## ABSTRACT

YUHASZ, GEORGE L. Berlekamp/Massey Algorithms for Linearly Generated Matrix Sequences. (Under the direction of Professor Erich Kaltofen.)

The Berlekamp/Massey algorithm computes the unique minimal generator of a linearly generated scalar sequence. The matrix generalization of the Berlekamp/Massey algorithm, the Matrix Berlekamp/Massey algorithm, computes a minimal matrix generator of a linearly generated matrix sequence. The Matrix Berlekamp/Massey algorithm has applications in multivariable control theory and exact sparse linear algebra. The fraction free version of the Matrix Berlekamp/Massey algorithm can be adapted into a linear solver for block Hankel matrices. A thorough investigation of the Matrix Berlekamp/Massey algorithm and the fraction free Matrix Berlekamp/Massey algorithm is presented. A description of the algorithms and their invariants are given. The underlying linear algebra of the algorithms is explored. The connection between the update procedures of the algorithms and the nullspaces of the related matrix systems is detailed.

A full definition and description of linearly generated matrix sequences and their various generators is given first as background. A new classification of all linearly generated matrix sequences is proven to exist. A complete description of the Matrix Berlekamp/ Massey algorithm and its invariants is then given. We describe a new early termination criterion for the algorithm and give a full proof of correctness for the algorithm. Our version and proof of the algorithm removes all rank and dimension constraints present in previous versions in the literature. Next a new variation of the Matrix Berlekamp/Massey algorithm is described. The fraction free Matrix Berlekamp/Massey algorithm performs its operations over integral domains. The divisions performed by the algorithm are exact. A full proof of the algorithm and its exact division is given. Finally, we describe two implementations of the Matrix Berlekamp/Massey algorithm, a Maple implementation and a C++ implementation, and compare the two implementations. The C++ implementation is done in the generic LinBox library for exact linear algebra and modeled after the Standard Template Library.

Berlekamp/Massey Algorithms for Linearly Generated Matrix Sequences

by
George Yuhasz

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fullfillment of the
requirements for the Degree of
Doctor of Philosophy

Mathematics

Raleigh, North Carolina

2009

APPROVED BY:

| | |
|---|---|
| Dr. Erich Kaltofen<br>Chair of Advisory Committee | Dr. Ilse Ipsen |
| Dr. Michael Singer | Dr. Agnes Szanto |

# DEDICATION

To mom, absent in person too soon, present in spirit always.

## BIOGRAPHY

George Yuhasz is the son of Roger and Jean Yuhasz. Born and raised in Meadowview, VA, he has one sister, Sarah and a niece Marilyn. After graduating high school, he entered Virginia Tech in Blacksburg, VA. George graduated Magna Cum Laude with is B.S. in Mathematics and a minor in Computer Science from Virginia Tech. He later completed a M.S. in Mathematics from Virginia Tech under the direction of Dr. Edward Green. George then began doctoral studies at North Carolina State University. His dissertation and research in the field of computer algebra has been conducted under the direction of Dr. Erich Kaltofen.

## ACKNOWLEDGMENTS

I want to acknowledge and thank important people who have helped get to this point in my life.

My Dad for always being encouraging and supportive.

My sister, Sarah, for always having excessive confidence in me.

My advisor, Dr. Kaltofen for his patience and instruction throughout my time at NCSU.

My friends, especially Benji and Tony

My graduate school colleagues and friends, especially Iti.

My professors at NCSU and my committee members.

My advisor at Virginia Tech, Dr Edward Green and all of my professors at Virginia Tech.

My fellow LinBox collaborators for all of their help.

The National Science Foundation for providing financial support for my research.

# TABLE OF CONTENTS

**LIST OF TABLES** ..... **vi**

**LIST OF FIGURES** ..... **vii**

**1 Introduction** ..... **1**

**2 Matrix Berlekamp Massey Algorithm** ..... **5**
2.1 Linearly Generated Matrix Sequences ..... 5
2.2 Matrix Berlekamp/Massey Algorithm ..... 10
2.2.1 **Algorithm** *Matrix Berlekamp/Massey* ..... 11
2.2.2 Auxiliary Gaussian Elimination Algorithm ..... 13
2.3 Properties and Correctness of the Matrix Berlekamp /Massey Algorithm ..... 15
2.3.1 Preliminary Invariants ..... 15
2.3.2 Nullspace theorem ..... 22
2.3.3 Correctness of the output ..... 26
2.4 Matrix Berlekamp/Massey and Rectangular Matrix Sequences ..... 31
2.5 Arithmetic complexity ..... 33

**3 Fraction Free MBM** ..... **36**
3.1 Introduction ..... 36
3.2 Fraction Free Matrix Berlekamp/Massey Algorithm ..... 37
3.2.1 Algorithm ..... 37
3.2.2 Normalizable Remainder Sequences ..... 39
3.3 Proof of correctness ..... 41
3.3.1 Minimality of the output ..... 41
3.3.2 Integrality ..... 44
3.4 Block Hankel Systems With Arbitrary Right Sides ..... 48
3.5 Examples and Conclusions ..... 48
3.5.1 Scalar Example ..... 48
3.5.2 Scalar Monic Minimal Generators ..... 50

**4 Implementing the Matrix Berlekamp/Massey Algorithm** ..... **54**
4.1 Maple Implementation ..... 54
4.2 LinBox Implementation ..... 55
4.2.1 BM_Seq description ..... 56
4.2.2 BM_iterator description ..... 61
4.2.3 BM_Seq example ..... 70
4.3 Comparison of implementations ..... 71

**5 Summary and Conclusions** ..... **75**

# LIST OF TABLES

Table 1.1 Algorithms for computing minimal linear generators ..... 2

Table 2.1 Arithmetic operation comparison ..... 35

Table 3.1 Example 1 ..... 49

Table 3.2 Example 2 ..... 50

Table 4.1 Implementation timings; $p = 2$ ..... 73

Table 4.2 Implementation timings: $p = 101$ ..... 73

Table 4.3 Implementation timings: $p = 2147483647 = 2^{31} - 1$ ..... 73

# LIST OF FIGURES

Figure 3.1 Berlekamp/Massey algorithm ..... 44

Figure 3.2 Block-Hankel Linear System Solver ..... 53

Figure 4.1 BM_Seq Declaration ..... 57

Figure 4.2 BM_Seq private members ..... 57

Figure 4.3 BM_Seq constructors ..... 58

Figure 4.4 BM_Seq member functions ..... 59

Figure 4.5 BM_Seq member functions continued ..... 60

Figure 4.6 BM_Seq BM_iterator member ..... 61

Figure 4.7 BM_iterator declaration ..... 62

Figure 4.8 BM_iterator private members ..... 64

Figure 4.9 BM_iterator state members ..... 64

Figure 4.10 BM_iterator TerminationState ..... 65

Figure 4.11 BM_iterator constructors ..... 67

Figure 4.12 BM_iterator increment operators ..... 68

Figure 4.13 BM_iterator increment operators ..... 69

Figure 4.14 BM_iterator dereference operators ..... 70

Figure 4.15 BM_iterator data access members ..... 71

Figure 4.16 Using BM_Seq and BM_iterator ..... 72

Figure 4.17 Methods of initialization and construction ..... 72

# Chapter 1

# Introduction

Generalizations of the Berlekamp/Massey algorithm originally formulated to compute linear generators of scalar sequences [5; 33] are powerful tools in the applications of matrix sequences. The Matrix Berlekamp/Massey algorithm will compute a minimal matrix generator of a matrix sequence if one exists. This application was first used to find partial realizations of matrix sequences in multivariable control theory [37; 14]. More recently, the Matrix Berlekamp/Massey has been incorporated into the area of sparse linear system solving over finite fields by its use in the Block Wiedemann algorithm [12]. This application is very natural since the scalar Berlekamp/Massey algorithm was used in the Wiedemann blackbox linear system solver that utilizes scalar sequences [47]. Finally, the fraction free version of the Matrix Berlekamp/Massey algorithm presented herein demonstrates how the algorithm can be utilized as a linear solver for block Hankel or block Toeplitz matrices.

Scalar and matrix sequences arise in many different applications and problems. As such, there have been many algorithms developed to compute the minimal generators of these sequences that vary in their method. A summary of the known methods for computing the minimal generator of scalar and matrix sequences is given in Table 1.1. The different algorithms are categorized by the approaches the algorithm takes. These different approaches include methods based on the original Berlekamp/Massey algorithm, methods based on the extended Euclidean algorithm, methods that compute a Padé approximant, methods that compute a $\sigma$-basis, and methods that solve the underlying Toeplitz/Hankel linear system. The methods that reduce the time complexity from quadratic in the dimension of the linear system to essentially linear (up to poly-logarithmic factors) by use of fast polynomial arithmetic by a "*."

Table 1.1: Algorithms for computing minimal linear generators

| *Style of algorithm* | *Scalar case* | *Matrix case* |
|---|---|---|
| Berlekamp/Massey | Berlekamp [5]<br>Massey [33]<br>Dornstetter [15]<br>Giesbrecht et al. [22], §2 | Rissanen [37]<br>Dickinson et al. [14]<br>Coppersmith [12]<br>Thomé [41]*<br>This paper |
| Extended Euclidean | Sugiyama et al. [40]<br>Dornstetter [15] | Kaltofen and Villard [30], §3.1* |
| Padé approximant | Brent et al. [6]* | Van Barel and Bultheel [43]<br>Van Barel and Bultheel [44]<br>Giorgi et al. [23]* |
| $\sigma$-basis | Kaltofen and Lee [28], §2.2 | Beckermann and Labahn [3]*<br>Villard [45]*, Turner [42] |
| Toeplitz/Hankel solver | Levinson [31]<br>Durbin [19] | Cabay et al. [10]<br>Kaltofen [27], Appendix A* |

*of essentially linear complexity in the degree

Clearly, all methods are related by virtue that they solve the same linear algebra problem. However, for the scalar case it was not until [15] that the Euclidean algorithm-based approach by [40] was reduced to the actual Berlekamp/Massey algorithm via a now classical unraveling of the doubly nested polynomial remainder chain-polynomial division loops to the single Berlekamp/Massey loop. A fraction-free version of the original Berlekamp/Massey algorithm is given in [22] and in [28] a re-interpretation as a specialized $\sigma$-basis algorithm is described. The first quadratic algorithm for computing linear generators in the generic case is Levinson's 21 years before Berlekamp and Massey. An essentially linear time randomized algorithm for the block Levinson-Durbin problem based on the theory of displacement rank and that is valid for all inputs and over any field is in [27]. The theory of $\sigma$-bases [3] in a different manner converts the matrix-Padé problem to a scalar linear algebra problem and, as has been observed by [45] and worked out in [42], can be employed for the computation of minimal linear generators of matrix sequences. [43] and [44] give algorithms to compute a matrix Padé approximant along a diagonal path [43] and along any path [44] in the Padé table. [23] give a polynomial matrix multiplication based algorithm to compute a matrix Padé approximant.

The research presented produced three results in the matrix generalization of the Berlekamp/Massey algorithm. First linearly generated matrix sequences and the Matrix Berlekamp/Massey algorithm are fully explored. A complete categorization of all linear

generated matrix sequences is given. A proof of correctness and complexity of the Matrix Berlekamp/Massey algorithm is also given. Next a new generalization of Berlekamp/Massey algorithm is described. A fraction free version of the Matrix Berlekamp/Massey algorithm is given. This algorithm is designed to work in integral domains with exact division. A full proof of the algorithm and its exact division is given. Finally, two implementations of the Matrix Berlekamp/Massey algorithm are described and compared. One is an implementation in the computer algebra system Maple and the second is a C++ implementation in the exact linear algebra library LinBox.

The Matrix Berlekamp/Massey algorithm as described in [14; 12] is described in detail here. We only consider the quadratic-time version; see [41] for a variant of essentially linear complexity. The objective is to give a stand-alone correctness proof based on explicit loop invariants for the algorithm which makes no restrictions: arbitrary rectangular sequences of matrices whose rank can gradually increase. The algorithms in [14; 12] are not completely general and the correctness proof in [14] seems incomplete. We are able to give an explicit early termination criterion that adapts to the relation of the bound on the determinantal degree of the generator, which has to be input, and the intermediate computations of the algorithm. Our termination criterion seems missing from the literature, even in the scalar case (see Remark 1 on page 30). We assume the determinantal degree bound is correct, and when possible the algorithm may diagnose an insufficient degree bound. However, since verifying the correctness of the degree bound requires any algorithm to process infinitely many sequence elements, no algorithm can certify that the degree bound is large enough. We can state that the output of the Matrix Berlekamp/Massey algorithm using the termination criterion is the minimal matrix generator for the completion of the matrix sequence prefix used to compute the generator. Using the termination criterion, we give a worst case complexity analysis that is applicable to any input. We also include a comparison of our algorithm and a method based on the fast power Hermite padé solver of [3]. Theorem 3 in Section 2.1 gives an absolute description of any linearly generated matrix sequence that seems to have never been shown before.

Fraction free algorithms perform their computations, including divisions, over integral domains such as the integers $\mathbb{Z}$ and a polynomial domain $F[x]$. The divisions performed by these algorithms are exact so that no fractions are computed. A fraction free version of the scalar Berlekamp/Massey algorithm was first given in [22]. A fraction free version of the Matrix Berlekamp/Massey algorithm is presented here. Unlike the field arithmetic Matrix

Berlekamp/Massey algorithm, the fraction free algorithm imposes certain restrictions on the matrix sequences. The algorithm requires that the matrices of the sequence are square and that the sequence be an *normalizable remainder sequence*. When the matrix algorithm is specialized to the scalar case, the resulting algorithm is a different scalar fraction free Berlekamp/Massey algorithm that has smaller intermediate values. A comparison of the two scalar algorithms is shown. Finally, an interesting property of linearly generated scalar sequences is proven.

