrix of dimension $d \times d$ does not exceed $3d$. In section 4 we review some of the widely used data structures for storing sparse matrices. In section 5 we discuss the new algorithm for performing sparse Gaussian elmination with pivotal ordering using the Markowitz strategy. In section 6, we report on our experimental results, from which we conjecture that the Niederreiter linear system maintains a high percentage of sparsity throughout the Gaussian elimination phase.

## 2 The Niederreiter Algorithm over the Binary Field

Let $F_2$ be the binary field of order 2 consisting only of the elements $0, 1$; it is thus understood that all polynomials described in this paper are monic. Let $f$ be a polynomial of degree $d$ over $F_2$, and $f = g_1^{e_1}...g_m^{e_m}$ be its canonical factorization of over the field. The Niederreiter algorithm reduces the factorization problem into an equivalent one solving the differential equation

$$(fh)' = h^2. \tag{1}$$

The solutions $h$ of eq. (1) form a linear subspace $L(f)$ of the vector space $F_2[x]$ of dimension $m$ over $F_2$. If $b$ denotes a monic factor of the square-free part of $f$, then we have $2^m$ possible such factors, and the solutions $h$ of (1) are given by the polynomials $h$ of the form $h = (f/b).b'$ for each $b$. It follows that $L(f)$ has dimension $m$ over $F_2$ and that $h$ has degree less than $d$ over $F_q$. In addition, since $b$ is square-free, $\gcd(b, b') = 1$ (e.g. see [8] for proof), and we have

$$\gcd(f, h) = \frac{f}{b}\gcd(b, b') = \frac{f}{b}$$

so that $b = f/\gcd(f, h)$. Write $h = h_0 + h_1 x + ... + h_{d-1}x^{d-1}$ with $h_i \in F_2$. Since we are working over the binary field, both $(fh)'$ and $h^2$ are polynomials in $x^2$. Comparing coefficients of the two polynomials in $x^2$ on both sides of (1) results in a system of $d$ equations in $h_i$, for $i = 0, ..., d-1$. If we denote the matrix of coefficients of $f$ by $N_f$, the system has the form

$$(N_f)(h_0, ..., h_{d-1})^T = (h_0^2, ..., h_{d-1}^2)^T$$

where $N_f$ is a $d \times d$ matrix over $F_q$. But we have $h_i^2 = h_i$ for $h_i \in F_2$ and so the system can be rewritten as

$$(N_f - I_d)\mathbf{h}^T = \mathbf{0}, \tag{2}$$

where $I_d$ is the $d \times d$ identity matrix over $F_2$ and $\mathbf{h} = (h_0, ..., h_{d-1})$. It can also be shown that $m = d - rank(N_f - I_{d\times d})$, so that the number of irreducible factors can be found once the linear system is solved. The equivalence between (1) and (2) reduces the factorization problem into solving a linear system over $F_2$, hence its classification as a *linear alegbra* based factorization algorithm.

For all details and proofs appearing in this section we refer the reader to [13, 14, 15]. For more extensive reviews and work on the algorithm, we refer the reader to [5, 7, 11, 16].

## 3 Setting the Niederreiter matrix over $F_2$

In [13], Niederreiter proposes that, due to the simple form of Eq. (1), the system (2) has the form:

$$\sum_{k=\max(2j+1-d,0)}^{\min(2j+1,d-1)} f_{2j+1-k}h_k = h_j \text{ for } 0 \leq j \leq d-1. \tag{3}$$

The proof of this assertion can be sketched as follows:

First, we know that $h^2(x) = h(x^2)$ in $F_2[x]$ and so the coefficients of $x^{2j}$ in the right-hand side of Eq. 1 are given by $h_j$ for $0 \leq j \leq d-1$. On the other hand, recall that the matrix $N_f$ is obtained by comparing the coefficients of $x^{2j}$ for $0 \leq j \leq d-1$ in both $(fh)'$ and $h^2$. Let

$$f(x) = \sum_{i=0}^{d} f_i x^i$$

and

$$h(x) = \sum_{k=0}^{d-1} h_k x^k.$$

The coefficient of $x^{2j}$ in $(fh)'$ is the coefficient of $x^{2j+1}$ in $(fh)$, and so the coefficients of $(fh)'$ can be expressed as

$$\sum_{k=\max(2j+1-d,0)}^{\min(2j+1,d-1)} f_{2j+1-k}h_k$$

for $0 \leq j \leq d-1$. The bounds on $k$ follow because of the following facts:

i. Since $\deg(h) \leq d-1$, $h_k = 0$ for all $k > d-1$ and so $k$ has to satisfy $k \leq d-1$. Also, since $f_{2j+1-k} = 0$ for all $2j+1-k < 0$, we have to maintain $k \leq 2j+1$. As a result, an upper bound for $k$ would be $\min(2j+1, d-1)$.

ii. In a similar way, $h_k = 0$ for $k < 0$ and $f_{2j+1-k} = 0$ for $2j+1-k > d$ (since $\deg(f) = d$). Therefore, $k$ has to satisfy $k \geq 0$ and $k \geq 2j+1-d$ and so a lower bound for $k$ is $\max(2j+1-d)$. $\square$

The above proposition establishes a fixed structure for the Niederreiter matrix from which the algorithm derives many of its attractive features over $F_2$. The following result appears originally in [13] but we provide a detailed proof of it.

**Theorem 1** *Let* $f = f_d x^d + ... + f_1 x + f_0$ *be a polynomial of degree* $d$ *over* $F_2$. *The elements in the Niederreiter matrix* $N_f$ *can be obtained as follows:*
*1. If d is even, then:*
*i.* $f_{2k'}$ *appears in rows* $i = k'+1, ..., (d/2)+k'$ *and occupies column* $2(i-k')$ *in row* $i$, *for* $k' = 0, ..., d/2$.
*ii.* $f_{2k'+1}$ *appears in rows* $i = k'+1, ..., (d/2)+k'$ *and occupies column* $2(i-k')-1$ *in row* $i$, *for* $k' = 0, ..., (d/2)-1$.
*2. If d is odd, then*
*i.* $f_{2k'}$ *appears in rows* $i = k'+1, ..., \frac{d-1}{2}+k'$ *and occupies column* $2(i-k')$ *in row* $i$, *for* $k' = 0, ..., \frac{d-1}{2}$.
*ii.* $f_{2k'+1}$ *appears in rows* $i = k'+1, ..., \frac{d-1}{2}+k'+1$ *and occupies column* $2(i-k')-1$ *in row* $i$, *for* $k' = 0, ..., \frac{d-1}{2}$.
*In other words, the Niederreiter matrix over* $F_2$ *can be written as*

$$\begin{pmatrix} f_1 & f_0 & 0 & 0 & 0 & 0 & 0 & ... & ... & ... & 0 \\ f_3 & f_2 & f_1 & f_0 & 0 & 0 & ... & ... & ... & ... & 0 \\ f_5 & f_4 & f_3 & f_2 & f_1 & f_0 & 0 & ... & ... & ... & 0 \\ ... & ... & ... & ... & ... & ... & ... & ... & ... & ... & ... \\ f_{d-1} & f_{d-2} & ... & ... & ... & ... & ... & ... & ... & ... & f_0 \\ 0 & f_d & f_{d-1} & ... & ... & ... & ... & ... & ... & ... & f_2 \\ 0 & 0 & 0 & f_d & f_{d-1} & ... & ... & ... & ... & ... & f_4 \\ ... & ... & ... & ... & ... & ... & ... & ... & ... & ... & ... \\ 0 & 0 & ... & ... & ... & ... & 0 & 0 & f_d & f_{d-1} & f_{d-2} \\ 0 & 0 & ... & ... & ... & ... & ... & ... & ... & 0 & f_d \end{pmatrix}$$

*if d is even, and*

$$\begin{pmatrix} f_1 & f_0 & 0 & 0 & 0 & 0 & 0 & ... & ... & ... & 0 \\ f_3 & f_2 & f_1 & f_0 & 0 & 0 & ... & ... & ... & ... & 0 \\ f_5 & f_4 & f_3 & f_2 & f_1 & f_0 & 0 & ... & ... & ... & 0 \\ ... & ... & ... & ... & ... & ... & ... & ... & ... & ... & ... \\ f_d & f_{d-1} & ... & ... & ... & ... & ... & ... & ... & ... & f_1 \\ 0 & 0 & f_d & f_{d-1} & ... & ... & ... & ... & ... & ... & f_3 \\ 0 & 0 & 0 & 0 & f_d & f_{d-1} & ... & ... & ... & ... & f_5 \\ ... & ... & ... & ... & ... & ... & ... & ... & ... & ... & ... \\ 0 & 0 & ... & ... & ... & ... & 0 & 0 & f_d & f_{d-1} & f_{d-2} \\ 0 & 0 & ... & ... & ... & ... & ... & ... & ... & 0 & f_d \end{pmatrix}$$

*if d is odd.*

**Proof:**
In the following, $i$ denotes the index of a row in $\{1, ..., d\}$. Let $i = j+1$; eq. (3) says that, for $j = 0, ..., d-1$, row $i$ contains the coefficient $f_{2j+1-k}$ in column $k+1$, for $k = \max(2j+1-d, 0), ..., \min(2j+1, d-1)$. Suppose that $d$ is even. To establish the proof of the theorem, we strive to show that $f_{2k'}$, for $k' = 0, ..., d/2$, and $f_{2k'+1}$, for

$k' = 0, ..., (d/2) - 1$, appear first in row $i = k' + 1$, and last in row $i = (d/2) + k'$, and occupy the columns stated in the theorem above.

For $j = 0, ..., \frac{d}{2} - 1$ we have

$$2j + 1 - d \leq 2\left(\frac{d}{2} - 1\right) + 1 - d = -1 \to \max(2j + 1 - d, 0) = 0. \tag{4}$$

Also,

$$2j + 1 \leq 2\left(\frac{d}{2} - 1\right) + 1 < d - 1 \to \min(2j + 1, d - 1) = 2j + 1. \tag{5}$$

Similarly, for $j = (d/2), ..., d - 1$, we have

$$\max(2j + 1 - d, 0) = 2j + 1 - d \tag{6}$$

and

$$\min(2j + 1, d - 1) = d - 1. \tag{7}$$

To establish the indices of the starting rows of $f_{2k'}$ and $f_{2k'+1}$, we proceed as follows. If $k' = 0, ..., (d/2) - 1$, row $i = k' + 1$ (for which $j = k' \in \{0, ..., (d/2) - 1\}$ contains $f_{2j+1-k}$ in column $k + 1$ for $k = 0, ..., 2j + 1$ (see (4) and (5) above). In particular, columns 1 and 2 contain $f_{2j+1} = f_{2k'+1}$ and $f_{2j} = f_{2k'}$ respectively. If $k' = d/2$, row $i = k' + 1$ (for which $j = d/2$) contains $f_{2j+1-k}$ in column $k + 1$, for $k = 2j + 1 - d, ..., d - 1$ (see (6) and (7)). In particular, column 2 contains $f_{2j+1-(2j+1-d)} = f_d = f_{2k'}$ .

To establish the indices of the terminating rows, we proceed as follows. If $k' = 0$, row $i = (d/2) + k'$ (for which $j = d/2 + k' - 1 = d/2 - 1$) contains $f_{2j+1-k}$ in column $k + 1$ for $k = 0, ..., 2j + 1$ (see (4) and (5)) so that columns $2j + 1 = d - 1$ and $2j + 2 = d$ contain $f_{2j+1-(2j)} = f_1$ and $f_{2j+1-(2j+1)} = f_0$ respectively. If $k' = 1, ..., d/2$, row $i = (d/2) + k'$ (for which $j = (d/2) + k' - 1 \in \{(d/2), ..., d - 1\}$) contains $f_{2j+1-k}$ in column $k + 1$ for $k = 2j + 1 - d, ..., d - 1$ (see (6) and (7)). In particular, columns $d - 1$ and $d$ contain $f_{2j+1-(d-2)} = f_{2j-d+3} = f_{2k'+1}$ and $f_{2j+1-(d-1)} = f_{2j-d+2} = f_{2k'}$ respectively.

Finally, we establish the indices of columns where the non-zero entries appear (hereafter we use the term *entry* to denote a non-zero element of the matrix). We know that each $f_{2j+1-k}$, for $j = 0, ..., d - 1$ occupies column $k + 1$, for $k = \max(2j + 1 - d, 0), ..., \min(2j + 1, d - 1)$. If we write $2j + 1 - k = 2k'$, for some $k' = 0, ..., d/2$, then

$$2j + 1 - k = 2k' \leftrightarrow k = 2j + 1 - 2k' = 2(i - 1) + 1 - 2k' = 2(i - k') - 1$$

so that $f_{2k'}$ occupies column $k + 1 = 2(i - k')$. If we write $2j + 1 - k = 2k' + 1$, for some $k' = 0, ..., (d/2) - 1$, then

$$2j + 1 - k = 2k' + 1 \leftrightarrow k = 2j - 2k' = 2(i - 1) - 2k' = 2(i - k') - 2,$$

so that $f_{2k'+1}$ occupies columns $k + 1 = 2(i - k') - 1$.

If $d$ is odd, we argue similarly as above using the following facts:

i. if $j = 0, ..., \frac{d-3}{2}$, $\max(2j+1-d, 0) = 0$ and $\min(2j+1, d-1) = 2j+1$.

ii. if $j = \frac{d-1}{2}$, $\max(2j+1-d, 0) = 0$ and $\min(2j+1, d-1) = d-1$.

iii. if $j = \frac{d+1}{2}, ..., \frac{d-1}{2}$, $\max(2j+1-d, 0) = 2j+1-d$ and $\min(2j+1, d-1) = d-1$. $\square$

Theorem 1 is at the basis of the assumption that, the matrix $N_f$ is sparse if the polynomial $f$ is sparse. It also establishes that there is no cost associated with arithmetic operations for setting up the matrix $N_f$, in the sense that the matrix coefficients can be read off immediately from those of $f$. In this paper, we support the main argument that the Niederreiter matrix is sparse by determining exactly the percentage of sparsity of the matrix upon set-up. The following theorem not only describes a consistent pattern of where the entries occur in the matrix $N_f - I$ if $f$ is a trinomial over $F_2$, but also the exact number of entries that initially occur in $N_f - I$. By a simple abuse of notation, we define the length of a row to be the number of non-zero entries appearing in it, rather than its actual length $d$. We then claim the following:

**Theorem 2** *Let $f(x) = f_d x^d + f_s x^s + f_0$ denote a trinomial over $F_2$. Let $M = N_f - I$ where $N_f$ is the $d \times d$ Niederreiter matrix and $I$ the $d \times d$ identity matrix over $F_2$. Then:*

*i. if $d$ is even, the matrix $M$ contains exactly one row of length 0, one row of length 1, $\frac{d}{2} - 1$ rows of length 2, and $\frac{d}{2} - 1$ rows of length 3.*

*ii. if $d$ is odd, the matrix $M$ contains exactly one row of length 0 and one row of length 1. In addition, if $s$ is odd, then $M$ contains exactly $\frac{d-3}{2}$ rows of length 2 and $\frac{d-1}{2}$ rows of length 3; else, if $s$ is even, then $M$ contains $\frac{d-1}{2}$ rows of length 2 and $\frac{d-3}{2}$ rows of length 3.*

**Proof:**

Suppose that $d$ is even and write $i = j+1$. By theorem 1, $f_0$ falls along rows $i = j+1$ of $N_f$, for $j = 0, ..., d/2$, and $f_d$ falls along rows $i = j+1$ of $N_f$, for $j = d/2, ..., d$. Since $f$ is a trinomial, Theorem 1 also implies that the maximum number of entries in any row of $N_f$ is two, those consisting of the pair $(f_s, f_0)$ or $(f_d, f_s)$, and so, the maximum number of entries in any row of $M$ is three. Let

$$Set_0 = \{j : 0 \le j \le d-1 | \text{ row } i \text{ of } M \text{ contains no entries}\},$$
$$Set_1 = \{j : 0 \le j \le d-1 | \text{ row } i \text{ of } M \text{ contains exactly one entry}\},$$
$$Set_2 = \{j : 0 \le j \le d-1 | \text{ row } i \text{ of } M \text{ contains exactly two entries}\},$$
$$Set_3 = \{j : 0 \le j \le d-1 | \text{ row } i \text{ of } M \text{ contains exactly three entries}\}.$$

We aim to show that $\#Set_0 = 1, \#Set_1 = 1, \#Set_2 = d/2 - 1$, and $\#Set_3 = d/2 - 1$. Let

$$R_{f_0} = \{j : 0 \le j < \frac{d}{2} | f_0 \text{ appears in row } i \text{ of matrix } M\},$$
$$R_{f_d} = \{j : \frac{d}{2} \le j < d | f_d \text{ appears in row } i \text{ of matrix } M\},$$
$$R_{f_s} = \{j : 0 \le j < \frac{d}{2} | f_s \text{ appears in row } i \text{ of matrix } M\},$$
$$R'_{f_s} = \{j : \frac{d}{2} \le j < d | f_s \text{ appears in row } i \text{ of matrix } M\}.$$