The last contribution of the research is a description and comparison of two implementations of the Matrix Berlekamp/Massey algorithm. The first implementation is a straight forward implementation in the computer algebra system Maple. The second implementation is done in C++ and explores more interesting programming designs. The implementation is done in LinBox, C++ library for exact linear algebra. The LinBox library includes many implementations of finite field arithmetic that we want to access. Further, the implementation is patterned after the Standard Template Library's container and iterator system. The algorithm and the matrix sequence it takes as input are combined into one overall data structure. The timings of the two implementations on the same test data are shown to compare the compiled and specialized field arithmetic of the LinBox implementation with the uncompiled and basic field arithmetic of the Maple implementation.

# Chapter 2

# Matrix Berlekamp Massey Algorithm

## 2.1 Linearly Generated Matrix Sequences

Linearly generated matrix sequences are an extension and generalization of linearly generated scalar sequences. We will begin with a definition of a linearly generated matrix sequence and a scalar generator. We will then extend the notion of a generator to include vector and matrix generators. Using results from module theory and linear control theory we define minimal matrix generators. Finally, we will show the relationship between minimal matrix generators and scalar generators.

The following defines both a linearly generated matrix sequence and the scalar polynomial generator of such a sequence. Both definitions are simple generalizations of results from the theory of linearly generated scalar sequences.

**Definition 1** *A sequence of matrices* $\{M_k\}_{k=0}^{\infty}$ *with* $M_k \in K^{N_{row} \times N_{col}}$ *is linearly generated if there exists a polynomial* $F(z) = \sum_{i=0}^{n} c_i z^i \in K[z]$ *with* $F \neq 0$ *where the following holds:*

$$(\forall l, l \geq 0) \colon \sum_{i=0}^{n} c_i M_{i+l} = 0^{N_{row} \times N_{col}}.$$

*The polynomial* $F$ *is called a scalar generator of the sequence* $\{M_k\}_{k=0}^{\infty}$*. The unique minimal scalar generator of the sequence* $\{M_k\}_{k=0}^{\infty}$ *is the monic generator of minimal degree.*

The definition above can be generalized by changing the coefficients from scalars into vectors in $K^{N_{\mathrm{col}}}$. Making such a change leads to the following definition.

**Definition 2** *$F(z) = \sum_{k=0}^{n} c_k z^k \neq 0 \in K^{N_{col}}[z]$ is a right vector generator of a matrix sequence $\{M_k\}_{k=0}^{\infty}$ if the following holds:*

$$(\forall l, l \geq 0) \colon \sum_{k=0}^{n} M_{k+l} c_k = 0^{N_{row}}.$$

*The sequence is said to be linearly generated from the right by the vector polynomial.*

An analogous definition can be made for left vector generators. We will only consider right generators and so will drop the term right and all generators will be right generators unless specified.

The introduction of vector generators allows us to define a matrix generator and a minimal matrix generator. We first describe the $K[z]$ module structure of the set of vector generators of a linearly generated matrix sequence.

**Fact 1** *Assume that $\{M_k\}_{k=0}^{\infty}$ is linearly generated, and let $W = \{F \in K^{N_{col}}[z] \mid F$ is a vector generator of $\{M_k\}_{k=0}^{\infty}$.*

1. *$W$ is a $K[z]$ submodule of $K^{N_{col}}[z]$.*
2. *$W$ has rank $N_{col}$.*
3. *$W$ has a basis.*

**Proof.** Let $g \in K[z]$, $g \neq 0$ be the linear generator for $\{M_k\}_{k=0}^{\infty}$. That $W$ is a submodule is straight-forward. To prove that $W$ has rank $N_{\text{col}}$, consider the following vector polynomials. For all $1 \leq i \leq N_{\text{col}}$ let $g_i = g \cdot e_i$, where $e_i$ is the standard coordinate vector. Each $g_i$ is a vector generator of the sequence and the $N_{\text{col}}$ vectors are linearly independent. Thus the submodule $W$ has rank at least $N_{\text{col}}$, but since $W$ is a submodule of $K^{N_{\text{col}}}[z]$, it can have rank no larger than $N_{\text{col}}$ [18,p. 371]. Therefore $W$ has rank $N_{\text{col}}$.

The fact that $W$ has a basis is a classic result of module theory [18,p. 371]. Any submodule of a free module over a principal ideal domain is also a free module and so any submodule has a basis. The result proves that any submodule of a free module over a P.I.D. has a basis with certain properties. The basis is defined by vectors $w_1, w_2, \ldots, w_{N_{\text{col}}} \in K^{N_{\text{col}}}[z]$ and elements $g_1, g_2, \ldots, g_{N_{\text{col}}} \in K[z]$ where the following properties hold. The vectors $g_1 \cdot w_1, g_2 \cdot w_2, \ldots, g_{N_{\text{col}}} \cdot w_{N_{\text{col}}}$ form a basis of $W$; the vectors $w_1, w_2, \ldots, w_{N_{\text{col}}}$ form a basis of $K^{N_{\text{col}}}[z]$; and $g_{N_{\text{col}}} \mid g_{N_{\text{col}}-1} \mid \ldots \mid g_2 \mid g_1$. This basis is related to the Smith Normal form of the submodule $W$. ■

We give an example of a matrix sequence that is not linearly generated and how such sequences violate Fact 1. Let $M_k = \begin{bmatrix} 0 & \binom{k}{\lceil k/2 \rceil} \end{bmatrix}$, were $(M_k)_{1,2}$ is the center element of the $k^{th}$ row of Pascal's triangle. This scalar sequence is is not linearly generated [26] and so the matrix sequence is not linearly generated. The matrix sequence has a nonzero vector generator, given by $F = \begin{bmatrix} 1 & 0 \end{bmatrix}^T$. So the submodule $W$ defined in Fact 1 is non trivial with rank at least 1. As in the proof above, $W$ is a $K[z]$ submodule of $K^2[z]$ and $W$ has a basis. Since the center Pascal numbers are not linearly generated, then any vector generator of the sequence must have 0 in the second row, and so the rank of $W$ is equal to 1. Thus linearly generated matrix sequences ensure that the annihilator submodule $W$ is full rank. We will now use the facts that $W$ is full rank and has a basis to define right matrix generators and right minimal matrix generators.

**Definition 3** *$F = \sum_{k=0}^{n} C_k z^k \neq 0 \in K^{N_{col} \times N_{col}}[z]$, $\det(F) \neq 0$, is a right matrix generator of a matrix sequence $\{M_k\}_{k=0}^{\infty}$ if the following holds:*

$$(\forall l, l \geq 0) \colon \sum_{k=0}^{n} M_{k+l} C_k = 0^{N_{row} \times N_{col}}.$$

*The sequence is said to be linearly generated from the right by the matrix polynomial.*

**Definition 4** *The matrix polynomial $F \in K^{N_{col} \times N_{col}}[z]$ is said to be a minimal right matrix generator of $\{M_k\}_{k=0}^{\infty}$ if $F$ is a right matrix generator and the columns of $F$ form a basis of the $K[z]$ submodule $W$ defined in Fact 1.*

The definition above describes a minimal matrix generator. Similar to integral bases of algebraic number rings, the determinant of a minimal matrix generator has minimal degree.

**Theorem 1** *$F \in K^{N_{col} \times N_{col}}[z]$ is a minimal matrix generator of $\{M_k\}_{k=0}^{\infty}$ if and only if $deg(det(F))$ is minimal over all right matrix generators of $\{M_k\}_{k=0}^{\infty}$.*

Like the scalar generator case, there are many different minimal matrix generators. In the scalar case, the monic generator of minimal degree is defined to be the unique minimal generator. For the matrix generator case, we use the following definition.

**Definition 5** *The unique minimal right matrix generating polynomial of the sequence $\{M_k\}_{k=0}^{\infty}$is the polynomial $F \in K^{N_{col} \times N_{col}}[z]$ such that $F$ is a minimal matrix generator and $F$ is in Popov form [36].*

The next theorem shows the connection between our definition of linearly generated matrix sequences and matrix generators.

**Theorem 2** *Let $F \in K^{N_{col} \times N_{col}}[z]$ be a minimal matrix generator of $\{M_k\}_{k=0}^{\infty}$, then $\{M_k\}_{k=0}^{\infty}$ is linearly generated. Further, the first invariant factor of $F$ is the unique minimal scalar generator of $\{M_k\}_{k=0}^{\infty}$.*

**Proof.** Let $f_1$ be the first or largest invariant factor of $F$, such that the Smith Normal form of $F$ is given by the matrix $\text{diag}(f_1, \ldots, f_N)$, with $f_i$ dividing $f_{i-1}$ for all $N \geq i > 1$. There exists a matrix polynomial $\bar{F} \in K^{N \times N}[z]$ such that $F^{-1} = (1/f_1) \cdot \bar{F}$. Further, we know that for any $H \in K^{N \times N}[z]$, $F \cdot H$ is a right matrix generator of the sequence $\{M_k\}_{k=0}^{\infty}$. Therefore, the sequence has a matrix generator $F \cdot \bar{F} = f_1 \cdot F \cdot F^{-1} = f_1 \cdot I_N$. This means that $f_1$ must be a scalar generator of $\{M_k\}_{k=0}^{\infty}$.

To see that $f_1$ is a unique minimal generator, use the additional fact that $f_1$ is the least common denominator of the entries of $F^{-1}$.

Now let $g$ be the unique minimal scalar generator of $\{M_k\}_{k=0}^{\infty}$. Since $f_1$ is a scalar generator of the sequence, then $g \mid f_1$. Further the matrix polynomial $g \cdot I_N$ is a matrix generator of $\{M_k\}_{k=0}^{\infty}$. Because $F$ is a minimal matrix generator, there exists a $G$ such that $F \cdot G = I_N \cdot g$. So then $F^{-1} = 1/g \cdot G$ and so from the previous paragraph, $f_1 \mid g$. But $f_1$ is monic and so $f_1 = g$. ■

An important motivation for studying linearly generated matrix sequences is the fact that the sequence $\{M_k\}_{k=0}^{\infty}$ is linearly generated if the sequence is defined by $M_k = U^T \cdot B^k \cdot V$, where $U$, $B$ and $V$ are matrices over $K$. Such matrix sequences are block bilinear projections of matrix powers. These projections arise in the analysis of the block Wiedemann and block Lanczos algorithms. The results in [27; 46; 30] detail properties of the matrix sequences generated by random bilinear projections and their minimal matrix generators. We will now demonstrate that all linearly generated matrix sequences can be viewed as a bilinear projection of a matrix power sequence.

**Theorem 3** *The sequence $\{M_k\}_{k=0}^{\infty}$ is linearly generated if and only if there exist matrices $U$, $B$ and $V$ with entries in $K$ such that $M_k = U^T \cdot B^k \cdot V$.*

**Proof.** Proving one direction is simple. If $M_k = U^T \cdot B^k \cdot V$ and $f$ is the characteristic polynomial of $B$, then $f$ is a scalar generator of $\{M_k\}_{k=0}^{\infty}$. Thus $\{M_k\}_{k=0}^{\infty}$ is linearly generated.

To prove the other direction we begin by proving the $1 \times 1$ or scalar case. Suppose $\{M_k\}_{k=0}^{\infty}$ is a linearly generated $1 \times 1$ matrix (scalar) sequence and let $f$ be a monic scalar generator of degree $m$. Let $B_f$ be the companion matrix of $f$. So $B_f$ is an $m \times m$ matrix of the following form:

$$B_f = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & 1 \\ -f_0 & -f_1 & \dots & -f_{m-2} & -f_{m-1} \end{bmatrix}.$$

So we let $B = B_f$. Further we define $U, V \in K^{m \times 1}$ in the following manner:

$$U = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \end{bmatrix} = e_1 \quad \text{and} \quad V = \begin{bmatrix} M_0 \\ M_1 \\ \vdots \\ M_{m-1} \end{bmatrix}.$$

By solving the equation in Definition 1 for $z^m$, then for any monic generator of degree $m$ the following is true:

$$(\forall l, l \geq 0) \colon M_{m+l} = \sum_{i=0}^{m-1} -f_i \cdot M_{i+l}.$$

Therefore we know the following is true about the action of $B^l$ on the vector $V$:

$$(\forall l, l \geq 0) \colon B^l \cdot V = \begin{bmatrix} M_l \\ M_{l+1} \\ \vdots \\ M_{m+l-1} \end{bmatrix}.$$

Since the action of $U^T$ is to select the top row of the vector $B^l \cdot V$, then $M_k = U^T \cdot B^k \cdot V$.