We first claim that $f_0$ can never appear as a diagonal entry in $N_f$; otherwise, if we write $f_0 = f_{2k}$ for $k = 0$, and since $f_0$ occupies the position $(i, 2(i-k))$ for some $i = 1, .., d/2$ (Theorem 1), we must have $i = 2(i-k) = 2i$, a contradiction, since $i \geq 1$. Thus, when computing row $i$ in $M$ for $i = 1, ..., d/2$, $f_0$ in $N_f$ is never cancelled out by a diagonal entry in $I_{d \times d}$, or equivalently, $||R_{f_0}|| = \frac{d}{2}$. Similarly, if we write $f_d = f_{2k}$ for $k = \frac{d}{2}$, then for $f_d$ to be a diagonal entry, we must have $i = 2(i - \frac{d}{2})$ or that $i = d$. It follows that $f_d$ appears as a diagonal entry of $N_f$ only in the last row $d$, and so never appears in row $d$ of $M$. This implies that

$$||R_{f_d}|| = \frac{d}{2} - 1.$$

Note that $s \neq 0, d$. For $f_s$ to occupy a diagonal entry in $N_f$, Theorem 1 implies that

$$i = 2(i-t) \leftrightarrow i = 2t$$

if $s = 2t$ for $t \geq 1$, or that

$$i = 2(i-t) - 1 \leftrightarrow i = 2t + 1$$

if $s = 2t + 1$, for $t \geq 0$. It follows that $f_s$ is a diagonal entry of $N_f$ (and hence disappears in $M$) only in row $s$, which as such contains either $f_0$ or $f_d$ alongside $f_s$. Also, by Theorem 1, $f_s$ appears in $\frac{d}{2}$ rows of $N_f$. As a result,

$$||R_{f_s}|| + ||R'_{f_s}|| = \frac{d}{2} - 1.$$

Now, since $f_0$ is never a diagonal entry in rows $i = 1, ..., d/2$ of $N_f$, $f_d$ is a diagonal entry only in row $d$ of the last $d/2$ rows, and row $d$ does not contain $f_s$, it follows that $Set_0 = \{d\}$. Since $f_s$ is a diagonal entry of $N_f$ only in row $s \neq 0, d$, which also contains either $f_0$ or $f_d$, $Set_1 = \{s\}$. Also,

$$(R_{f_0} - R_{f_s}) \cup (R_{f_d} - R'_{f_s}) = Set_2 \cup \{i = s\}$$

where $Set_2$ contains the diagonal and one of $f_0$ or $f_d$. Thus,

$$\begin{aligned} ||Set_2|| &= ||R_{f_0} - R_{f_s}|| + ||R_{f_d} - R'_{f_s}|| - 1 \\ &= ||R_{f_0}| + ||R_{f_d}|| - (||R_{f_s}|| + ||R'_{f_s}||) - 1 \\ &= \frac{d}{2} - 1. \end{aligned}$$

Since

$$||Set_0|| + ||Set_1|| + ||Set_2|| + ||Set_3|| = d,$$

we have $||Set_3|| = d/2 - 1$.

If $d$ is odd, a very similar argument holds by noting the following:

i. $f_0$ is found in $\frac{d-1}{2}$ rows of $M$.

ii. $f_d$ is found in $\frac{d+1}{2} - 1$ rows of $M$.

iii. If $s$ is odd, then $f_s$ is found in $\frac{d+1}{2} - 1$ rows of $M$. Else, if $s$ is even, then $f_s$ is found in $\frac{d-1}{2} - 1$ rows of $M$.

The proof thereafter is straighforward and we leave the details to the reader. $\square$

## 4 Data Structures for the Sparse Matrix $M$

In this section we describe briefly the data structures that are widely used to represent sparse matrices. A major review of the subject can be found in [4] (see also [18]). Without loss of generality we assume the matrix to be a square $d \times d$ matrix. Let $\tau$ denote the number of non-zero elements appearing in the sparse matrix, and *entries* refer to the non-zero elements. Initially, it is always convenient to supply the matrix in a *coordinate scheme*, a set of triples containing the value of each entry, together with its row and column indices. Since we are working modulo 2, an entry can only have the value 1, hence our disposal of the data structure and algorithmic details which deal with storing the non-zero values, modifying them, or maintaining the numerical stability of the algorithm. The coordinate scheme is one favoured form of supplying the matrix elements, but not for performing Gaussian elimination, for instance, since this would require easy access to rows and columns. For this, two main data structures can be used. In [10], a data structure which transforms the matrix into a collection of sparse row and column vectors is introduced. For simplicity, we assume that the indexing of arrays starts at 1 (not 0). In the collection of sparse row vectors, and for each row of the matrix, we store a pointer to its starting entry and its length. For each entry in that row, we store its column location. For this, we use two integer arrays (say, $row\_start$ and $row\_length$) of size $d$, and one integer array (say $jcn$) of size $\tau$. The components in each row vector can be ordered or unordered. As such, all entries along a particular row $i$ have indices $s = row\_start[i], ..., row\_start[i] + row\_length[i] - 1$. Setting the row vector representation from the coordinate scheme can be established using $O(\tau) + O(d)$ operations (see [4]). The collection of rows allows access for entries along a particular row, but not along columns. To perform this, a similar structure designating a collection of sparse columns is established, with arrays $col\_start$, $col\_length$, and $irn$ allowing access to the start of each column and information about its length, as well as the row indices of entries found in the column vectors. This data structure requires four integer arrays of size $d$ and two integer arrays of size $\tau$ in total. One main difficulty associated with it arises when a new row becomes longer as a result of row operations introducing new entries. In this case, the current space allocated for the row vector has to be wasted temporarily, so that the new row vector is added to the end of the structure. Subsequently, rows become disordered, and after several such additions requiring what is called *elbow room* at the end of the data structure, one is forced to compress the structure by re-ordering the rows to occupy the free space wasted previously, a process that is known as *compression*. This constitutes the only major disadvantage of the data structure, hence the alternative of linked lists.

A sparse matrix can be represented as a collection of rows, each in a linked list. For each row we store a pointer to its starting entry, say in a $d$ dimensional integer array, $row\_header$. All subsequent elements in the row are forwardly linked by links, stored in a $\tau$ dimensional integer array, $row\_fwd\_link$. The column locations of all entries are stored in a $\tau$ dimensional integer array $jcn$, and an array $value$ as defined above is needed whenever applicable. The fact that the list can be updated without reference to the actual physical location of entries allows proper insertion of new elements upon fill-in without having to have elbow room or perform compression. The list can be ordered or unordered by increasing index of entries. To make insertion or deletion easier and less expensive, the list can be modified into a doubly linked one where entries in a particular row are further linked to their backward neighbors in the list (this made possible through the use of a $\tau$ dimensional integer array $row\_backwd\_link$). Since the collection by rows allows only access to entries within a particular row but not a column, a similar solution as above consists in establishing a collection of the matrix columns as linked lists and storing the row indices of entries found along the columns. This transforms the structure into a two dimensional list, whose elements can be singly or doubly linked, ordered or unordered. However, this scheme results in large memory overheads by associating four integers with each entry if the list is not doubly linked, and six integers otherwise. Curtis and Reid [3] suggested that the arrays $irn$ and $jcn$ can be discarded, so that the negation of the row (column) indices of entries along a particular row (column) are stored in the last link of the row (column). A basic difficulty associated with this data structure is that the integers stored can be as large as $\tau$, in contrast to the corresponding upper bound $d$ associated with the Gustavson's data structure, which allows for the use of half-word storage if the array entries fit in 16-bit (32-bit) computer words. Although the issues of elbow room and compression are circumvented, the initial memory requirements can be larger than that of Gustavson's if the doubly linked list is used, consisting of four arrays of size $d$, and four arrays of size $\tau$.

Summarizing, the advantages and disadvantages of the data structures can be listed as follows. The singly linked lists need the same amount of memory as that required by Gustavson's data structure at the time of set-up, but less memory than the doubly linked lists. However, their use requires extensive searches as entries are added or deleted, and the alternative is at the expense of increasing memory requirements through the use of doubly linked lists, or elbow room and compression through Gustavson's structure. The sequential operational complexities for performing sparse Gaussian elimination using all of the structures above are the same (see [4]), although differences emerge for parallel applications, where the two dimensional unordered double list is of lowest operational complexity (see [18] for a detailed account of complexities). The decision as to which data structure to use becomes a problem-dependent choice which serves the priorities of the application, those being either improvements on running time or savings in memory.

## 5 A New Sparse Gaussian Elimination Algorithm

Solving the linear system constitutes the bottle-neck in the Niederreiter algorithm for very large polynomial degrees. As a result, the challenges in implementing polynomial factorization consist in first, performing the algorithm for as large an input size as feasible, and second, performing it in the fastest possible way. Accordingly, our main interest in this paper is aimed at achieving savings in memory to deal with very large sparse linear systems over $F_2$. These facts, coupled with our interests in factorizing as large a trinomial as our resources can afford, motivated us to consider an effective data structure such as Gustavson's and investigate how the problems of elbow room and compression associated with it can be avoided to yield a more space-efficient data structure. Our new algorithm for performing sparse Gaussian elimination using Gustavson's data structure consists of a series of major sub-tasks, each of which is described in due course, together with its operational complexity. We first give few definitions and notations. Recall that we have to compute the left nullspace of the system (2), and as a result we have to perform row operations consisting in interchanging two rows, adding a multiple of one row to another, and multiplying any row by a non-zero field element. Since we are working over the binary field, our algorithm consists of the first two operations only, where the second operations simplifies to adding one row to another. To further apply the pivotal Markowitz strategy, we also have to perform column interchanges. To establish the representation of the matrix $M$ as a collection of sparse row vectors, we claim that this does not require that the matrix be given in a coordinate structure. In particular, we have

**Theorem 3** *The matrix $M$ for a trinomial over $F_2$ can be represented as a collection of ordered sparse row vectors without the use of a coordinate scheme.*

**Proof:**
By theorem 2, entries along any row of $M$ can be specified according to their column location. This directly provides the information in arrays $row\_start$, $row\_length$, and $jcn$. □

The collection of sparse column vectors can be easily set up by scanning the rows, in what requires $O(\tau)$ operations. We define $A(a' \to b', c' \to d')$ to be the block matrix comprising rows $a'$ to $b'$ and columns $c'$ to $d'$ of some matrix $A$. We let $\gamma_c$ and $\gamma_r$ denote the average number of entries along a non-zero column or row respectively, $M^{(q)}$ the transformed matrix corresponding to $M$ during some stage of Gaussian elimination, and $M'$ the image of $M^{(q)}$ under a transformation which involves any of the following:

1. An interchange $I_c$ of columns.
2. An interchange $I_r$ of rows.
3. Replacing row $i$ with $i + j$ where $j$ is some other row in the matrix. Over $F_2$, this step reduces to a process of adding and/or removing 1's from row $i$ wherever applicable, and hence constitutes a composition of a finite number of transformations of the form $A_e$ and $R_e$, where $A_e$ and $R_e$ represent adding and removing an entry respectively.

The row and column operations in a sparse algorithm appear differently from what can be seen in dense algorithms, where any of these operations is performed as in their literal definition, such that elements of the dense data structure are interchanged whenever rows/columns are so, or added together whenever a row is added to another. In our sparse algorithm, the operations are achieved through a series of changes updating the information in the various arrays describing the data structure, based on the assumption that the transformed matrix is still sparse and hence can be represented by the same structure as that of the original matrix. The sequence according to which the updates are performed is not arbitrary, in that some arrays need to be modified before others. For instance, the lengths of rows and columns have always to be udpated first, affecting the pointers to the starts, which then affect the row or column indices of entries. Our algorithm maintains this order of dependence among arrays and this is implicitly assumed to hold in all forthcoming descriptions of the sub-tasks.

## 5.1 Accessing Entries Along a Column

Often enough during any stage of Gaussian elimination one has to be able to locate entries below the pivotal element. To check whether there exists an entry in some position $(a, b)$, we can choose to either access the row $a$ looking for an entry whose column is $b$, or access the column $b$ looking for an entry whose row is $a$. If we know that this entry is likely to be situated in the start of a column (for instance, if $b$ happens to be a pivotal column), then accessing the entries by columns would be more efficient. Locating an entry in some position $(a, b)$ can be performed by scanning al entries $s \in \{col\_start[b], ..., col\_start[b] + col\_length[b] - 1\}$. In the remaining of this paper we denote by $Location\_by\_column(a, b)$ the sub-routine which when input the length and starting index of column $b$, returns $PASS$ if there exists an entry in location $(a, b)$ and $FAIL$ otherwise. It can be seen that the sub-routine requires at most $\gamma_c$ field operations, since in the worst case analysis, one would have to scan an entire column before finding an entry in location $(a, b)$.

## 5.2 Implementing the Markowitz Strategy

In our present implementation we use the Markowitz criterion [12] for locally minimizing the *fill-in* during each step of the Gaussian elimination, as opposed to other global methods which preserve the general sparsity pattern of the matrix [4]. The Markowitz strategy consists in locating good candidates for pivotal elements during each step of Gaussian elimination. By a good pivotal candidate $a_{ij}$ we mean a non-zero entry in the active part of the matrix which minimizes the *Markowitz count* $(r_i - 1)(c_j - 1)$, where $r_i$ and $c_j$ represent the lengths of row $i$ and column $j$ respectively, and where the minimum is over all entries of the active sub-matrix. Note that the Markowitz count represents the maximum amount of fill-in that could arise using a pivotal entry $a_{ij}$. The Markowitz criterion requires a further numerical stability test to be satisfied by the pivotal candidate, but this is not of concern in our implementation over $F_2$, since all arithmetic is exact. A straightforward implementation of the Markowitz search is likely to require scanning all entries in the active sub-matrix before a candidate is found. Curtis and Reid (see [3]) introduced methods to avoid an expensively naive search. We describe the method only very briefly here and refer the reader to [4] for more details. In principle, the approach consists of storing the various row and column lengths and searching for a pivotal candidate through the rows and columns in increasing order of counts (i.e. number of entries). This can be achieved by collecting all the rows (columns) in a set of doubly linked lists of rows (columns) having the same count. The row (column) lists can be constructed using two integer arrays, $row\_fwd\_link$ and $row\_backwd\_link$ (or $col\_fwd\_link$ and $col\_backwd\_link$), each of size $d$, and can be accessed through header pointers, stored in a $d$ dimensinal integer array, $row\_header$ ($col\_header$). The search for pivotal elements begins along rows and columns of least count, and progresses in increasing order, whereby the corresponding row and column lists are scanned. At any stage of the search, one can determine a bound on the Markowitz count of all unsearched entries, which helps terminate the search before all rows and columns are scanned. The following description can be found in [4] and we describe it only briefly here. Suppose that one is about to search rows with $r_i$ entries, so that that all columns whose count is less than $r_i$ have been scanned. This leaves only entries whose count is

$$(r_i - 1)(c_j - 1) \geq (r_i - 1)(r_i - 1) = (r_i - 1)^2. \tag{8}$$

Thus, the first entry whose count satisfies (8) is chosen as pivot. On the other hand, if one is searching columns with count $c_j$, then, since rows of count $c_j$ have already been scanned, the count of remaining unsearched entries along columns with count $c_j$ is

$$(r_i - 1)(c_j - 1) \geq (c_j + 1 - 1)(c_j - 1) = c_j(c_j - 1), \tag{9}$$

so that the search is terminated as soon as an entry satisfying (9) is encountered. Although there is no theoretical justification that this method always improves upon the $O(\tau)$ process through a naive search, experience has shown that it is likely to be successful after looking through only a few rows and columns (see [4] for figures).

### 5.2.1 Updating Lists of Columns Having the Same Length

As before, all following arguments work for columns as well as rows, by replacing the arrays with appropriate ones. A number of row and column operations may result in the column lengths being changed, and as a result, the lists of columns having the same count must undergo a corresponding modification. Suppose, for instance, that column $i$, of original length $a$, becomes of a new length $b$. This corresponds to removing $i$ from the list of columns of length $a$ and inserting it into the list of columns of length $b$. There are several positions into which one can choose to insert the column, the most natural being the head of the list. Obviously, this does not preserve the order of columns by increasing indices, something which, we argue, has advantages in the following:

i. Our application of the Markowitz strategy requires that we have an efficient way of determining the rows and columns of minimum count but only those in the active sub-matrix. Ordering the linked lists allows for a quick way of locating the active rows and columns of some particular count.

ii. Interchanging columns takes place when a pivotal candidate is chosen having some minimum Markowitz count $c$ and along a column $j$ that is different from the pivotal column, say $i$. As will become clearer later on, our algorithm for interchanging two columns $i$ and $j$ is most efficient when the total number of entries found along columns $k = i, ..., j$ is the least possible. All other columns containing entries whose count is $c$ and having the same length as $j$ belong to the same list, which when ordered in increasing order of column indices helps that we choose column $j$ such that $j$ is closest to $i$ among all columns of its list.

With these advantages, we are inclined to accept the extra cost of maintaining ordered linked lists. We denote the sub-routine for updating the lists as a result of changes in column lengths $Column\_chain(i, a, b)$[2], which updates the lists of lengths $a$ and $b$ as

[2] A similar algorithm, *Row_chain*, can be analogously constructed, by changing reference to the appropriate arrays.

the length of $i$ changes from $a$ to $b$. This can easily be seen to require at most $O(d)$ operations by considering the worst case analysis of inserting a column to the end of a list of $d$ columns.

### 5.3 Interchanging Columns

Because of the symmetry involved in exchanging rows and columns, we discuss only column interchanges. For $k = 1, ..., d$, let $c_k$ denote the vector occupying column $k$, $l(k)$ the length of $c_k$ in $M^{(q)}$ and $I_c(k)$ its length in $M' = I_c(M^{(q)})$. Similary, if $s$ denotes the index of an entry in the representation of $M$, then $I_c(s)$ denotes the index of that same entry in the representation of $M'$, whether viewed as a collection of sparse rows or columns. We will further require information about the greatest column index $c$ less than $k$ such that column $c$ is non-zero. This integer can be stored at location $k$ of the integer array *previous_column* of size $d$.[3] If no such integer exists, *previous_column*$[k]$ is set to $-1$. The array *previous_column* can be set at the start of the algorithm and later modified only when a zero column is displaced, a non-zero column has become of length zero, or a zero column has become non-zero. This modification will be assumed to hold implicitly in any of these cases and we leave it to the reader to verify that this comes at a negligible cost. We note that, if $c \neq -1$, then

$$column\_start[k] = column\_start[c] + column\_length[c].$$

The interchange of columns forces us to keep track of the corresponding change in the order of the coordinate entries of the unknown column vector $\mathbf{h^T}$ solving the linear system (2). This is necessary for the correctness of the final solution of the system after column interchanges have been performed. For every column $k = 1, ..., d$, we associate an integer $y$ representing the original index of $c_k$ in $M$ and store it in *original_column*$[k]$, where *original_column* is an integer array of size $d$. The array can also be initialized at the beginning of the program such that *original_column*$[k] = k$ for $k = 1, ..., d$ and modified accordingly when two columns get interchanged.