Let $\{M_k\}_{k=0}^{\infty}$ be a linearly generated matrix sequence of $N_{\text{row}} \times N_{\text{col}}$ matrices and let $f$ be the unique minimal scalar generator of the sequence of degree $m$. Let $B_f$ be the companion matrix of $f$ as defined previously. We will generalize the scalar construction above to define the matrices $U$, $B$ and $V$. Since $\{M_k\}_{k=0}^{\infty}$ is linearly generated by $f$, we can fix an $i$ and $j$ and see that the scalar sequence defined by $\{(M_k)_{i,j}\}_{k=0}^{\infty}$ is a linearly

generated scalar sequence and $f$ is a generator. As above, let us define the vector $V_{i,j}$ to be the following:

$$V_{i,j} = \begin{bmatrix} (M_0)_{i,j} \\ (M_1)_{i,j} \\ \vdots \\ (M_{m-2})_{i,j} \\ (M_{m-1})_{i,j} \end{bmatrix}.$$

We can now define the matrices $U$, $B$ and $V$ as block matrices. $B$ will be a $(N_{\text{row}} \cdot m) \times (N_{\text{row}} \cdot m)$ block diagonal matrix of the form:

$$B = \text{diag}(B_f, \ldots, B_f).$$

$U$ is a sparse $(N_{\text{row}} \cdot m) \times N_{\text{row}}$ matrix given by:

$$U = \text{diag}(e_1, \ldots, e_1).$$

$V$ is an $(N_{\text{row}} \cdot m) \times N_{\text{col}}$ matrix defined in the following block form:

$$V = \begin{bmatrix} V_{1,1} & V_{1,2} & \ldots & V_{1,N_{\text{col}}} \\ V_{2,1} & V_{2,2} & \ldots & V_{2,N_{\text{col}}} \\ \vdots & \vdots & \vdots & \vdots \\ V_{N_{\text{row}},1} & V_{N_{\text{row}},2} & \ldots & V_{N_{\text{row}},N_{\text{col}}} \end{bmatrix}.$$

Due to the block nature of the matrices $U$, $B$ and $V$, and the fact that the scalar sequences embedded in the $i, j$ entries of the matrices in $\{M_k\}_{k=0}^{\infty}$ are linearly generated by $f$, then $M_k = U^T \cdot B^k \cdot V$. ■

## 2.2 Matrix Berlekamp/Massey Algorithm

In this section we present a version of the Matrix Berlekamp/Massey algorithm that will compute a minimal (right) matrix generating polynomial of a matrix sequence. Other versions of this algorithm can be found in [32], and [12]. The algorithm given below removes a condition found in the versions above. Both [12] and [32] require that the initial matrix in the sequence be full rank, but by changing the initialization and updating procedure, we have removed this requirement.

In the previous section, the matrix sequences were generically rectangular. There were no assumptions made on the relationship between the row dimension $N_{\text{row}}$ and the column dimension $N_{\text{col}}$. All of the properties of minimal matrix generators are valid over all linearly generated matrix sequences of all possible dimensions. The row and column dimension of the matrix sequence will define the dimension and number of variables needed during the Matrix Berlekamp/Massey algorithm. Further, the invariants and proof of the Matrix Berlekamp/Massey algorithm are dependent on the dimensions of the matrix sequence. For the purpose of simplification, we present a description and proof of the Matrix Berlekamp/Massey algorithm for square matrix sequences, ie $N_{\text{row}} = N_{\text{col}}$. In a later section, a description and proof of the rectangular algorithm will be given. Because we are assuming $N_{\text{row}} = N_{\text{col}}$, we will drop the subscripts and just let $N$ be the row and column dimension of the matrix sequence.

In the algorithm description we will refer to the following matrix rank, which is computed implicitly (see Lemma 5 on page 17 below).

**Definition 6** *By $r_t$ we denote the rank of $\begin{bmatrix} M_0 & M_1 & \cdots & M_t \end{bmatrix}$, and we set $r_{-1} = 0$.*

If $M_0$ is singular, $r_t$ can increase beyond $t = 0$. The algorithm then performs an "on-the-fly" initialization (see step GE24 on page 15 below).

### 2.2.1 Algorithm *Matrix Berlekamp/Massey*

**Input** $M(z) \in K^{N\times N}[[z]]$ with $M_i \in K^{N\times N}$ the coefficient of $z^i$

$\delta \geq 0$ an upper bound on the determinantal degree of the matrix generator.

**Output** $F(z) \in K^{N\times N}[z]$ a minimal matrix generator.

In the case that the input $\delta$ is not an upper bound for the determinantal degree, the algorithm may return "insufficient bound" or an incorrect $F$.

**Variables** $f(z) \in K^{N\times 2N}[z]$ where the first $N$ columns $f_1, \ldots, f_N$ represent the reversal of the current generator and the last $N$ columns $f_{N+1}, \ldots, f_{2N}$ are what we call the *auxiliary polynomials*.

$d_1, d_2, \ldots, d_{2N}$ the nominal degree of each column. We have throughout the algorithm that $\deg(f_j) \leq d_j$. Since $f_j$ are reversed polynomials, $d_j$ are upper bounds for the degrees of their reversals $z^{d_j} f_j(z^{-1})$.

$\tau \in K^{2N\times 2N}$ a linear transformation

$\Delta \in K^{N\times 2N}$ the discrepancy matrix

$\beta$ the minimum of the nominal degrees of the auxiliary polynomials

$\sigma$ the sum of the nominal degrees of the generator polynomials

$\mu$ the maximum of the nominal degrees of the generator polynomials

MBM1 $f \leftarrow \begin{bmatrix} I_N & 0^{N\times N} \end{bmatrix}$
$d_1 \leftarrow d_2 \leftarrow \cdots \leftarrow d_N \leftarrow 0$; $d_{N+1} \leftarrow d_{N+2} \leftarrow \cdots \leftarrow d_{2N} \leftarrow 1$
$t \leftarrow -1$
$\beta \leftarrow 1$
$\sigma \leftarrow 0$
$\mu \leftarrow 0$.

MBM2 while $\beta < \delta - \sigma + \mu + 1$ do steps MBM3 through MBM9

MBM3 $t \leftarrow t + 1$

MBM4 $\Delta \leftarrow \mathit{Coeff}\,(t; M(z)\cdot f(z))$
Note that $\Delta = \begin{bmatrix} C & L^{[t-1]} \end{bmatrix}$ where $C$, $L^{[t-1]} \in K^{N\times N}$ and $L^{[t-1]}$ is lower triangular with rank $r_{t-1}$ computed in the previous step MBM5 below. By definition, we set $L^{[-1]} = 0^{N\times N}$.

MBM5 Call Algorithm 2.2.2 below to compute $\tau$. As a side-effect, $d_1, \ldots, d_{2N}$ are updated. Let $\Delta\cdot\tau = \begin{bmatrix} Z & L^{[t]} \end{bmatrix}$. We have that $Z \in K^{N\times N}$ has $r_t - r_{t-1}$ nonzero columns and if $j$ is the index of a nonzero column of $Z$, then $d_j = t+1$. Furthermore, $L^{[t]} \in K^{N\times N}$ is lower triangular of rank $r_t$.

MBM6 $d_{N+i} = d_{N+i} + 1$ for all $i = 1, 2, \ldots N$

MBM7 $\beta \leftarrow \min_{N+1\le i\le 2N} d_i$; $\quad \mu \leftarrow \max_{1\le i\le N} d_i$; $\quad \sigma \leftarrow \sum_{i=1}^{N} d_i$
These updates are most efficiently performed during Algorithm 2.2.2 below.

MBM8 if $\sigma \ge \delta + 1$ then return "insufficient bound"

MBM9 $f(z) \leftarrow f(z)\cdot\tau\cdot \mathrm{diag}(I_N, z\cdot I_N)$

MBM10 $F \leftarrow \begin{bmatrix} z^{d_1} f_1(z^{-1}) & z^{d_2} f_2(z^{-1}) & \ldots & z^{d_N} f_N(z^{-1}) \end{bmatrix}$

MBM11 return $F$

### 2.2.2 Auxiliary Gaussian Elimination Algorithm

**Input** $\Delta \in K^{N\times 2N}$ where $\Delta$ has rank $R$ and the last $N$ columns of $\Delta$ have rank $r$.
$d_1, d_2, \ldots, d_{2N}$ nominal degrees of the columns.

**Output** $\tau \in K^{2N\times 2N}$ a transformation such that $\Delta\tau = \begin{bmatrix} Z & L \end{bmatrix}$. $Z$ will be a matrix with $R-r$ nonzero columns. $L$ is a lower triangular matrix of rank $R$.
$d_1, d_2, \ldots, d_{2N}$ a reordering of the nominal degrees. Note that no $d_j$ for $1 \le j \le N$ is decreased.

GE1 $\tau \leftarrow I_{2N}$

GE2 $\Gamma \leftarrow \{1, 2, \ldots, N\}$

GE3 for $i = 1$ to $N$ do steps GE4 through GE24

In [14,p. 36] four types of eliminations and updates are distinguished. We will annotate which steps of our Gaussian elimination algorithm correspond to each case. Except for Case 1 of [14], the updates defined in each case can be performed as $i$ ranges from 1 to $N$. For a fixed $i$ however, only one type of update will be performed. Case 1 in [14] requires the same type of update for each $i$.

GE4 $\Pi_i \leftarrow \{j \in \Gamma \text{ and } \Delta_{i,j} \neq 0\} \cup \{N+i\}$

GE5 $l \leftarrow l \in \Pi_i \mid d_l = \min_{i\in\Pi_i} d_i$

There may be different choices for the column index $l$ of the pivot element $\Delta_{i,l}$. Our current implementation employs the following rules. Assume that $d_r$ and $d_s$ are equal and minimal with $r, s \in \Pi_i$ and $r < s$. If $1 \le r < s \le N$ then $l = r$. If $1 \le r \le N$ and $N+1 \le s \le 2N$, then $l = s$.

GE6 $\Pi_i \leftarrow \Pi_i \setminus \{l\}$

GE7 for all $j \in \Pi_i$ do steps GE8 through GE24

Case 1 of [14] occurs if $\Pi_i$ is empty for all $1 \le i \le N$. Then $\tau = I_{2N}$ and so none of the generating vectors are updated. This situation only occurs when $C$ defined in step MBM4 is $0^{N\times N}$.

GE8 if $l = N + i$ do steps GE9 and GE10

These steps perform a minor change of the generating vectors, as defined in Case 3 of [14]. The nominal degrees of all the generating vectors are unchanged.

GE9 Perform the column operation $\Delta_j \leftarrow \Delta_j - \frac{\Delta_{i,j}}{\Delta_{i,N+i}}\Delta_{N+i}$
Now $\Delta_{i,j} = 0$.

GE10 Perform the column operation $\tau_j \leftarrow \tau_j - \frac{\Delta_{i,j}}{\Delta_{i,N+i}}\tau_{N+i}$

GE11 if $l < N + i$ do steps GE12 through GE24

At this stage, the pivot column is a previous generator column, and so $\sigma$ will be increased by step GE20 or step GE23. If $\sigma - d_l + d_{N+i} > \delta$ then any further calculations are unnecessary since step MBM8 will return "insufficient bound".

GE12 if $j < N + i$ do steps GE13 and GE14

GE13 Perform the column operation $\Delta_j \leftarrow \Delta_j - \frac{\Delta_{i,j}}{\Delta_{i,l}}\Delta_l$
Now $\Delta_{i,j} = 0$.

GE14 Perform the column operation $\tau_j \leftarrow \tau_j - \frac{\Delta_{i,j}}{\Delta_{i,l}}\tau_l$

GE15 if $j = N + i$ do steps GE16 through GE24

GE16 if $\Delta_{i,N+i} \neq 0$ do steps GE17 through GE20

The update performed here is defined as a major change in Case 4 of [14]. The nominal degree of one vector generator will be increased and an auxiliary vector will be replaced by a former generating vector.

GE17 Perform the column operation $\Delta_{N+i} \leftarrow \frac{-\Delta_{i,l}}{\Delta_{i,N+i}}\Delta_{N+i} + \Delta_l$

GE18 Perform the column operation $\tau_{N+i} \leftarrow \frac{-\Delta_{i,l}}{\Delta_{i,N+i}}\tau_{N+i} + \tau_l$

GE19 Switch column $l$ with column $N + i$ in $\Delta$ and $\tau$

GE20 Switch $d_l$ and $d_{N+i}$.
Now $\Delta_{i,l} = 0$ and $d_{N+i}$ is minimized.

GE21 if $\Delta_{i,N+i} = 0$ do steps GE22 through GE24

GE22 Perform the column operation $\tau_{N+i} \leftarrow \tau_{N+i} + \tau_l$

GE23 Switch $d_l$ and $d_{N+i}$.

GE24 $\Gamma \leftarrow \Gamma \setminus \{l\}$

Note that steps GE22 through GE24 initialize a new column in the auxiliary polynomials. If $\Delta_{i,N+i} = 0$, then a new component as defined by Case 2 of [14] has been found and a new auxiliary vector must be added. The corresponding discrepancy is skipped over via $d_l = t + 1$, necessitated by an increase of $r_t$. If $M_0$ is non-singular [12], the initialization only occurs for $t = 0$.

GE25 Return $\tau$ and $d_1, d_2, \ldots, d_{2N}$.

## 2.3 Properties and Correctness of the Matrix Berlekamp / Massey Algorithm

### 2.3.1 Preliminary Invariants

We now establish several properties of the computed quantities that remain invariant during the iterations. In order to distinguish the values of variables at different iterations, we will use the superscript $[t]$ for the value of a variable for a given loop index $t$ at step MBM2.

**Lemma 1** *For each iteration at step* MBM2 $\deg(f_j^{[t]}) \leq d_j^{[t]}$.

**Proof.** Assuming $\deg(0^N) \leq 0$, then this is true by construction on initialization.

We proceed by induction. Suppose $t \geq -1$ and $\deg(f_j^{[t]}) \leq d_j^{[t]}$ for all $1 \leq j \leq 2N$. Steps MBM5 and MBM6 update the nominal degrees while step MBM9 updates $f$. These steps can be performed within the iteration steps GE3 and GE7. Note that each auxiliary column $f_{N+i}$ is updated only once after which multiplication by $z$ and incrementing $d_{N+i}$ can be performed. So we can analyze each column operation performed by Algorithm 2.2.2 to prove the property is maintained.

During each iteration of GE7, $l$ is the index of the pivot column. Algorithm 2.2.2 performs two types of column operations, adding a scalar multiple of the pivot column into another column and switching column $l$ with column $N + i$. The latter operation preserves the property, since the column switch in step GE19, is followed by a nominal degree switch

in step GE20. Since $l$ is minimal in nominal degree, then steps GE10, GE14, GE18 and GE22 preserve the property in the following manner: $\deg(f_j^{[new]}) \leq \max\{\deg(f_j^{[old]}), \deg(f_l)\} \leq \max\{d_j, d_l\} = d_j$. Step GE23 minimizes the nominal degree of the auxiliary polynomial but it preserves the property since $\deg(f_l) \leq d_l \leq d_{N+i}$. Finally, when column $N+i$ is multiplied by $z$, its degree increases by 1, but its nominal degree is also incremented by 1 and so the property is maintained. So for $1 \leq j \leq 2N$, $\deg(f_j^{[t+1]}) \leq d_j^{[t+1]}$. ■

**Lemma 2** *For each iteration at step* MBM2 *one has* $\sum_{i=1}^{2N} d_j = N \cdot (t+2)$.

**Proof.** For $t = -1$ the property is true by the initialization in step MBM1. Because the last $N$ columns of $f(z)$ have their nominal degrees incremented by 1 in step MBM6, the summation holds by induction for all $t \geq 0$. ■

**Lemma 3** *For each iteration at step* MBM2 *one has*

$$\det([f_1(0) \quad f_2(0) \quad \ldots \quad f_N(0)]) \neq 0.$$

**Proof.** This is true by construction on initialization. At step MBM9, the elementary column operations encoded in $\tau$ are applied to the constant coefficient, thus leaving it non-singular. ■

**Lemma 4** *For each iteration at step* MBM2 *the following equivalence is true for each matrix* $L$ *produced by Algorithm 2.2.2 in step* MBM5*:*

$$(\forall i, 1 \leq i \leq N)\colon L_{i,i} \neq 0 \Longleftrightarrow L_i \neq 0^N \Longleftrightarrow f_{N+i} \neq 0^N,$$

*where* $L_i$ *is the* $i$*-th column of* $L$*.*

**Proof.** At every stage, $L_{i,i} \neq 0 \Longrightarrow L_i \neq 0^N$ is apparent. Also, since $L_{N+i} = \mathit{Coeff}(t+1; M(z) \cdot f_{N+i}(z))$, $L_i \neq 0^N \Longrightarrow f_{N+i} \neq 0^N$. We will show by induction that at every stage $t \geq -1$, $f_{N+i} \neq 0^N \Longrightarrow L_{i,i} \neq 0$.

If $t = -1$, then for all $1 \leq i \leq N$, $f_{N+i}^{[-1]} = 0^N$. By definition $L^{[-1]} = 0^{N \times N}$ and so the induction basis is immediate.

Suppose $t \geq -1$ and $f_{N+i}^{[t]} \neq 0^N \Longrightarrow L_{i,i}^{[t]} \neq 0$ for all $i$. Let $1 \leq i \leq N$ be such that $f_{N+i}^{[t+1]} \neq 0^N$. If $f_{N+i}^{[t+1]} = z \cdot f_{N+i}^{[t]}$ then $L_{i,i}^{[t+1]} = L_{i,i}^{[t]} \neq 0$ by the induction hypothesis. So now we consider the possibility that $f_{N+i}^{[t+1]} \neq z \cdot f_{N+i}^{[t]}$.

If $f_{N+i}^{[t+1]} \neq z \cdot f_{N+i}^{[t]}$, then we know that $f_{N+i}^{[t+1]} = z \cdot \sum_{j=1}^{N+i-1}(\alpha_j f_j^{[t]})$. Since auxiliary columns are never interchanged in Algorithm 2.2.2, then there exists $1 \leq j \leq N$ such that $\alpha_j \neq 0$. Because column $N+i$ of $f^{[t]}$ was replaced then at stage $i$ of Algorithm 2.2.2, there was an $1 \leq l \leq N$ such that $\Delta_{i,l}^{[t+1]} \neq 0$ and column $l$ had minimal nominal degree. Noted that $\Delta^{[t+1]}$ here is not the original matrix passed to Algorithm 2.2.2, but the eliminated form being computed by the algorithm. So for all $1 \leq m < i$, we have $\Delta_{m,l}^{[t+1]} = 0$. Since column $l$ of $\Delta^{[t+1]}$ has a discrepancy in row $i$ and has minimal nominal degree, then either step GE19 or step GE22 will place the contents of column $l$ into column $N+i$ of $\Delta^{[t+1]} \cdot \tau^{[t+1]}$. At the completion of Algorithm 2.2.2, column $i$ of $L^{[t+1]}$ is equal to column $N+i$ of $\Delta^{[t+1]} \cdot \tau^{[t+1]}$. Thus $L_{i,i}^{[t+1]} \neq 0$ since $\Delta_{i,l}^{[t+1]} \neq 0$, completing the induction argument. ■

**Lemma 5** *For each iteration at step* MBM2 *the nonzero columns of $L$ form a basis for the column space of $[M_0 \quad M_1 \quad \ldots \quad M_t]$.*

**Proof.** Suppose $t = -1$. By definition, $L^{[-1]} = 0^{N \times N}$, meaning there are no nonzero columns. Since the matrix $M_{-1}$ is empty, its column space is empty. Therefore the invariant holds.

Suppose $t \geq -1$ and suppose the lemma holds for $t$. Step MBM4 defines $\Delta^{[t+1]} = \begin{bmatrix} C^{[t+1]} & L^{[t]} \end{bmatrix}$. The induction hypothesis allow us to write $C^{[t+1]}$ as

$$\begin{aligned} C^{[t+1]} &= M_{t+1}[f_1(0) \ldots f_N(0)] + \sum_{i=1}^{t+1} M_{t+1-i}\, Coeff\,(i; [f_1(z) \ldots f_N(z)]) \\ &= M_{t+1}[f_1(0) \ldots f_N(0)] + L^{[t]}\Xi \quad \text{for some } \Xi \in K^{N \times N}. \end{aligned}$$

By hypothesis and Lemma 3 we deduce that the columns of $\Delta^{[t+1]}$ span the columnspace of $[M_0 \ldots M_{t+1}]$. Algorithm 2.2.2 performs elementary column operations so the columns of $[Z \quad L^{[t+1]}]$ span the columnspace of $[M_0 \ldots M_{t+1}]$ as well. We need to argue that the non-zero columns in $Z$ are not needed. In Step GE21 any non-zero column of $Z$ replaces by Lemma 4 a zero column of $L$ and is then removed from further updates in Step GE24. Therefore the remaining columns in $Z$ and the columns of $L$ still span $[M_0 \ldots M_{t+1}]$. However, at the conclusion of Algorithm 2.2.2 all remaining columns in $Z$ are zero. Since $L^{[t+1]}$ is triangular, its non-zero columns are linearly independent. ■

**Lemma 6** *For each iteration at step* MBM2 *we have $r_t$ (see Definition 6 on page 11) $= |\{j \mid N+1 \leq j \leq 2N$ and $f_j \neq 0^N\}|$ (the number of non-zero auxiliary polynomials).*

**Proof.** For all $N+1 \leq j \leq 2N$, Lemma 4 implies that $f_j^{[t]} \neq 0^N$ if and only if $L_{j-N}^{[t]} \neq 0^N$. Lemma 5 implies that $L^{[t]}$ has exactly $r_t$ nonzero columns. So there are exactly $r_t$ indices $j$ such that $N+1 \leq j \leq 2N$ and $f_j^{[t]} \neq 0^N$. ■

**Lemma 7** *For each iteration at step* MBM2 *for all $l$ such that $N+1 \leq l \leq 2N$, $f_l = 0^N$ if and only if $d_l = t+2$.*

**Proof.** If $t = -1$ then $d_l^{[-1]} = 1 = t+2$ and $f_l^{[-1]} = 0^N$.

Let $t \geq -1$ and suppose the invariant holds for $t$.

The update of $f$ performed in step MBM9 dependent on the computation of $\tau^{[t+1]}$ by Algorithm 2.2.2 in step MBM 5 forces two possibilities for the update of $f_l^{[t+1]}$. Either $f_l^{[t+1]} = z \cdot f_l^{[t]}$ or $f_l^{[t+1]} = z \cdot \sum_i (\alpha_i \cdot f_i^{[t]})$ where $i \leq l$, $f_i^{[t]} \neq 0^N$ and $\alpha_i \neq 0$ for all $i$. The proof of Lemma 4 illustrates that Algorithm 2.2.2 adjusts the column $L_{l-N}^{[t]}$ if and only if $L_{l-N}^{[t+1]} \neq 0^N$. So $L_{l-N}^{[t+1]} = L_{l-N}^{[t]} = 0^N$ implies that $f_l^{[t+1]} = z \cdot f_l^{[t]}$. $L_{l-N}^{[t+1]} \neq 0^N$ implies that $f_l^{[t+1]} = z \cdot \sum_i (\alpha_i \cdot f_i^{[t]})$. Further, Lemma 4 implies $L_{l-N}^{[t+1]} = 0^N$ if and only if $f_l^{[t+1]} = 0^N$.

So $f_l^{[t+1]} = 0^N$ if and only if $L_{l-N}^{[t+1]} = 0^N$ if and only if $L_{l-N}^{[t]} = 0^N$. By Lemma 4 and the previous paragraph, $f_l^{[t+1]} = 0^N$ implies that $f_l^{[t]} = 0^N$ and $f_l^{[t+1]} = z \cdot f_l^{[t]}$. Thus $d_l^{[t+1]} = d_l^{[t]} + 1 = t+2+1 = t+3$ by the induction hypothesis.

Conversely, $f_l^{[t+1]} \neq 0^N$ if and only $L_{l-N}^{[t+1]} \neq 0^N$. So $f_l^{[t+1]} = z \cdot \sum_i (\alpha_i \cdot f_i^{[t]})$ and the induction hypothesis implies that $d_i^{[t]} < t+2$ for all $i$. Thus the $d_l^{[t+1]} \neq t+3$. ■

**Lemma 8** *For each iteration at step* MBM2 *we have*

$$\sum_{1 \leq j \leq 2N \text{ and } f_j \neq 0^N} d_j = r_t \cdot (t+2).$$

**Proof.** The invariant holds at $t = -1$ by initialization. By definition $r_{-1} = 0$ and the $N$ nonzero columns of $f^{[-1]}$ are the first $N$ columns and each column has nominal degree 0.

By Lemma 5 there are $N - r_t$ zero columns, each of which by Lemma 7 has nominal degree $t+2$. Therefore by using Lemma 2 we have the following equation:

$$\begin{aligned} \sum_{1 \leq j \leq 2N \text{ and } f_j \neq 0^N} d_j &= (\sum_{j=1}^{2N} d_j) - (N - r_t) \cdot (t+2) \\ &= N \cdot (t+2) - (N - r_t) \cdot (t+2) = r_t \cdot (t+2). \end{aligned}$$

■

**Lemma 9** *For each iteration at step* MBM2 *one has*

$$(\forall j, 1 \leq j \leq 2N)(\forall l, d_j \leq l \leq t)\colon \mathit{Coeff}(l; M(z) \cdot f_j(z)) = 0^N. \tag{2.1}$$

**Proof.** This invariant is one of the main conditions in [12,p. 338].

The invariant is true by default at initialization i.e. $t = -1$ since every column $j$ has $d_j^{[-1]} > -1$ and so the range is empty.

Let $t \geq -1$ and assume the invariant holds at $t$. Let $j$ be such that $1 \leq j \leq 2N$. We consider two cases, $j \leq N$ and $j > N$.

If $j > N$, then $f_j^{[t+1]} = z \cdot \sum_i \alpha_i f_i^{[t]}$ and $d_j^{[t+1]} = d_m^{[t]} + 1$ where $m$ is the column of maximal nominal degree in the linear combination. The induction hypothesis implies that for all $i$ in the linear combination and all $l$ such that $d_m^{[t]} \leq l \leq t$ we have $\mathit{Coeff}(l; M(z) \cdot f_i^{[t]}(z)) = 0^N$. Therefore, for all $l$ such that $d_j^{[t+1]} \leq l \leq t+1$ we have $\mathit{Coeff}(l; M(z) \cdot f_j^{[t+1]}(z)) = \mathit{Coeff}(l-1; M(z) \cdot \sum_i \alpha_i f_i^{[t]}(z)) = 0^N$.

If $j \leq N$, then $f_j^{[t+1]} = \sum_i \alpha_i \cdot f_i^{[t]}$. If $d_j^{[t+1]} = t+2$, then the condition holds trivially as in the base case. If $d_j^{[t+1]} \leq t+1$ then $d_j^{[t+1]} = d_m^{[t]}$ where $m$ is the column in the linear combination with maximal nominal degree. The induction hypothesis implies that for all $i$ in the linear combination and all $l$ such that $d_m^{[t]} \leq l \leq t$ then $\mathit{Coeff}(l; M(z) \cdot f_i^{[t]}(z)) = 0^N$. Therefore $\mathit{Coeff}(l; M(z) \cdot f_j^{[t+1]}(z)) = 0^N$ for all $l$ such that $d_j^{[t+1]} \leq l \leq t$. Further, since $d_j^{[t+1]} < t+2$, then column $j$ of $Z^{[t+1]}$ is $0^N$. Thus $\mathit{Coeff}(t+1; M \cdot f_j^{[t+1]}(z)) = Z_j^{[t+1]} = 0^N$. So for all $l$ such that $d_j^{[t+1]} \leq l \leq t+1$ we have $\mathit{Coeff}(l; M \cdot f_j^{[t+1]}(z)) = 0^N$.

Thus (2.1) holds at stage $t+1$ and by induction the invariant is true for every $t$.

■

We will make use of the following vectors.