Now, let $i$ and $j$ denote the two columns to be interchanged such that $i < j$. The matrix $M' = I_c(M^{(q)})$ can be obtained through a series of changes affecting its column lengths and starts, as well as the row and column indices of its entries appearing. The algorithm for interchanging columns is presented as follows:

**Algorithm 1** *Interchange_Columns*$(i, j)$
*Input: the matrix* $M^{(q)}$*;*
*Output: the matrix* $M' = I_c(M^{(q)})$*, where* $I_c$ *represents the interchange of two columns* $i$ *and* $j$ *of* $M^{(q)}$*. Without loss of generality we may assume that* $i < j$ *and that at most one of* $i$ *or* $j$ *is zero.*
*1. Switch(original_column*$[i]$*,original_column*$[j]$*);*
*2.* $y \leftarrow$ *col_length*$[i]$*, col_length*$[i] \leftarrow$ *col_length*$[j]$*;*
*3. Column_chain*$(i, y, col\_length[i])$*;*

[3] A similar array, *previous_row*, can be used in the collection of rows.

*4.* $col\_length[j] \leftarrow y$;
*5.* $Column\_chain(j, col\_length[i], y)$;
**for** $k \in \{i, ..., j\}$ **do**
    **If** $(col\_length[k] = 0)$ **do**
*6.*         $col\_start[k] \leftarrow 0$;
    **else do**
*7.*         $c \leftarrow previous\_column[k]$;
        **If** $(c = -1)$ **do**
*8.*             $col\_start[k] \leftarrow 1$;
        **else do**
*9.*             $col\_start[k] \leftarrow col\_length[c] + col\_start[c]$;
        **end**;
    **end**;
**end**;
*10.* $Initialize\_to\_zero(new\_array)$, $sum \leftarrow 0$;
**for** $k \in \{i+1, ..., j-1\}$ **do**
*11.*     $sum \leftarrow sum + col\_length[k]$;
**end**;
**If** $(col\_length[i] \neq 0)$ **do**
*12.*     $a \leftarrow sum + col\_length[j]$;
    **for** $y \in \{col\_start[i], ..., col\_start[i] + col\_length[i] - 1\}$ **do**
*13.*         $new\_array[y] \leftarrow irn[y + a]$;
    **end**;
**end**;
*14.* $a \leftarrow col\_length[j] - col\_length[i]$;
**for** $k \in \{i+1, ..., j-1\}$ **do**
    **If** $col\_length[k] \neq 0$ **do**
        **for** $y \in \{col\_start[k], ..., col\_start[k] + col\_length[k] - 1$ **do**
*15.*             $new\_array[y] \leftarrow irn[y + a]$;
        **end**;
    **end**;
**end**;
**If** $(col\_length[j] \neq 0)$ **do**
*16.*     $a \leftarrow sum + col\_length[i]$;
    **for** $y \in \{col\_start[j], ..., col\_start[j] + col\_length[j] - 1\}$ **do**
*17.*         $new\_array[y] \leftarrow irn[y - a]$;
    **end**;
**end**;
*18.* $Copy(new\_array, irn)$, $Initialize\_to\_zero(new\_array)$;
**for** $k \in \{1, ..., d\}$ **do**
    **If** $(row\_length[k] \neq 0)$ **do**
*19.*         $sum \leftarrow 0$;
        **for** $s \in \{row\_start[k], ..., row\_start[k] + row\_length[k] - 1$ **do**

```
                 If (jcn[s] > i) and (jcn[s] < j) do
20.                   sum ← sum + 1;
                 end;
            end;
21.         t1 ← Location_by_column(k, i), t2 ← Location_by_column(k, j);
            If (t1 = FAIL) and (t2 = PASS)
            do
                 for s ∈ {row_start[k], ..., row_start[k] + row_length[k] − 1}
                 do
                      If (jcn[s] > i) and (jcn[s] < j) do
22.                      new_array[s − 1] ← jcn[s];
                      else if (jcn[s] = i) do
23.                      new_array[s + sum] ← j;
                      end;
                 end;
            end;
            If (t1 = PASS) and (t2 = FAIL) do
                 for y ∈ {row_start[k], ..., row_start[k] + row_length[k] − 1}
                 do
                      If (jcn[s] > i) and (jcn[s] < j) do
24.                      new_array[s + 1] ← jcn[s];
                      else if (jcn[s] = j) do
25.                      new_array[s − sum] ← i;
                      end;
                 end;
              end;
       end;
end;
27. Copy(new_array, jcn), Initialise_to_zero(new_array).
```

**Theorem 4** *Algorithm 1 performs correctly and requires* $O(d[\gamma_r+\gamma_c]+\tau)$ *field operations.*

**Proof:**

Steps 1-5 of the algorithm perform the initial changes that have to do with switching the lengths of columns as well as the original indices of columns occupying positions $i$ and $j$. As a result of changes in column lengths, the corresponding lists of columns of the same count have to be updated through calls to *Column_chain*, each requiring about $O(\gamma_c)$ operations.

In steps 6-9 we perform the changes to the starting pointers of columns. We claim that only columns $i, ..., j$ of $M^{(q)}$ can have the pointers to their starting entries changed in the representation of $M'$. Let $s$ denote the index of the starting entry of any column in the matrix. Since columns less than $i-1$ retain the same number and distribution of entries, it follows that all columns $k \in \{1, ..., i-1\}$ retain the same pointers to their starting entries. Suppose $k = i$ and let $c = previous\_column[i]$. If $i$ is the zero column in $M'$ (which, according to our original assumption, implies that $i$ is not zero in $M^{(q)}$), then $I_c(s) = 0 \neq s$. Else, if $i \neq 0$, and if $c = -1$, then $I_c(s) = 1 = s$; else, if $c \neq -1$, then

$$\begin{aligned} I_c(s) &= I_c(l(c)) + I_c(s(c)) \\ &= l(c) + s(c) \text{ since } 1 \leq c < i \\ &= s. \end{aligned}$$

Now suppose that $k = i+1, ..., j$ and column $k \neq 0$. If $I_c(l(i)) \neq l(i)$, the start of column $k$ changes as a result, if $k$ is not the zero column. In particular, if $c = previous\_column[k]$, then $I_c(s(k)) = l(c) + s(c)$ if $c \neq -1$, and $I_c(s(k)) = 1$ otherwise. If $k > j$, then since the total number of entries in the matrix block $M'(1 \to n; 1 \to j)$ is the same as that in the matrix block $M^{(q)}(1 \to n, 1 \to j)$, the start of column $k$ remains unchanged. The loop across steps 6-9 is iterated $j - i$ times, involving only array accesses, so that it requires at most $O(j - i)$ steps. In the worst cases analysis when $i = 1$ and $j = d$, the operational count is of the order $O(d)$.

It is immediate to see that the lengths of rows in $M'$ do not change as a result of column interchanges, and hence, the pointers to the starts of rows remain unchanged.

Steps 10-18 update the row indices of entries in the representation of $M'$ as a collection of columns. When $i$ and $j$ are interchanged, some entries in that collection will be displaced and as a result, the values in $irn$ will have to be updated accordingly to fit the new displacement. Because of the dependence of the new values of $irn$ on former values of the same array, the updates on $irn$ are copied first into the auxiliary array $new\_array$ which is then copied onto $irn$ when all necessary changes have been performed. Since only columns $k = i, ..., j$ in $M'$ had the pointers to their starting entries changed, it follows that only entries along these columns undergo a shift in their indices which then affects the values in $irn$. In particular, if $s$ denotes the index of an entry $e$ in $M^{(q)}$ as a collection of columns and $I_c(s)$ is its index in $M'$, then $irn[s] = irn[I_c(s)]$. To establish what the exact changes to $irn$ will be, it suffices to determine the exact value of $I_c(s)$ in each of the following cases:

a. Suppose $e \in j$ in $M^{(q)}$, then $e \in i$ in $M'$. Since $i < j$, and since the indexing of entries along the collection of columns is ordered increasingly, the index of $e$ in $M'$ is shifted to the left as $e$ gets displaced from $j$ to $i$. Furthermore, the amount of shift corresponds to the total number of entries being moved ahead of $e$ as $i$ and $j$ are interchanged. In particular, this amounts to the total number of entries found on columns $k = i, ..., j-1$ in $M$. Summarizing, $I_c(s)$ can be written as $I_c(s) = s - a$, where

$$a = \sum_{l(i) \leq l < l(j)} l = sum + l(i) = sum + I_c(l(j))$$

and $sum$ is as defined in the algorithm. Write $s' = I_c(s)$. Then $s' \in \{col\_start[i], ..., col\_start[i] + col\_length[i] - 1\}$. Since $new\_array[s'] = irn[s]$, we have $new\_array[s'] = irn[s' + a]$. Analogously, it is easy to see that, if $e \in i$ in $M^{(q)}$, then $e \in j$ in $M'$ so that

$I_c(s) = s + a$, where

$$a = \sum_{l(i)<l\leq l(j)} l.$$

The loops in steps 13 and 17 cover all entries in columns $i$ and $j$ and involve mainly array accesses so that their total cost amouns to $O(\gamma_c)$ field operations.

b. Suppose $e \in k$ such that $k = i+1, ..., j-1$. Interchanging $i$ and $j$ would result in shifting the index of $e$ to the left by $l(i) - l(j)$, since $i < j$, so that $I_c(s) = s - a$, where

$$a = l(i) - l(j) = I_c(l(j)) - I_c(l(i)).$$

The loop covers all entries of indices $s' = I_c(s)$ along $k$ and assigns $new\_array[s'] = irn[s'+a]$. The iterations of the loop cover all entries $e$ along columns $k = i+1, ..., j-1$ involving mainly array accesses so that the total cost of step 15 is $O\left((j-i)\gamma_c\right)$. When all the updates are performed, we copy $new\_array$ onto $irn$ and re-initialise $new\_array$ to zero. This requires about $O(\tau)$ operations. In the worst case analysis when $j = d$ and $i = 1$, the total cost for updating $irn$ thus becomes $O\left(\tau + d)\gamma_c\right)$.

Steps 19-27 aim at updating the columns of entries as their indices in the collection by rows change. As before, we store the new values of $jcn$ in $new\_array$. Upon interchanging columns $i$ and $j$, we have seen that only entries found along columns $k' = i, ..., j$ get displaced. Also, if $s$ denotes the index of an entry in the representation of $M^{(q)}$ as a collection of rows such that $I_c(s)$ is the index of that same entry in the representation of $M'$, then $jcn[s] = jcn[I_c(s)]$. For all rows $k = 1, ..., d$ we argue as follows:

a. If $k$ has entries in both columns $i$ and $j$, or does not have entries in both columns $i$ and $j$, then its representation in the collection by rows does not change as a result of interchanging the two columns. As a result, the column indices of all its entries remain unchanged.

Let $S$ denote the number of entries occupying position $(k, k')$ for $k' = i+1, ..., j-1$ and let $sum = \#S$.

b. If $k$ has an entry $e$ in column $i$ but not in column $j$ of $M^{(q)}$, and since $i < j$, then interchanging the two columns shifts the index $s$ of $e$ by $sum$ units to the right as a result of displacing elements of $S$ to the left. In other words, we have $I_c(s) = s + sum$, and in particular, $new\_array[I_c(s)] = j$. On the other hand, if $e'$ is an entry along row $k$ of $M^{(q)}$ such that $jcn[e'] = i+1, ..., j-1$, then switching columns $i$ and $j$ causes only $e$ to be shifted to the right ahead of $e'$. If $s'$ denotes the index of $e'$ in $M^{(q)}$, we have

$$I_c(s') = s' - 1 \text{ and } new\_array[s'-1] = jcn[s'].$$

c. In a very similar way, if there exists an entry $e$ in column $j$ but not in column $i$ of $M^{(q)}$, then $I_c(s) = s - sum$ where $s$ is the index of $e$ in $M^{(q)}$ and $sum$ is as above. In particular, we have $new\_array[I_c(s)] = i$. Also, for $e'$ occupying $(k, k')$ and $k' = i+1, ..., j-1$, $I_c(s') = s'+1$ so that $new\_array[s'+1] = jcn[s']$.

The updates on the column indices can thus be performed through a loop ranging over all rows of the matrix. Each loop involves two calls of the function *Location_by_column*, whose cost was seen to be $O(\gamma_c)$, as well as a series of updates on entries falling between columns $i$ and $j$. In the worst case analysis when $j = d$ and $i = 1$, steps 19-27 require $d.[\gamma_r + \gamma_c]$ operations.

Summing up the sub-costs of this algorithm, and using $\gamma_c < d$, it can be seen to require $O\left(d[\gamma_r + \gamma_c] + \tau\right)$ field operations. □

### 5.4 Adding Rows

Let $i$ denote the pivotal row in a particular stage of Gaussian elimination. We choose to view the process of replacing row $j > i$ with $j + i$ as a series of a composition of two sub-tasks which involve inserting a new entry to, or removing an already existing one from, row $j$, resulting in *fill-in* and *fill-out* respectively. Since we are working with integers modulo 2, where an entry is either zero or 1, the *fill-out* becomes of considerable significance, for the zeros we obtain as a result of elimination are not *accidental zeros*. In other words, if there exists some entry $e$ in the location $(i, k)$ and another entry $e'$ in $(j, k)$, then $e'$ is definitely (and not accidentally) transformed to zero as a result of the operation $j \leftarrow j + i$. Furthermore, our algorithm for replacing row $j$ with one that is probably longer than itself escapes previous restrictions in that it does not require adding a fresh copy of the modified row at the end of the data structure. In particular, our approach does not require the use of any elbow room beyond what is needed to accomodate for only the extra number of *fill-in* - *fill-out*. In some cases when this quantity is negative, the empty space is simply allocated at the end of the structure. Most of the updates capable of achieving this involve a shifting procedure as described in the previous section, and since this comes at a higher operational cost than using compression and elbow room, the trade off we establish is between savings in memory versus increase in running time.

We first present the following two sub-algorithms:

**Algorithm 2** *Remove_entry*$(j, k)$
*Input:* $M^{(q)}$, *where location* $(j, k)$ *is occupied by a non-zero entry.*
*Output:* $M' = R_e(M^{(q)})$, *where location* $(j, k)$ *is empty.*
1. $\tau \leftarrow \tau - 1$, $x \leftarrow row\_length[j]$, $row\_length[j] \leftarrow row\_length[j] - 1$;
2. $Row\_chain(j, x, row\_length[j])$;
3. $x \leftarrow col\_length[k]$, $col\_length[k] \leftarrow col\_length[k] - 1$;
4. $Column\_chain(k, x, col\_length[k])$;
**If** $(row\_length[j] = 0)$ **do**
5. $\quad row\_start[j] \leftarrow 0$;
**end**;
**If** $(col\_length[k] = 0)$ **do**
6. $\quad col\_start[k] \leftarrow 0$;
**end**;
**for** $s \in \{j + 1, ..., d\}$ **do**
$\quad$ **If** $(row\_start[s] \neq 0)$ **do**
7. $\quad\quad row\_start[s] \leftarrow row\_start[s] - 1$;
$\quad$ **end**;

**end**;
**for** $s \in \{k+1, ..., d\}$ **do**
**If** $(col\_start[s] \neq 0)$ **do**
*8.* $col\_start[s] \leftarrow col\_start[s] - 1$;
**end**;
**end**;
*9.* $new\_array[\tau + 1] \leftarrow 0$;
**If** $row\_start[j] \neq 0$ **do**
**for** $s \in \{row\_start[j], ..., row\_start[j] + row_length[j] - 1\}$ **do**
**If** $(jcn[s] \geq k)$ **do**
*10.* $new\_array[s] \leftarrow jcn[s+1]$;
**end**;
**end**;
**end**;
**for** $t \in \{j+1, ..., d\}$ **do**
**If** $(row\_start[t] \neq 0)$ **do**
**for** $s \in \{row\_start[t], ..., row\_start[t] + row\_length[t] - 1\}$ **do**
*11.* $new\_array[s] \leftarrow jcn[s+1]$;
**end**;
**end**;
**end**;
*12.* $Copy(new\_array, jcn)$, $Initialize\_to\_zero(new\_array)$;
*13.* $new\_array[\tau + 1] \leftarrow 0$;
**If** $(col\_start[k] \neq 0)$ **do**
**for** $s \in \{col\_start[k], ..., col\_start[k] + col\_length[k] - 1\}$ **do**
**If** $(irn[s] \geq j)$ **do**
*14.* $new\_array[s] \leftarrow irn[s+1]$;
**end**;
**end**;
**end**;
**while** $t \in \{k+1, ..., d\}$ **do**
**If** $(col\_start[t] \neq 0)$ **do**
**for** $(s \in \{col\_start[t], ..., col\_start[t] + col\_length[t] - 1\}$ **do**
*15.* $new\_array[s] \leftarrow irn[s+1]$;
**end**;
**end**;
**end**;
*16.* $Copy(new\_array, irn)$, $Initialize\_to\_zero(new\_array)$.

**Theorem 5** *Algorithm 2 performs correctly and requires $O(d + \tau)$ field operations.*

**Proof:**
As one entry is deleted, the total number of entries in the matrix and the lengths of row $j$ and column $k$ are all decreased by 1. Any change in the lengths of rows or columns has

to be followed by the corresponding changes in the lists of rows and columns of the same count. In total, steps 1-4 of the algorithm require $O(d)$ field operations through the two calls to *Row_chain* and *Column_chain*.

Steps 5-8 perform the updates on row and column starts. If row $j$ becomes zero, we set its row start to be zero. Else, let $s$ denote the index of the starting entry of $j$ in $M^{(q)}$. If $j = 1$, then $R_e(s) = s = 1$. Now suppose that $j > 1$ and let $M' = R_e(M^{(q)})$. Since the total number and distribution of entries in the matrix block $M'(1 \to j-1; 1 \to d)$ is the same as that in $M^{(q)}(1 \to j-1, 1 \to d)$, it follows that all rows less than or equal to $j$ retain the same pointer to their starting entries. If we further have $j < d$, we claim that, for rows $r = j+1, ..., d$, $R_e(s) = s - 1$ (where $s$ is the index of the starting entr of $r$) since removing one entry from row $j$ shifts the indexing of all entries in the remaining rows one unit to the left. As such, the updates on row starts require about $O(d-j)$ operations, which in the worst case analysis ($j = 1$) require $O(d)$ operations. A similar argument holds for the updates on column starts and hence can be skipped.