**Definition 7** *If* $v \in K[z]^N = \sum_{i=0}^m v_i z^i$, $v_i \in K^N$, *then let* $\mathit{CoeffVec}(d; v)$ *where* $d \geq m$ *be a block vector in* $K^{N(d+1)}$ *of the following form:*

$$\mathit{CoeffVec}(d; v) = \begin{bmatrix} 0^N \\ \vdots \\ 0^N \\ v_m \\ v_{m-1} \\ \vdots \\ v_0 \end{bmatrix} \begin{matrix} \left.\begin{matrix} \\ \\ \\ \\ \end{matrix}\right\} d-m \text{ blocks} \\ \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} m+1 \text{ blocks} \end{matrix} .$$

These vectors are generalizations of coefficient vectors. One can think of them as embedding the coefficient vector of a vector polynomial into a larger coefficient vector. Further, the block entries of the coefficient vector are vectors in $K^N$.

**Lemma 10** *For each iteration at step* MBM2 *for* $d = \min\{\max_{1\le j\le 2N}\{d_j\}, t+1\}$ *the set of vectors in* $K^{N(d+1)}$ *defined by*

$$U_t = \{CoeffVec(d; z^i f_j) \mid 1 \le j \le 2N \text{ and } f_j \neq 0^N, 0 \le i \le d - d_j\},$$

*is linearly independent.*

**Proof.** We proceed by induction on $t$.

So for $t = -1$ at initialization, $d = 0$ and the vectors in $U_{-1}$ are the columns of the matrix $I_N$.

Let $t \ge -1$ and suppose the invariant holds at stage $t$. After the $t+1$ stage of the algorithm we can construct a matrix $\tau$ that is derived from $\tau^{[t]}$ and is a product of the corresponding elementary column operations on the shifted coefficient vectors of the generator and the auxiliary polynomials. Let $\tilde{U}_{t+1} = U_t\tau$. Here $U_t$ is considered to be a matrix with the appropriate columns. By hypothesis, the columns of $\tilde{U}_{t+1}$ are linearly independent.

If $d$ is unchanged at the $t+1$ stage, then $U_{t+1} \subset \tilde{U}_{t+1}$. So $U_{t+1}$ is linearly independent.

If $d$ changes, then it can only increase by one. Further, since $N$ polynomials did not increase in nominal degree, we must have $|U_{t+1}| = |U_t| + N$. We can reconstruct $U_{t+1}$ as the following matrix:

$$\begin{bmatrix} 0^{N\times N} & \tilde{U}_{t+1} \\ F & 0^{N\times|U_t|} \end{bmatrix},$$

where

$$F \in K^{Nd\times N} = \left[CoeffVec(d-1; f_1^{[t+1]}) \quad \ldots \quad CoeffVec(d-1; f_N^{[t+1]})\right].$$

We leave out the proof that this is in fact $U_{t+1}$.

Now consider the following linear system:

$$\begin{bmatrix} 0^{N\times N} & \tilde{U}_{t+1} \\ F & 0^{N\times|U_t|} \end{bmatrix} \begin{bmatrix} c_1 \\ \vdots \\ c_N \\ c_{N+1} \\ \vdots \\ c_{N+|U_t|} \end{bmatrix} = 0^{N(d+1)}.$$

By Lemma 3, we know that $c_1 = c_2 = \cdots = c_N = 0$ and the induction hypothesis tells us that $c_{N+1} = \cdots = c_{N+|U_t|} = 0$. Further since $U_{t+1}$ is equal to the columns of the matrix above, $U_{t+1}$ is linearly independent. ■

**Corollary 1** *Let $d' \leq d$ as defined in Lemma 10. The set of vectors CoeffVec$(d'; z^i f_j(z))$, where $1 \leq j \leq 2N$ and $f_j \neq 0^N$, $0 \leq i \leq d' - d_j$, is linearly independent.*

**Proof.** The vectors are formed from vectors in $U_t$ by removing top segments that are zero. ■

**Lemma 11** *For each iteration at step* MBM2 *for all $j, d, i$ with $1 \leq j \leq 2N$, $d_j \leq d \leq t$ and $0 \leq i \leq d - d_j$, CoeffVec$(d; z^i f_j(z))$ is in the right nullspace of the following block Hankel matrix:*

$$H = \begin{bmatrix} M_0 & M_1 & \cdots & M_d \\ M_1 & M_2 & \cdots & M_{d+1} \\ \vdots & \vdots & \vdots & \vdots \\ M_{t-d} & \cdots & \cdots & M_t \end{bmatrix}.$$

**Proof.** This proof relies on Lemma 9. First, let $c_0, c_1, \ldots, c_{d_j}$ be the coefficients of $f_j$. Then the following holds:

$$H \cdot CoeffVec(d; z^i f_j) = H \cdot \begin{bmatrix} 0^N \\ \vdots \\ 0^N \\ c_{d_j} \\ c_{d_j-1} \\ \vdots \\ c_0 \\ 0^N \\ \vdots \\ 0^N \end{bmatrix} \begin{matrix} \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} d - d_j - i \\ \left.\begin{matrix} \\ \\ \\ \\ \\ \end{matrix}\right\} d_j + 1 \\ \left.\begin{matrix} \\ \\ \\ \end{matrix}\right\} i \end{matrix} = \begin{bmatrix} \sum_{l=0}^{d_j} M_{d-i-l} c_l \\ \sum_{l=0}^{d_j} M_{d+1-i-l} c_l \\ \vdots \\ \sum_{l=0}^{d_j} M_{t-i-l} c_l \end{bmatrix}$$

$$= \begin{bmatrix} Coeff\left((d-i; M(z)\, f_j(z)\right) \\ Coeff\left(d+1-i; M(z)\, f_j(z)\right) \\ \vdots \\ Coeff\left(t-i; M(z)\, f_j(z)\right) \end{bmatrix}$$

Since $d_j \leq d - i < d + 1 - i < \cdots < t - i < t + 1$, Lemma 9 implies that every coefficient in the vector above is $0^N$. ■

### 2.3.2 Nullspace theorem

**Theorem 4** *For each iteration at step* MBM2 *let* $d = \min\{\max\{\mu, \beta - 1\}, t\}$. *Then the right nullspace of the block Hankel matrix given by*

$$H_t = \begin{bmatrix} M_0 & M_1 & \cdots & M_d \\ M_1 & M_2 & \cdots & M_{d+1} \\ \vdots & \vdots & \vdots & \vdots \\ M_{t-d} & \cdots & \cdots & M_t \end{bmatrix} \tag{2.2}$$

*has a basis defined by the set*

$$V_t = \{CoeffVec(d; z^i f_j(z)) \mid 1 \leq j \leq 2N \text{ and } d_j \leq d, 0 \leq i \leq d - d_j\}. \tag{2.3}$$

*Further, for all $l$ with $N+1 \leq l \leq 2N$ and $d < d_l \leq t$, any wider and shallower block Hankel matrix given by*

$$H_{t,d_l} = \begin{bmatrix} M_0 & M_1 & \cdots & M_{d_l} \\ M_1 & M_2 & \cdots & M_{d_l+1} \\ \vdots & \vdots & \vdots & \vdots \\ M_{t-d_l} & \cdots & \cdots & M_t \end{bmatrix} \tag{2.4}$$

*has a right nullspace basis*

$$V_{t,d_l} = \{CoeffVec(d_l; z^i f_j(z)) \mid 1 \leq j \leq 2N \text{ and } d_j \leq d_l, 0 \leq i \leq d_l - d_j\}. \tag{2.5}$$

**Proof.** By Corollary 1 on page 21 the vectors in $V_t$ and $V_{t,d_l}$ are linearly independent, and by Lemma 11 on page 21 they are in the nullspace of $H_t$ and $H_{t,d_l}$, respectively. We proceed by induction to show that the vectors indeed span the corresponding nullspaces.

If $t = -1$ then the invariant is trivially true. Obviously, since we may set $H_{-1} = [0]$ whose nullspace basis is the empty set. But $d = t = -1$ and $d_j^{[-1]} > -1$ for all $j$, so $V_{-1} = \emptyset$.

The case $t = 0$ can be proven as a base case as well. Such a proof would be a specialized version of the proof for any $t \geq 0$ where $d = t$. Thus we will use $t = -1$ as our base case for induction.

Suppose $t \geq -1$ and the theorem holds for $t$. To prove the theorem holds at $t+1$, we must break the update of the algorithm into the various cases which can take place. Let $\mu^{[t+1]}$ and $\beta^{[t+1]}$ as computed by Algorithm 2.2.1 and $d$ as stated in the theorem. The first separation of cases is a condition on $d$. The case $d = t+1$ can be viewed as an extension of the base case. If $d < t+1$, we will then break the proof down into two cases, each based on $\mu^{[t+1]}$ and $\beta^{[t+1]}$. These cases are: $\mu^{[t+1]} + 1 < \beta^{[t+1]}$, $\mu^{[t+1]} + 1 \geq \beta^{[t+1]}$. The latter has two subcases, $\mu^{[t]} = \mu^{[t+1]}$ and $\mu^{[t]} \neq \mu^{[t+1]}$. The four cases are exhaustive.

**Case 1:** $d = t+1$**.**

Then $H_{t+1} = \begin{bmatrix} M_0 & M_1 & \cdots & M_{t+1} \end{bmatrix}$. By Lemma 6 $H_{t+1}$ has rank $r_{t+1}$. We show that $|V_{t+1}| = N(t+2) - r_{t+1}$, which is the full dimension of the nullspace of $H_{t+1}$ and proves that Case. Obviously, the cardinality of $V_{t+1}$ depends on counting the non-negative integers in the range from 0 to $t+1-d_j^{[t+1]}$ for each nonzero column $f_j^{[t+1]}$. This is true even for any $j$ such that $d_j^{[t+1]} > t+1$. If $d_j^{[t+1]} > t+1$, then $d_j^{[t+1]} = t+2$ and the number of non-negative integers in the range of above is $t+1-d_j^{[t+1]}+1 = t+2-(t+2) = 0$, and so can be included in a full summation without causing problems. By Lemma 3 on page 16

and Lemma 6 on page 17 there are $N + r_{t+1}$ nonzero columns of $f^{[t+1]}$. These facts and Lemma 8 on page 18 give us the following:

$$\begin{aligned}|V_{t+1}| &= \sum_{1\le j\le 2N \text{ and } f_j^{[t+1]}\neq 0^N} (t+1-d_j^{[t+1]}+1) \\ &= (N+r_{t+1})(t+2) - \sum_{1\le j\le 2N \text{ and } f_j^{[t+1]}\neq 0^N} d_j^{[t+1]} \\ &= (N+r_{t+1})(t+2) - r_{t+1}(t+3) = N(t+2) - r_{t+1}.\end{aligned}$$

**Case 2:** $d < t+1$ **and** $\mu^{[t+1]} + 1 < \beta^{[t+1]}$**.**

Then $d = \beta^{[t+1]} - 1$. The condition $\mu^{[t+1]} + 1 < \beta^{[t+1]}$ implies that $\Delta^{[t+1]} = [0^{N\times N} \quad L]$, for otherwise Algorithm 2.2.2 would have made a corresponding update on the nominal degrees. Therefore $\mu^{[t+1]} = \mu^{[t]}$ and the first $N$ columns of $f(z)$ are unchanged during the update. Furthermore, $\beta^{[t+1]} = \beta^{[t]} + 1$ and $d$ is incremented by 1, so $H_{t+1}$ is formed from $H_t$ by adding on right block column. Now, for all $j$ with $1 \le j \le N$ the range $0 \le i \le d - d_j^{[t+1]}$ has increased in cardinality by 1, so there are $N$ more vectors in $V_{t+1}$. Since that is the maximal increase for the nullspace, $V_{t+1}$ forms a basis.

The same proof applies to all $H_{t+1,\eta}$, where $d < \eta = d_j^{[t+1]} = d_j^{[t]} + 1 < t+2$ for some $N+1 \le j \le 2N$.

**Case 3:** $d < t+1$**,** $\mu^{[t+1]} + 1 \ge \beta^{[t+1]}$ **and** $\mu^{[t+1]} = \mu^{[t]}$**.**

Then $d = \mu^{[t]}$ and has not changed, so $H_{t+1}$ is formed from $H_t$ by adding at the bottom one block row. Let $w \in K^{N(d+1)}$ be an element of the right nullspace of $H_{t+1}$. Consider $w$ to be a degree $d$ coefficient vector with a corresponding zero constant coefficient. By Lemma 3 on page 16 we may assume that the corresponding constant coefficient of $w$ is zero.

Then $w$ is also in the right nullspace of $H_t$. From the induction hypothesis, $w$ can be written as a linear combination of vectors from $V_t$. Therefore let

$$w = \sum \alpha_\rho v_\rho, \tag{2.6}$$

where for all $\rho$, $v_\rho \in V_t$. Thus $v_\rho = \mathit{CoeffVec}(d; z^i f_j^{[t]}(z))$, for some $j$ with $1 \le j \le 2N$ and some $i$ in the proper range. We first show that for all $\rho$ we have $i \ge 1$. The constant coefficients of the auxiliary polynomials are always zero, so by Lemma 3 the generating polynomials must be shifted by at least $z$ so the corresponding constant coefficient of $w$ can be zero. For $j$ with $N+1 \le j \le 2N$ and $f_j^{[t]} \neq 0$ we know that $\mathit{Coeff}(t+1; M(z)\cdot f_j^{[t]}(z)) =$

$L_j^{[t]}$, which are non-zero and linearly independent (see Lemma 4). Because the shifted vectors corresponding to the columns of $f^{[t]}$ by hypothesis zero the new block row in $H_{t+1}$, if any $CoeffVec(d; f_j^{[t]})$ were in the linear combination, then $w$ could not be a nullspace element of $H_{t+1}$. Note that the argument shows that at least one vector of $V_t$ cannot be in the nullspace of $H_{t+1}$, hence the rank of $H_{t+1}$ is at least one more than the rank of $H_t$.

We can carry out the steps MBM9 and MBM6 in Algorithm 2.2.1 within the iteration steps GE3 and GE7 in Algorithm 2.2.2. Note that each column $f_{N+i}$ is updated only once, after which multiplication by $z$ can be performed. We shall show that at each iteration at step GE7 the vector $w$ in (2.6) remains in the span of $V$ in (2.3). We shall use the superscripts $[old]$ and $[new]$ for the contents of variables before and after each iteration in step GE7. From the updates $f_j^{[new]} = f_j^{[old]} + \gamma f_l$ in steps GE10 and GE14 we obtain

$$CoeffVec(d; z^i f_j^{[old]}) = CoeffVec(d; z^i f_j^{[new]}) - \gamma CoeffVec(d; z^i f_l). \tag{2.7}$$

Since $d_l \le d_j^{[old]} = d_j^{[new]}$, for all $i$ in the range $1 \le i \le d - d_j^{[old]}$ the two vectors on the right-side of (2.7) are in the new set $V_{new}$ of (2.3). Hence, all vectors in $V_{old}$ of (2.3) are still spanned by $V_{new}$ and $w$ remains in the span of $V_{new}$.

From the updates

$$f_l^{[new]} = \tfrac{1}{\gamma} f_{N+i}^{[old]} + f_l^{[old]},\ d_l^{[new]} = d_{N+i}^{[old]}, \quad f_{N+i}^{[new]} = z f_l^{[old]},\ d_{N+i}^{[new]} = d_l^{[old]} + 1$$

in steps GE18–GE20 and subsequent multiplication with $z$, we obtain $CoeffVec(d; z^i f_l^{[old]}) = CoeffVec(d; z^{i-1} f_{N+i}^{[new]})$ and

$$CoeffVec(d; z^i f_{N+i}^{[old]}) = \gamma CoeffVec(d; z^i f_l^{[new]}) - \gamma CoeffVec(d; z^{i-1} f_{N+i}^{[new]}). \tag{2.8}$$

Again since $d_l^{[old]} \le d_{N+i}^{[old]}$, the required old range for $i$ is included in the ranges of $i$ and $i-1$ of the new right-side vectors in (2.8).

Finally, steps GE22–GE24 cannot occur because $\mu^{[t+1]} = d < t+1$. Therefore, at the conclusion of all updates for $f$ the vector $w$ is remains the linear span of $V_{t+1}$, as was to be shown.

We now show that for all $\eta$ such that $\mu^{[t+1]} = d < \eta = d_j^{[t+1]} < t+2$ where $N+1 \le j \le 2N$, the set defined in (2.5) on page 23 is a nullspace basis of (2.4). We know that $\eta = d_j^{[t+1]} = d_\xi^{[t]} + 1$ for some $1 \le \xi \le 2N$. Further, since $d_\xi^{[t]} + 1 > \mu^{[t+1]} = \mu^{[t]}$, then $d_\xi^{[t]} \ge \mu^{[t]}$. Therefore $j = \xi$ and $f_j^{[t+1]} = z \cdot f_j^{[t]}$, for otherwise we would have $d = \mu^{[t+1]} > \mu^{[t]}$. So $\eta = d_j^{[t+1]} = d_j^{[t]} + 1 > \mu^{[t+1]}$. Since $\eta - 1 \ge \mu^{[t]}$, then the induction hypothesis implies

that the set $V_{t,\eta-1}$ is a nullspace basis of the matrix $H_{t,\eta-1}$. Note that $H_{t,\eta-1} = H_t$ and $V_{t,\eta-1} = V_t$ is possible. $H_{t+1,\eta}$ has one more block column than $H_{t,\eta-1}$ and so the nullspace of $H_{t+1,\eta}$ has dimension at most $|V_{t,\eta-1}| + N$. It is sufficient then to show that $|V_{t+1,\eta}| = |V_{t,\eta-1}| + N$.

The definition of $V_{t,\eta-1}$ in (2.5) implies the following:

$$|V_{t,\eta-1}| = \sum_{\{\zeta | d_\zeta^{[t]} \leq \eta-1\}} (\eta - 1 - d_\zeta^{[t]} + 1).$$

For all $1 \leq k \leq N$, $d_k^{[t+1]} = d_{\varpi(k)}^{[t]} \leq \mu^{[t+1]} \leq \eta-1$ where $\varpi$ is an bijection from $\{1, 2, \ldots, 2N\}$ into $\{1, 2, \ldots, 2N\}$. So for each $k$, the range between $\eta$ and $d_k^{[t+1]}$ is one more than the range between $\eta - 1$ and $d_{\varpi(k)}^{[t]} = d_k^{[t+1]}$. Also for any auxiliary column $s$ with $d_s^{[t+1]} \leq \eta$, then the range between $d_{\varpi(s)}^{[t]}$ and $\eta - 1$ is equal to the range between $d_s^{[t+1]} = d_{\varpi(s)}^{[t]} + 1$ and $\eta$. Therefore we have the following equation:

$$|V_{t+1,\eta}| = \sum_{\{\zeta | d_\zeta^{[t+1]} \leq \eta\}} (\eta - d_\zeta^{[t+1]} + 1) = N + \sum_{\{\zeta | d_{\varpi(\zeta)}^{[t]} \leq \eta-1\}} (\eta - 1 - d_{\varpi(\zeta)}^{[t]} + 1) = |V_{t,\eta-1}| + N.$$

Thus $V_{t+1,\eta}$ exhausts the nullspace of $H_{t+1,\eta}$.

**Case 4:** $d < t + 1$, $\mu^{[t+1]} + 1 \geq \beta^{[t+1]}$ **and** $\mu^{[t+1]} > \mu^{[t]}$**.**

Then $d = \mu^{[t+1]} = d_l t$ for some $l$ with $N + 1 \leq l \leq 2N$. Let $\eta = d_l^{[t]}$. We know that $\eta > \max_{1 \leq j \leq N} \{d_j^{[t]}\}$. By the induction hypothesis, we have a right nullspace basis $V_{t,\eta}$ of the block Hankel matrix $H_{t,\eta}$, which is $H_{t+1}$ without the last block row. So we may use the same argument as in Case 3, replacing $H_t$ and $V_t$, with $H_{t,\eta}$ and $V_{t,\eta}$.

Finally, for all $\lambda = d_j^{[t+1]} > d = \eta$ the set $V_{t+1,\lambda}$ exhausts the nullspace of $H_{t+1,\lambda}$. Again we use the same argument in Case 3 by replacing $\mu^{[t+1]} = \mu^{[t]}$ with $\eta$ and $\eta$ with $\lambda$.

■

### 2.3.3 Correctness of the output

**Lemma 12** *For each iteration at step* MBM2 *we have the following. Consider* $\bar{M}_0 = M_0$, $\ldots$, $\bar{M}_t = M_t$, $\bar{F}(z) = z^\mu \left[ f_1(z^{-1}) \ldots f_N(z^{-1}) \right] \in K^{N \times N}[z]$ *and*

$$\bar{M}_{t+\theta} = -\left( \sum_{i=0}^{\mu-1} \bar{M}_{t+\theta-\mu+i} \, Coeff\,(i; \bar{F}) \right) \left[ f_1(0) \ldots f_N(0) \right]^{-1} \quad \text{for } \theta = 1, 2, \ldots.$$

*Then* $F(z) = \left[ z^{d_1} f_1(z^{-1}) \quad z^{d_2} f_2(z^{-1}) \quad \ldots \quad z^{d_N} f_N(z^{-1}) \right]$ *is the minimal matrix generator for* $\bar{M}_0, \bar{M}_1, \ldots$

**Proof.** Since $\left[f_1(0)\dots f_N(0)\right] = \mathit{Coeff}(\mu;\bar{F})$ the polynomial $\bar{F}$ is a matrix generator in the sense of Definition 3. In particular, $C^{[t+1]} = C^{[t+2]} = \cdots = 0^{N\times N}$ in step MBM4. Now let $n$ be the degree of the minimal generator, denoted by $G$. If $\mu+1 > \beta$ or $d < n$, then continue the algorithm until $d \geq n$ and $\beta \geq \mu+1$. Because the generator polynomials do not change any further on the sequence $\{\bar{M}_i\}$, $\min\{\beta-1, t\}$ will eventually catch up with both. Let $G_k$ be any column of $G$, let $e = \deg(G_k)$ and let $g(z) = z^e G_k(z^{-1})$. Because $g$ is a generator vector of $\{\bar{M}_i\}$, $\mathit{CoeffVec}(d; g(z))$ is in the nullspace of $H_t$ on page 22. Since $d \geq n$ and $\mu \leq \beta - 1$, by Theorem 4 we have

$$\mathit{CoeffVec}(d; z^{d-e}g(z)) = \sum_{j=1}^{N}\sum_{i=0}^{d-d_j} \alpha_{j,i}\,\mathit{CoeffVec}(d; z^i f_j(z)) \quad \text{for some } \alpha_{i,j} \in K.$$

Hence $G_k(z) = \sum_j \sum_i \alpha_{j,i} z^{d-d_j-i} F_j(z)$, where $F_j$ is the $j$-th column of $F$. Hence every column of $G$ is in the $K[z]$ submodule of $F$ and $F$ is a minimal generator. ■

**Lemma 13** *Suppose that $\{M_i\}$ is linearly generated. In Algorithm 2.2.1, there exists a $T$ such that for all $t \geq T$ the matrix polynomial*

$$F(z) = \left[z^{d_1} f_1(z^{-1}) \quad z^{d_2} f_2(z^{-1}) \quad \dots \quad z^{d_N} f_N(z^{-1})\right]$$

*is the minimal matrix generator for $\{M_i\}$.*

**Proof.** We introduce the notion of *defect* (cf. [3]) for each $1 \leq j \leq 2N$: $\mathrm{dfct}_j^{[t]} = t - d_j^{[t]}$. We now follow the defect of each vector polynomial $f_j$ throughout the algorithm. When a column is placed into the auxiliary part of $f$ and shifted in steps GE19 and MBM9 its defect remains unchanged when reaching step MBM5 again, because the nominal degree is incremented in step MBM6 and $t$ is incremented in step MBM3. The corresponding column that is switched into the generator part has its defect incremented by 1 due to the increment of $t$, as have all other generator vectors.

Now let $\nu$ be the degree of the first invariant factor of the minimal matrix generator, which by Theorem 2 is the scalar minimal generator. We claim that the defects of the generator columns eventually grow to $\nu$. The minimal defect $\min_{1\leq j\leq N}\{\mathrm{dfct}_j^{[t]}\}$ is $t - \mu^{[t]}$. When $\mu$ grows, in Case 1 or 4 in the proof of Theorem 4, the minimal defect can shrink. However, the corresponding column $f_l$ used in the update has its defect incremented. Since initially the defects are $-1$ and $-2$, after at most $N(\nu+2)$ cases where $\mu^{[t+1]} > \mu^{[t]}$ the defects of all generator columns must be at least $\nu$. In particular, then $\mu \leq t$.

Suppose now that at $t = T$ the defect of all generator polynomials is at least $\nu$. We complete the proof by showing that in Lemma 12 we have $\bar{M}_{T+\theta} = M_{T+\theta}$ for all $\theta \geq 1$. For $t = T$ in Theorem 4 we have $d \geq \mu$ and $T \geq \mu + \nu$. Furthermore, from (2.3) on page 22

$$M_{T-\kappa}\left[f_1(0)\dots f_N(0)\right] + \sum_{i=0}^{\mu-1} M_{T-\kappa-\mu+i}\,Coeff\,(i;\bar{F}) = 0^{N\times N} \quad \text{for } \kappa = 0, 1, \dots, \nu-1.$$

Let $z^\nu - \sum_{k=0}^{\nu-1} c_k z^{\nu-k}$ be the minimal scalar generator for $\{M_i\}$. Then

$$\begin{aligned} 0^{N\times N} &= \sum_{\kappa=0}^{\nu-1} c_\kappa M_{T-\kappa}\left[f_1(0)\dots f_N(0)\right] + \sum_{\kappa=0}^{\nu-1}\sum_{i=0}^{\mu-1} c_\kappa M_{T-\kappa-\mu+i}\,Coeff\,(i;\bar{F}) \\ &= M_{T+1}\left[f_1(0)\dots f_N(0)\right] + \sum_{i=0}^{\mu-1} M_{T+1-\mu+i}\,Coeff\,(i;\bar{F}). \end{aligned}$$

Therefore, $\bar{M}_{T+1} = M_{T+1}$. For larger $\theta$ we use induction. ■

**Lemma 14** *After each iteration of steps* MBM3 *through* MBM7 *of Algorithm 2.2.1,* $\beta^{[t+1]} + \sigma^{[t+1]} > \beta^{[t]} + \sigma^{[t]}$.

**Proof.** During each iteration of steps MBM3 through MBM7, $\sigma$ can never decrease so that $\sigma^{[t+1]} \geq \sigma^{[t]}$ for all $t \geq -1$.

If $\sigma^{[t+1]} = \sigma^{[t]}$, then steps GE20 and GE23 will not be performed by the call to Algorithm 2.2.2 in step MBM5. So $\beta^{[t+1]} = \beta^{[t]} + 1$, and thus $\beta^{[t+1]} + \sigma^{[t+1]} > \beta^{[t]} + \sigma^{[t]}$.