Steps 9-12 perform the updates on the column indices of entries in the collection by rows. As one entry $e$ is removed from row $j$, the location at the end of the array $jcn$ is freed and the indices of all entries following $e$ in the collection by rows shift one unit to the left. As before, the new updates are stored in *new_array* which is then copied onto $jcn$ before being re-initialized to zero. A very similar argument holds for the row indices of entries in the collection by columns when an entry is removed from column $k$, and we leave the details to the reader. The updates on $jcn$ and $irn$ thus require at most $O(\tau)$ steps, considering the worst case when $j = k = 1$. This brings the total cost of the algorithm to $O(d + \tau)$ field operations. □

**Algorithm 3** *Add_entry*$(j, k)$
*Input:* $M^{(q)}$, *where location* $(j, k)$ *is empty.*
*Output:* $M' = R_e(M^{(q)})$, *where location* $(j, k)$ *is occupied by a non-zero entry.*
*1.* $\tau \leftarrow \tau + 1$, $x \leftarrow row\_length[j]$, $row\_length[j] \leftarrow row\_length[j] + 1$;
*2.* $Row\_chain(j, x, row\_length[j])$;
*3.* $x \leftarrow col\_length[k]$, $col\_length[k] \leftarrow col\_length[k] + 1$;
*4.* $Column\_chain(k, x, col\_length[k])$;
**If** $(row\_start[j] = 0)$ **do**
*5.* $c \leftarrow previous\_row[j]$;
    **If** $(c \neq -1)$ **do**
*6.* $row\_start[j] \leftarrow row\_start[c] + row\_length[c]$;
    **else do**
*7.* $row\_start[j] \leftarrow 1$;
    **end**;
**end**;
**for** $t \in \{j+1, ..., d\}$ **do**
    **If** $(row\_start[t] \neq 0)$ **do**
*8.* $row\_start[t] \leftarrow row\_start[t] + 1$;
    **end**;
**end**;

```
for t ∈ {k + 1, ..., d} do
        If (col_start[t] ≠ 0) do
9.              col_start[t] ← col_start[t] + 1;
        end;
end;
10. t ← PASS;
If  (row_length[j] = 1) do
11.      new_array[row_start[j]] ← k, t ← FAIL;
else do
        for s ∈ {row_start[j], ..., row_start[j] + row_length[j]) − 2} do
                If (jcn[s] > k) and (t = PASS) do
12.                     new_array[s] ← k, t ← FAIL;
                end;
                If (jcn[s] > k) and (t = FAIL) do
13.                     new_array[s + 1] ← jcn[s];
                end;
        end;
end;
If (t = PASS) do
14.      new_array[s] ← k;
end;
for t ∈ {j + 1, ..., d} do
        If (row_start[t] ≠ 0) do
                for s ∈ {row_start[t], ..., row_start[t] + row_length[t] − 1} do
15.                     new_array[s] ← jcn[s − 1];
                end;
        end;
end;
16. Copy(new_array, jcn), Initialize_to_zero(new_array);
17. t ← PASS;
for s ∈ {col_start[k], ..., col_start[k] + col_length[k]) − 2} do
        If (irn[s] > k) and (t = PASS) do
18.             new_array[s] ← j, t ← FAIL;
        end;
        If (irn[s] > j) and (t = FAIL) do
19.             new_array[s + 1] ← irn[s];
        end;
end;
If (t = PASS) do
20.      new_array[s] ← j;
end;
for t ∈ {k + 1, ..., d} do
        If (col_start[t] ≠ 0) do
```

**for** $s \in col\_start[t], ..., col\_start[t] + col\_length[t] - 1\}$ **do**

*21.* $new\_array[s] \leftarrow irn[s-1]$;

**end**;

**end**;

**end**;

*22.* $Copy(new\_array, irn)$, $Initialize\_to\_zero(new\_array)$.

**Theorem 6** *Algorithm 3 performs correctly and requires $O(d+\tau)$ operations.*

**Proof:**
When an entry is added to the location $(j, k)$, the total number of entries in the matrix increases by 1, and so do the lengths of row $j$ and column $k$. The corresponding lists of rows and columns of the same length are modified through the calls to $Column\_chain$ and $Row\_chain$. Steps 1-4 as such require $O(d)$ field operations.

Steps 5-9 perform the updates on the row and column starts. As in algorithm 2, for rows $r = 1, ..., j$, $A_e(s) = s$, where $s$ is the index of the starting entry of a non-zero row $r$ (check proof of theorem 5 above), unless $j$ was originally the zero row, in which case we argue as follows. If $j$ becomes the first non-zero row of $M'$, then $A_e(s) = 1$; else,

$$A_e(s) = row\_start[c] + row\_length[c]$$

where $c = previous\_row[j]$. If we further have $j < d$, then for rows $r = j+1, ..., d$, $A_e(s) = s+1$ since adding one entry to row $j$ shifts the indexing of all entries in the remaining rows one unit to the right. A similar argument holds for the updates on the starts of columns in the matrix. For column $k$, however, we argue that $A_e(s) = s$, since the inserted entry will always appear after the start of column $k$ when this is the pivotal column. It can be further established that the operational complexity of the updates on the starts of rows and columns is of the order $O(d-j) + O(d-k)$ field operations, which in the worst case analysis ($j = k = 1$) becomes of the order $O(d)$.

Steps 10-16 perform the updates on the column indices of entries in the collection of rows. Let $e'$ denote the entry to be inserted. If row $j$ becomes of length 1, then $e'$ is given the column index $k$, and the algorithm skips to step 15 where updates on the column indices for entries in rows $r > j$ take place. Else, suppose that row $j$ becomes of length greater than 1. Let $e$ denote an entry in $j$ and of index $s$ in $M^{(q)}$. If $jcn[s] < k$, then $e$ also represents an entry of $j$ in $M'$ whose index is not affected, since it appears before $e'$. Else, if $e$ is the first entry of $j$ in $M^{(q)}$ such that $jcn[s] > k$, then $e'$ takes up the index of $e$ so that $new\_array[s] = k$ in $M'$. For all entries $e$ whose columns are greater than $k$ in $M^{(q)}$, $A_e(s) = s+1$ so that $new\_array[s+1] = jcn[s]$. If no entry $e$ of row $j$ in $M^{(q)}$ was found to have a column index greater than $k$ (step 14), then $e'$ is the last entry to appear in row $j$ and hence has index $s = row\_start[j] + row\_length[j] - 1$ which is achieved through exit of the loop in preceding step 12. For all entries $e$ along rows $r$ greater than $j$ and of index $s$ in $M^{(q)}$, $A_e(s) = s+1$ as a result of inserting $e'$ in $j$. When all updates are performed, $new\_array$ is copied onto $jcn$ and re-initialized to zero. In total, and considering the worse case analysis when $j = 1$, this step can be seen to require at most $O(\tau)$ field operations. A very similar argument can be established for updating the row values of entries in $M'$ as a collection of columns and we leave the details to the reader. Summing up the sub-costs of the algorithm, the total cost is $O(d+\tau)$ field operations. $\Box$

The algorithm for adding rows can now be described using the previous two sub-tasks. Our discussion above also demonstrates that no compression or creation of elbow room is required to accomodate for new copies of modified rows (or columns). Whatever extra space created corresponds only to the amount of fill-in minus fill-out, when this amount is positive. In particular, the algorithm can be stated as follows:

**Algorithm 4** *Add_Rows*$(i, j)$
*Input: Rows* $i$ *and* $j$ *such that* $i$ *is the pivotal row during one stage of Gaussian elimination and* $j$ *is some other row in the active sub-matrix of* $M^{(q)}$.
*Output: Row* $j$ *such that* $j \leftarrow j + i$.
**for** $s \in \{row\_start[i], ..., row\_start[i] + row\_length[i] - 1\}$ **do**
*1.* $\quad k \leftarrow jcn[s]$;
$\quad$ **If** $location\_by\_column(j,k) = PASS$ **do**
*2.* $\quad\quad Remove\_entry(j,k)$;
$\quad$ **else do**
*3.* $\quad\quad Add\_entry(j,k)$;
$\quad$ **end**;
**end**.

**Theorem 7** *Algorithm 4 performs correctly and requires* $O(\gamma_r(d+\tau))$ *field operations.*

**Proof:**
Replacing $j$ with $j+i$ modulo 2 can be peformed in distinct steps as follows. If both rows $i$ and $j$ contain an entry in the same column $k$, then performing $j+i$ will cancel out this entry in location $(j,k)$; else, this will introduce a new entry in $(j,k)$. The loop through steps 1-3 iterates $\gamma_r$ times during which a call to *Remove_entry* or *Add_entry* is performed. By theorems 5 and 6, the total running time of algorithm 4 is at most

$$O\left(\gamma_r(d+\tau)\right)$$

field operations. $\Box$

## 5.5 A Complete Sparse Gaussian Elimination Algorithm

The results of the previous subsections can now be used to construct a complete sparse Gaussian Elimination Algorithm. The foward sweep of the algorithm which reduces the matrix into Echelon form consists of at most $d$ pivotal steps. In each step, a Markowitz search is performed to find a suitable pivotal element. Accordingly, an interchnage of rows and/or columns is performed, and/or an elimination of entries falling below the pivotal element is performed by adding suitable rows. If we consider the worst case analysis in which the Markowitz search requires $O(\tau)$ operations, and combining the costs in theorems 4 and 7, we find that each pivotal step requires at most $O\left(d(\gamma_r+\gamma_c)+\tau\gamma_r\right)$ field operations, so that the forward sweep is of the order

$$O(d^2(\gamma_r+\gamma_c)+\tau d\gamma_r) \tag{10}$$

field operations. Our experience has shown that roughly speaking, the average number of entries in a row or column is negligible compared to $d$ (see table 1 below), so that the cost in (10) is of the order $O(k^2d^2)$, where $k$ is the maximum ratio $\tau/d$ attained during any stage of the reduction. It can also be seen that the backward sweep of the algorithm, which produces the basis elements once the matrix is reduced, requires about $O(d\gamma_r)$ field operations, so that the total cost of the algorithm involving both its forward and backward sweeps is bounded by $O(k^2d^2)$. When compared with the dense and explicit Gaussian elimination of order $O(d^3)$, our algorithm performs faster provided that $k \lessapprox \sqrt{d}$. In terms of its memory requirements, we have shown that the algorithm requires twelve integer arrays of size $d$, and three integer arrays of size $\tau = kd$. Compared to the spatial requirement $d^2$ of the explicit methods, our algorithm is more memory efficient provided $k \lessapprox (d-12)/3$. In total, the algorithm performs better on both levels, provided $k < (\sqrt{4d-39}-3)/2$ if $d$ is large enough ($d \gtrapprox 10$), an upper bound that easily holds in our implementations below.