If $\sigma^{[t+1]} > \sigma^{[t]}$, then during step MBM5, step GE20 or step GE23 were performed at least once. Therefore $\beta^{[t+1]} = \beta^{[t]} + 1$, or $\beta^{[t+1]} = d_j^{[t]} + 1$ for some $1 \leq j \leq N$. In the former case, then $\beta^{[t+1]} + \sigma^{[t+1]} = \beta^{[t]} + 1 + \sigma^{[t+1]} > \beta^{[t]} + \sigma^{[t]}$. In the latter case, then $\beta^{[t+1]} + \sigma^{[t+1]} = d_j^{[t]} + 1 + \sigma^{[t+1]} \geq d_j^{[t]} + 1 + \sigma^{[t]} - d_j^{[t]} + \beta^{[t]} > \beta^{[t]} + \sigma^{[t]}$. ■

**Lemma 15** *Given any input matrix sequence* $\{M_k\}_{k=0}^{\infty}$ *and any bound* $\delta$*, Algorithm 2.2.1 returns "insufficient bound" or a candidate minimal generator* $F$ *after processing at most* $2\delta$ *elements.*

**Proof.** Suppose that Algorithm 2.2.1 is run until $t = 2\delta - 1$, thus processing the first $2\delta$. Since $\beta^{[-1]} + \sigma^{[-1]} = 1$, then Lemma 14 implies that $\beta^{[2\delta-1]} + \sigma^{[2\delta-1]} \geq 2\delta + 1$. If $\sigma \geq \delta + 1$ then "insufficient bound" is returned. If $\sigma \leq \delta$ then $\beta^{[2\delta-1]} + \sigma^{[2\delta-1]} \geq 2\delta + 1 \geq \delta + \mu^{[2\delta-1]} + 1$ since $\mu \leq \sigma$ by definition. Thus $\beta$ fails the condition of MBM2 and so Algorithm 2.2.1 will return a candidate minimal generator $F$. ■

**Theorem 5** *Suppose* $\{M_k\}_{k=0}^{\infty}$ *is linearly generated by a minimal matrix generator with minimal degree* $d$ *and determinantal degree* $\delta_M \leq \delta$. *Then Algorithm 2.2.1 returns a minimal generator of* $\{M_k\}_{k=0}^{\infty}$ *after processing at most* $d+\delta$ *elements. Otherwise,* $\{M_k\}_{k=0}^{\infty}$ *is not linearly generated or has minimal generator of determinantal degree* $\delta_M > \delta$. *In either case, Algorithm 2.2.1 returns "insufficient bound" or an incorrect generator after processing at most* $\min(\bar{d}, \delta) + \delta$ *elements where* $\bar{d}$ *is the degree of the generator candidate computed at* $t = 2\delta - 1$ *(see Lemma 12).*

**Proof.** We will show that Algorithm 2.2.1 returns a minimal matrix generator of a linearly generated sequence if the bound $\delta$ is large enough. We also give a tighter bound on the number of elements that need to be processed before terminating.

We begin by assuming the sequence is linearly generated and $\delta_M \leq \delta$. Since Lemma 13 states that Algorithm 2.2.1 will find a minimal generator, we can assume that $\mu \leq d$ for all $t$. We proceed as in Lemma 15 and assume the algorithm is run until $t = d+\delta-1$. Then $\beta+\sigma \geq d+\delta+1$ and so $\sigma > \delta$ or $\beta \geq \delta - \sigma + \mu + 1$. If $\sigma > \delta$ then step MBM8 returns "insufficient bound". If $\sigma \leq \delta$ then the condition of step MBM2 is violated. So the algorithm will terminate and return either "insufficient bound" or a candidate minimal generator.

If "insufficient bound" is returned, then $\sigma \geq \delta + 1$. Lemma 13 implies that there exists a $T$ such that if Algorithm 2.2.1 is computed to stage $T > t = d+\delta-1$ or beyond, then $F(z)$ is a minimal matrix generator. Since $\sigma^{[t]}$ is an increasing sequence, then $\sigma^{[T]} \geq \sigma > \delta$. If we can show that $\delta_M = \deg(\det(F(z))) = \sigma^{[T]} > \delta$, then we have proven $\delta_M > \delta$ and reached a contradiction. The upper bound $\deg(\det(F(z))) \leq \sigma^{[T]}$ is straightforward since Lemma 1 and Lemma 3 imply that $\deg(F_j(z)) = d_j$ for all $1 \leq j \leq N$. The multi-linear property of the determinant allows us to write $\det(F(z))$ in the following manner:

$$\det(F(z)) = z^{\sigma^{[T]}} \cdot \det(\begin{bmatrix} f_1(0) \dots f_N(0) \end{bmatrix}) + \text{ lower order terms.}$$

By Lemma 3, we know $\det(\begin{bmatrix} f_1(0) & \dots & f_N(0) \end{bmatrix}) \neq 0$ and so $\deg(\det(F(z))) = \sigma^{[T]}$. Therefore $\delta_M > \delta$ and so we have a contradiction.

If "insufficient bound" is not returned then we have $\delta \geq \sigma$, $F(z)$ a possible minimal generator and $\beta - \mu > \delta - \sigma$. Since $\delta - \sigma \geq 0$ then $\beta > \mu$. Suppose $F(z)$ is not the minimal generator of $\{M_k\}_{k=0}^{\infty}$. Lemma 13 proves that there is a $T \geq t = d+\delta-1$ such that Algorithm 2.2.1 returns a minimal generator. Since $F(z)$ is not a generator, then as the

algorithm is run until $T$, there must be a nonzero discrepancy. Since $\beta > \mu$, then $\sigma$ will be increased such that $\sigma^{[T]} \geq \sigma + \beta - \mu > \delta$. As in the previous case, this implies that $\delta_M > \delta$ and a contradiction has been found. Therefore $F(z)$ is a minimal generator of $\{M_k\}_{k=0}^{\infty}$.

Otherwise, assume $\{M_k\}_{k=0}^{\infty}$ is not linearly generated or is linearly generated with determinantal degree $\delta_M > \delta$. Suppose Algorithm 2.2.1 is run until $t = \min(\bar{d}, \delta) + \delta - 1$. So $\beta + \sigma \geq \min(\bar{d}, \delta) + \delta + 1$. If $\sigma > \delta$ then "insufficient bound" is returned. If $\sigma \leq \delta$ then we know that $\mu \leq \min(\bar{d}, \delta)$ by definition of $\mu$ and $\bar{d}$ and so $\beta \geq \delta - \sigma + \mu + 1$. So Algorithm 2.2.1 will return a possible generator $F$ with determinantal degree $\sigma$. Since $\{M_k\}_{k=0}^{\infty}$ is not linearly generated or is linearly generated with determinantal degree $\delta_M > \delta \geq \sigma$, then $F$ is an incorrect generator. ■

**Remark 1** Note that Theorem 5 is also applicable in the scalar case for the classical Berlekamp/Massey algorithm: if a degree bound $\delta$ for the linear generator is given, the Berlekamp/Massey algorithm can stop after processing $d + \delta$ sequence elements, where $d$ is the actual degree of the generator. That early termination property is useful, for example, in the Wiedemann algorithm for solving sparse linear systems over finite fields [47; 29].

Theorem 5 gives an exhaustive analysis of the way Algorithm 2.2.1 terminates for all possible input. If the the bound $\delta$ is sufficient, then a minimal generator is returned. If the bound $\delta$ is too small then the algorithm may return "insufficient bound" or an incorrect candidate generator. We ignore the possibility that the sequence is not linearly generated since only a finite number of sequence elements are processed and the sequence can always be extended to a linearly generated sequence (see Lemma 12). It is important to note that no algorithm, including Algorithm 2.2.1, can determine if the bound $\delta$ is sufficient. Such verification requires infinitely many sequence elements be processed, since the determinantal degree can increase at any stage. The correctness of the output of Algorithm 2.2.1 is dependent on $\delta$, and the algorithm must assume the bound $\delta$ is sufficient. If "insufficient bound" is returned, then the algorithm has proven that the bound $\delta$ is too small since the proof of Theorem 5 shows that the sequence can have no matrix generator of determinantal degree less than or equal to $\delta$. Otherwise Algorithm 2.2.1 returns a minimal generator of the completed sequence from Lemma 12 and assumes that the input sequence and the completed sequence are the same, which is true if the bound $\delta$ is sufficient. The output of Algorithm 2.2.1 is always correct for a given input $\delta$ that is assumed to be sufficient.

## 2.4 Matrix Berlekamp/Massey and Rectangular Matrix Sequences

Algorithms 2.2.1 and 2.2.2 and the subsequent proofs assume that the matrix sequence $\{M_k\}_{k=0}^{\infty}$ consists of square matrices. In this section we will detail the necessary changes that allow the algorithms to compute a minimal right generator of a rectangular matrix sequence. We revive our notation $N_{\text{row}}$ and $N_{\text{col}}$ to denote the row and column dimensions of the sequence. Rather than rewriting the algorithms, we only detail the differences between the square and rectangular algorithms. Finally, we give justification for our changes by appealing to the square algorithm and its output given a special input.

The input to Algorithm 2.2.1 will now be $M \in K^{N_{\text{row}} \times N_{\text{col}}}[[z]]$. The output of the algorithm will be $F(z) \in K^{N_{\text{col}} \times N_{\text{col}}}[z]$. The variable $f(z)$ is now an $N_{\text{col}} \times (N_{\text{col}} + N_{\text{row}})$ matrix. For the entire algorithm, the quantity $2N$ will be replaced by $N_{\text{col}} + N_{\text{row}}$. The matrix $\Delta$ will now have dimensions $N_{\text{row}} \times (N_{\text{col}} + N_{\text{row}})$. At Step MBM1, $f$ is initialized to be the matrix $\begin{bmatrix} I_{N_{\text{col}}} & 0^{N_{\text{col}} \times N_{\text{row}}} \end{bmatrix}$. The nominal degrees will be initialized thusly, $d_i = 0$ for all $i \leq N_{\text{col}}$ and $d_i = 1$ otherwise. In steps MBM4 and MBM5, $L^{[t]} \in K^{N_{\text{row}} \times N_{\text{row}}}$. Also $C^{[t]}$, $Z^{[t]} \in K^{N_{\text{row}} \times N_{\text{col}}}$. The diagonal block matrix in Step MBM9 will be changed to $\text{diag}(I_{N_{\text{col}}}, z \cdot I_{N_{\text{row}}})$. In steps MBM6, MBM7 and MBM10, $N$ is changed to $N_{\text{col}}$.

The changes that must be made to Algorithm 2.2.2 are as follows. As above $2N$ becomes $N_{\text{col}} + N_{\text{row}}$. In Step GE3, $N$ will be changed to $N_{\text{row}}$. Everywhere else, in Step GE2 and steps GE4 through GE24, $N$ is changed to $N_{\text{col}}$.

The specific changes given above are a result of the general changes that must occur when moving from square the rectangular matrix sequences. As in Section 2.1 the minimal right generator is a square matrix of dimension $N_{\text{col}}$. Thus the algorithm must reflect this dimension change. Because the sequence has row dimension $N_{\text{row}}$, then the algorithm must maintain $N_{\text{row}}$ auxiliary vectors. This reflects the possible rank of the columns of the matrix sequence. The number of generator columns and auxiliary columns affects the initialization and computation of $f$ and its associated variables. The differentiation between generator and auxiliary columns also forces the changes in Algorithm 2.2.2.

We now give justification for the changes listed above. Rather than redo the complete proof of the square algorithm, we will use the square algorithm to prove the rectangular algorithm. Given the rectangular matrix sequence $\{M_k\}_{k=0}^{\infty}$, we will construct a square matrix $\{\bar{M}_k\}_{k=0}^{\infty}$. The square sequence will have dimension $\max\{N_{\text{row}}, N_{\text{col}}\}$. If

$N_{\text{row}} > N_{\text{col}}$, then $\bar{M}_k = \left[ M_k \quad 0^{N_{\text{row}} \times (N_{\text{row}} - N_{\text{col}})} \right]$. Otherwise $\bar{M}_k$ is $M_k$ padded by $N_{\text{col}} - N_{\text{row}}$ zero rows. Now we consider the execution and output of Algorithm 2.2.1 when the sequence $\{\bar{M}_k\}_{k=0}^{\infty}$ is given as input. We assume that the input $\delta$ is the same for both sequences.

Suppose $N_{\text{col}} > N_{\text{row}}$. Then the square sequence contains the original sequence as full column submatrices. Thus any generator of $\{\bar{M}_k\}_{k=0}^{\infty}$ is a generator of $\{M_k\}_{k=0}^{\infty}$. The converse is also true since the added rows are zero rows. Thus the output of Algorithm 2.2.1 is a minimal generator of $\{M_k\}_{k=0}^{\infty}$, when $\{\bar{M}_k\}_{k=0}^{\infty}$ is given as input. However, the square algorithm contains too many auxiliary columns. At every stage $t$, the bottom $N_{\text{col}} - N_{\text{row}}$ rows of $\Delta^{[t]}$ are zero. Thus the last $N_{\text{col}} - N_{\text{row}}$ auxiliary columns will remain $0^{N_{\text{col}}}$ throughout the algorithm. It is therefore, unnecessary to compute those columns and those rows of $\Delta^{[t]}$. By changing the algorithms as listed above these computations are eliminated.

If $N_{\text{row}} > N_{\text{col}}$, then the output of Algorithm 2.2.1 given $\{\bar{M}_k\}_{k=0}^{\infty}$ has incorrect dimensions to be a minimal generator of $\{M_k\}_{k=0}^{\infty}$. Since the last $N_{\text{row}} - N_{\text{col}}$ columns of the matrix sequence are zero columns, then the last $N_{\text{row}} - N_{\text{col}}$ columns of $C^{[t]}$ are zero columns. This means that the column $f_i^{[t]}$ is unchanged since initialization and so $d_i^{[t]}$=0 for all $N_{\text{col}}+1 \leq i \leq N_{\text{row}}$. Therefore if $\bar{F}$ is the output of Algorithm 2.2.1 when given $\{\bar{M}_k\}_{k=0}^{\infty}$ as input, we know that $\bar{F} = \text{diag}(F, I_{N_{\text{row}} - N_{\text{col}}})$, with $F \in K^{N_{\text{col}} \times N_{\text{col}}}[z]$. $F$ is a minimal right generator of $\{M_k\}_{k=0}^{\infty}$ since the $\deg(\det(\bar{F})) = \deg(\det(F))$. So Algorithm 2.2.1 computes the minimal generator of $\{M_k\}_{k=0}^{\infty}$ when given $\{\bar{M}_k\}_{k=0}^{\infty}$ as input. The rectangular algorithm described above eliminates the computation of the last $N_{\text{row}} - N_{\text{col}}$ generator columns. These columns do not need to be computed since they are precomputed by the nature of the padded sequence.