# 6 Implementation and Run Times

| $d$ | $m$ | $k$ | $\max(\gamma_r, \gamma_c)$ | time |
|---|---|---|---|---|
| 8000 | 6 | 6 | 253 | 39.4 |
| 8000 | 7 | 2 | 33 | 1.2 |
| 8000 | 30 | 4 | 150 | 7.5 |
| 16000 | 7 | 2 | 34 | 4.7 |
| 16000 | 10 | 9 | 430 | 258.5 |
| 16000 | 14 | 2 | 126 | 6.1 |
| 32000 | 31 | 3 | 240 | 54.9 |
| 32000 | 10 | 15 | 854 | 19400 |
| 64000 | 7 | 2 | 40 | 77.8 |
| 64000 | 10 | 3 | 248 | 215.7 |
| 128000 | 10 | 3 | 363 | 812.9 |
| 128000 | 14 | 2 | 178 | 343.5 |
| 256000 | 14 | 2 | 288 | 1896.1 |
| 300000 | 11 | 2 | 52 | 1634.3 |
| 400000 | 11 | 2 | 49 | 5061.1 |

Table 1: Run times for setting up the Niederreiter matrix and solving the associated system.

All programs were written in C. The work was carried out at Oxford University Supercomputing center (OSC) on the Oswell machine. Only one processor of the Sun cluster was used to perform the serial work. The UltraSPARC III processors run at about 122.2 Mflops each. Table 1 represents the run times in minutes for setting up and solving the Niederreiter linear system for a trinomial of degree $d$ with $m$ irreducible factors over $F_2$.

We note the negligible values of $\max(\gamma_r, \gamma_c)$ compared to $d$, which seem to grow with $k$. The run times verify the effect of increasing $k$ or $d$ on the total execution time, and our findings assert that the matrix remains considerably sparse throughout the reduction phase no matter how large the degree of the trinomial grows. As a result, our sparse algorithm performs efficiently well without having to be transformed into a dense algorithm towards the final stages of Gaussian elimination. Our impression is that a similar behaviour might still be observed for sparse polynomials of more than three terms, particulary because of the distribution of the Niederreiter matrix, which heuristically seems to preserve its sparsity throughout the elimination phase.

## 7 Conclusion

In this paper we have investigated the initial and most limiting phase of the Niederreiter algorithm for trinomials over the binary field and determined the exact initial sparsity level of the associated Niederreiter linear system. A new sparse Gaussian elimination algorithm using the Markowitz strategy was developed for producing a basis of the solution set of the sparse linear system. The new algorithm exploits the Gustavson data structure but circumvents the problems associated with it regarding creation of elbow room and compression. The resulting algorithm is also more memory efficient than the two dimensional doubly linked list which has been the most efficient structure among linked-lists based data structures. The gains in spatial requirements come at the expense of running time where our new algorithm requires $O(k^2d^2)$ field operations, in contrast to the $O(k^2d)$ cost of other sparse algorithms. Our algorithm was used in solving very large sparse Niederreiter linear systems for trinomials over $F_2$. Our results have shown that the linear system is not only initially sparse, but remains considerably sparse throughout the elimination phase. The algorithm can also serve as an irreducibility test for trinomials over $F_2$, where a trinomial is irreducible if and only if $m = d - rank(N_f - I_{d\times d}) = 1$ [13]. Although our algorithm can be easily modified to become a general linear solver in various other applications, we expect it to be particulary effective in solving the Niederreiter linear system for sparse polynomials over $F_2$. Our work in this paper can be combined with results of [1] where the irreducible factors of $f$ are extracted from a basis of the solution set using a parallal approach to the Göttfert algorithm over $F_2$. When compared with work in [17] for a dense explicit linear algebra approach to the Niederreiter algorithm, the resulting hybrid algorithm is of a better spatial complexity for the factorization of large trinomials over $F_2$, provided $k \underset{\approx}{<} (d-12)/3$, a criterion that is easy to establish in the sparse Niederreiter linear system. Our algorithm achieves factorization degrees that are inaccessible to the dense implementation up to 16 processors (see [1]), and performs the nullspace stage for $d = 300000$ in about 27 hours using only one processor in contrast to the same performance in [17] requiring about ten hours and 256 nodes. Our algorithm also reaches factorizations beyond this degree. At the time of writing, tests for input sizes of $d = 512000, 1024000$, and 1500000 are being undertaken. If proven successful for such degrees, our algorithm can compete with the implementations of [6] for achieving very high record factorizations for trinomials over $F_2$.

**Acknowledgements:** The author is grateful to the Oxford Supercomputing Centre for allowing the use of its facilities to generate the reported experiments.

# References


[1] F. Abu Salem, "A BSP Parallel Model of the Göttfert Algorithm for Polynomial Factorization over $F_2$", Report PRG-RR-03-14, Oxford University Computing Laboratory, July 2003, *to appear in the proceedings of PPAM 2003.*

[2] E. R. Berlekamp, "Factoring Polynomials over Finite Fields", *The Bell System Technical Journal*, 1967, vol. 46, pp. 1853-1859.

[3] A. R. Curtis and J. K. Reid, "The solution of large sparse unsymmetric systems of linear equations", *J. Inst. Maths. Applics.*, vol. 8, 1971, pp. 344-353.

[4] I. S. Duff, A. M. Erisman, and J. K. Reid, "Direct Methods for Sparse Matrices", Oxford University Press, 1986.

[5] P. Fleischmann, "Connections Between the Algorithms of Berlekamp and Niederreiter for Factoring Polynomials over $F_q$", *Linear Algebra and its Applications*, vol. 192, 1993, pp. 101-108.

[6] P. Fleischmann, M. Holder, and P. Roelse, "The Black-Box Niederreiter Algorithm and its Implementation over the Binary Field", *Mathematics of Computation*, v. 72, 2003, pp. 1887-1899.

[7] S. Gao and J. von zur Gathen, "Berlekamp's and Niederreiter's Polynomial Factorization Algorithms", *Contemporary Mathematics*, vol. 168, 1994, pp. 101-116.

[8] J. von zur Gathen and J. Gerhard, *Modern Computer Algebra*, Cambridge University Press, Cambridge, 1999.

[9] R. Göttfert, "An Acceleration of the Niederreiter Factorization Algorithm in Characteristic 2", *Mathematics of Computation*, vol. 62, no. 206, 1994, pp. 831-839.

[10] F. G. Gustavson, "Some Basic Techniques for Solving Sparse Systems of Linear Equations", in *Sparse Matrices and their Applications*, D. J. Rose and R. A. Willoughby, eds., Plenum Press, New York, 1972, pp. 41-52.

[11] T.C.Y. Lee and S. A. Vanstone, "Subspaces and polynomial factorizations over finite fields", *AAECC*, vol. 6, 1995, pp. 147-157.

[12] H. M. Markowitz, "The Elimination Form of the Inverse and its Application to Linear Programming", *Management Sci.*, vol. 3, 1957, pp. 255-269.

[13] H. Niederreiter, "A New Efficient Factorization Algorithm for Polynomials over Small Finite Fields", *AAECC*, vol. 4, 1993, pp. 81-87.

[14] H. Niederreiter, "Factorization of Polynomials and Some Linear Algebra Problems over Finite Fields", *Linear Algebra and its Applications*, vol. 192, 1993, pp. 301-328.

[15] H. Niedrreiter, "Factoring Polynomials over Finite Fields Using Differential Equations and Normal Bases", *Mathematics of Computation*, vol. 62, 1994, pp. 819-830.

[16] H. Niederreiter and R. Göttfert, "Factorization of Polynomials over Finite Fields and Characteristic Sequences", *Journal of Symbolic Computation*, vol. 16, no. 5, 1993, pp. 401-412.

[17] P. Roelse, "Factoring high-degree polynomials over $F_2$ with Niederreiter's Algorithm on the IBM SP2", *Mathematics of Computation*, vol. 68, no. 226, 1999, pp. 869-880.

[18] A. F. van der Stappen, R. H. Bisseling, and J. G. G. van de Vorst, "Parallel Sparse LU Decomposition on a Mesh Network of Transputers", *SIAM J. Matrix Anal. Appl.*, vol. 14, no. 3, 1993, pp. 853-879.