The augmentations of Algorithm 2.2.1 and Algorithm 2.2.2 described above eliminate the unnecessary computations performed when a rectangular matrix sequence is converted to a square matrix sequence by padding the sequence with zeroes. The proof of the original algorithm in Section 2.3 implies that the rectangular algorithm is correct. This completely generalizes the Matrix Berlekamp/Massey algorithm and implies that given any linearly generated matrix sequence and a proper bound on the determinantal degree of a minimal generator, the Matrix Berlekamp/Massey algorithm will compute a minimal generator.

## 2.5 Arithmetic complexity

We end with a discussion of the complexity of Algorithm 2.2.1 and empirical evidence of its performance versus another method for computing minimal matrix generators. The results of Lemma 15 and Theorem 5 on page 28 give a bound on the number of elements Algorithm 2.2.1 processes before terminating. By amortizing the cost of each iteration, the bound will lead to a worst case complexity for the algorithm. Finally, we provide the results of a comparison between Algorithm 2.2.1 and a method based on the fast power Hermite padé solver of [3].

**Theorem 6** *Given any bound $\delta$ and any matrix sequence $\{M_k\}_{k=0}^{2\delta-1}$, Algorithm 2.2.1 has a worst case complexity of $O(\delta^2 N^2 + \delta N^3)$ field operations, where $N$ is the matrix dimension of the sequence.*

**Proof.** We will analyze steps MBM 4, MBM 5, and MBM 9 of Algorithm 2.2.1. These steps represent the major field operation costs of each iteration. In step MBM 4, we assume $L$ is computed by step MBM 5 of the previous iteration, and so only $C$ is computed. This assumption allows the cost of steps MBM 4 and MBM 9 to be given in terms of $\delta$. The results of Lemma 15 and Theorem 5 state that Algorithm 2.2.1 processes at most $2\delta$ elements of the sequence.

Since step MBM 5 is Gaussian elimination, it has a complexity of $O(N^3)$ field operations. We will amortize the costs of steps MBM 4 and MBM 9 by analyzing the computation cost column by column. Using matrix times vector products and matrix column operations instead of matrix multiplication will result in the stated complexity. Computing column $j$ of $C$ requires $d_j + 1$ matrix times vector products and $d_j$ vector additions for each $1 \leq j \leq N$. By summing over the range of $j$, then step MBM 4 has a cost of $\sigma N^2 + N^3 + \sigma N$ field operations. Since $\sigma \leq \delta$, then each iteration of step MBM 4 has a complexity of $O(\delta N^2 + N^3)$ field operations. To analyze step MBM 9, we use matrix column operations instead of matrix multiplication. The transformation matrix $\tau$ computed in step MBM 5 has two types of column operations. Each column operation is either a column switch or an addition of one polynomial vector and a second polynomial vector of a lesser degree times a scalar. Each generator column requires at most $N$ of the latter column operations and so $\tau$ has at most $N^2$ of these column operations. Each of these operations has a complexity of $O(d_j N + N)$ field operations where $1 \leq j \leq N$. So the total cost of computing column $j$ of $f$ has a

complexity of $O(d_j N^2 + N^2)$. The auxiliary columns will be computed as a result of the computations of the generator columns. By summing over $1 \leq j \leq N$, then the total cost of step MBM 9 has a complexity of $O(\sigma N^2 + N^3)$ field operations. During step MBM 5, $\sigma$ is updated. If $\sigma \leq \delta$, then the total cost of step MBM 9 has a complexity of $O(\delta N^2 + N^3)$ field operations. If $\sigma > \delta$, then step MBM 8 will return "insufficient bound" and step MBM 9 will not be performed.

Algorithm 2.2.1 processes each element of the sequence with worst case complexity of $O(\delta N^2 + N^3)$ field operations and it processes at most $2\delta$ elements. So Algorithm 2.2.1 has a worst case complexity of $O(\delta^2 N^2 + \delta N^3)$ field operations. ■

We now compare the performance of Algorithm 2.2.1 with an algorithm based on the fast power Hermite padé solver of [3]. In [42], Turner describes the relationship between the minimal matrix generator of a linearly generated matrix sequence and a $\sigma$-basis of an associated Padé system. Turner implemented his methods in Maple and provided a copy to the authors. We altered his original code to work over finite fields and added an operation count so the comparison could be performed. A Maple implementation of Algorithm 2.2.1 was used for comparison.

We test the algorithms using random square matrix sequences over GF2 with matrix dimensions varying from 1 to 10 and a fixed degree bound for each dimension. For each dimension, 10 random sequences are constructed and the minimal matrix generator of each sequence is computed. During each computation the number of arithmetic operations performed by each algorithm is calculated and an average for each dimension is determined. The sequences are constructed using two methods described below.

Experiment A For each dimension $i$, the sequences are constructed from bilinear block projections. A random $100 \times 100$ matrix $A$, and two random $100 \times i$ matrices $X$ and $Y$ are used to create the sequence $\{M_k\}_{k=0}^{\infty} = X^T A^k Y$. The degree bound for each $i$ is fixed at 100.

Experiment B For each dimension $i$, random matrix generators and initial sequences are used to construct each sequence. A matrix generator of degree $\lceil 100/i \rceil$ is constructed from random coefficients and a random initial sequence of length $\lceil 100/i \rceil$ is defined. Then using Definition 3 the sequence is completed. The degree bound for each dimension $i$ is $\lceil 100/i \rceil i \geq 100$.

Table 2.1: Arithmetic operation comparison

| Matrix Dimension | Experiment A | | Experiment B | |
|---|---|---|---|---|
| | Algorithm 2.2.1 | [42] | Algorithm 2.2.1 | [42] |
| 1 | 29946 | 99034 | 30720 | 100155 |
| 2 | 64665 | 380339 | 64133 | 384443 |
| 3 | 101948 | 868197 | 104877 | 907893 |
| 4 | 143652 | 1562988 | 140967 | 1557380 |
| 5 | 185175 | 2450470 | 180659 | 2456240 |
| 6 | 228962 | 3559226 | 234787 | 3696432 |
| 7 | 280705 | 4871117 | 301006 | 5350720 |
| 8 | 327771 | 6373395 | 343334 | 6888984 |
| 9 | 394147 | 8061927 | 423928 | 9409654 |
| 10 | 433936 | 10009119 | 429072 | 9992999 |

The results of the comparison given in Table 2.1 show that Algorithm 2.2.1 performs significantly faster than the $\sigma$-bases methods of [42]. For most dimensions the results for each experiment are similar. For some dimensions, such as dimension 9, the results of Experiment B are much larger than Experiment A. This is due to the larger degree bound used in Experiment B, which in the case of dimension 9 is 108 as compared to 100 for Experiment A.

# Chapter 3

# Fraction Free MBM

## 3.1 Introduction

There has been much work done on translating algorithms designed with field arithmetic into fraction free algorithms that work over integral domains. A well known algorithm is fraction free Gaussian elimination [1]. In [35; 13; 48], the authors describe extensions of the fraction free Gaussian elimination into other common linear algebra algorithms such LU decomposition, diagonalization, etc. Fraction free forms of the Euclidean algorithm for polynomials have been explored by [11; 7; 9; 8]. An important result in [9] is the Fundamental Theorem of Subresultants. A fraction free variant of the Berlekamp/Massey algorithm is given in [22]. The method is an application of the Fundamental Theorem of Subresultants to the Dornstetter transformation of the algorithm [15].

A fraction free version of the Matrix Berlekamp/Massey algorithm seems to be missing from the literature. A fraction free version of of the Matrix Berlekamp/Massey algorithm for square matrix sequences will be given. The algorithm requires that the matrix sequence be a *normalizable remainder sequence*. These normalizable remainder sequences are similar to the sequences in [30]. Normalizable here requires the non-singularity of the leading matrix coefficients in the remainders (discrepancies), not that the degrees of all quotient polynomials are 1 in the "normal polynomial remainder sequences" in the literature. All scalar sequences are normalizable in our sense. Our normality requirement is also related to the *normal points* used in to compute fraction free matrix padé systems in [2]. A fraction free matrix Berlekamp/Massey algorithm for arbitrary sequences has not been found. When the matrix algorithm is specialized to the scalar case, the resulting algorithm

is different than that of [22]. The latter algorithm relies on pseudo-division as prescribed by its use of subresultants. Our algorithm builds the pseudo-division step by step in the manner of [25] instead of using a large power multiplication. Thus the intermediate values of the algorithm are smaller or equal to those of the previous algorithm.

Section 3.2 describes the fraction free Matrix Berlekamp/Massey algorithm and the issues surrounding the algorithm. Section 2.3 gives a proof of integrality and a discussion of the minimality of the algorithm. Section 3.5 has two example scalar sequences and a theorem describing the unique minimal generators of scalar sequences.

## 3.2 Fraction Free Matrix Berlekamp/Massey Algorithm

We begin by describing the fraction free Matrix Berlekamp/Massey algorithm. The algorithm is given as a matrix algorithm but the scalar equivalents are given when they are simplified by the considering the case $N = 1$. A discussion of normalizable remainder sequences and their importance to Algorithm 3.2.1 follows.

### 3.2.1 Algorithm

**Input** $M(z) \in D^{N\times N}[[z]]$ the power series defined by the matrix sequence where $M(z) = \sum_{i=0} M_i z^i$ and $M_i \in D^{N\times N}$ with $D$ an integral domain.

$\delta$ an upper bound on the degree of the matrix generator.

Note that the algorithm will process at most $2 \cdot \delta$ elements of the sequence.

**Output** $F(z) \in D^{N\times N}[z]$ a minimal matrix generator with $\deg(F(z)) \leq \delta$.

In the case that the input sequence is not a normalizable remainder sequence, then the algorithm returns "singular sequence".

If the bound $\delta$ is too small then $F$ may be incorrect.

**Variables**

$t$ an index for the current sequence element being processed.

$\Lambda_t(z) \in D^{N\times N}[z]$ the reversal current generator.

$B_t(z) \in D^{N\times N}[z]$ the auxiliary polynomial used to eliminate discrepancies.

$L_t$ the nominal degree of $\Lambda_t(z)$. We have throughout the algorithm that $\deg(\Lambda_t) \leq L_t$.

$\Delta_t$, the coefficient of $z^t$ in the polynomial $M(z)\cdot\Lambda_{t-1}(z)$, often referred to in literature as the current discrepancy.

$\rho \in D$ the diagonal entry of the evaluation of $B_{t-1}(z)$. $\rho$ is initialized to 1.

$\epsilon$ a counter of the number of times the current $\rho$ has been used.

$\gamma$ the difference in the new and previous degree when $L_t$ is increased.

$g \in D$ and $h \in D$ scalar divisors used to control coefficient growth.

FFMBM1 $t \leftarrow 0$
$\Lambda_{-1}(z) \leftarrow I_N$
$B_{-1}(z) \leftarrow 0^{N\times N}$
$L_{-1} \leftarrow 0$
$\rho \leftarrow 1$
$\gamma \leftarrow -1$
$\epsilon \leftarrow -1$
$g \leftarrow h \leftarrow 1$.

FFMBM2 while $t + 1 - L_{t-1} \leq \delta$ do steps FFMBM3 through FFMBM12

FFMBM3 $\Delta_t \leftarrow \mathit{Coeff}\,(t; M(z) \cdot \Lambda_{t-1}(z))$

FFMBM4 if $\Delta_t \neq 0^{N\times N}$ and $2L_{t-1} < t + 1$ do step FFMBM5

FFMBM5 if $\det(\Delta_t) = 0$ then return "singular sequence"

We must compute $\mathrm{adj}(\Delta_t)$ and $\det(\Delta_t)$. Both can be computed using the fraction free diagonalization algorithm in [35].

$$\Lambda_t(z) \leftarrow \rho \cdot \Lambda_{t-1}(z) - B_{t-1}(z) \cdot \Delta_t$$

The multiplication on the left is scalar multiplication and the multiplication on the right is matrix multiplication. If $N = 1$ then both multiplications are scalar multiplications.

$$B_t(z) \leftarrow z \cdot \Lambda_{t-1}(z) \cdot \mathrm{adj}(\Delta_t)$$

Note that if $N = 1$, then $\mathrm{adj}(\Delta_t) = 1$.

$$\rho \leftarrow \det(\Delta_t)$$

$L_t \leftarrow t + 1 - L_{t-1}$

$\gamma \leftarrow t + 1 - 2L_{t-1}$

$\epsilon \leftarrow 0$

FFMBM6 if $\Delta \neq 0^{N \times N}$ and $2L_{t-1} \geq t + 1$ do step FFMBM7

FFMBM7 $\Lambda_t(z) \leftarrow \rho \cdot \Lambda_{t-1}(z) - B_{t-1}(z) \cdot \Delta_t$

$B_t(z) \leftarrow z \cdot B_{t-1}(z)$

$L_t \leftarrow L_{t-1}$

$\epsilon \leftarrow \epsilon + 1$

The use of $\epsilon$ is an implementation of the pseudo-division described in [25]. The algorithm skips over zeros in the pseudo-division and recovers the necessary multiplications in step FFMBM11. [8] describes a similar improvement of the PRS algorithm.

FFMBM8 if $\Delta_t = 0^{N \times N}$ do step FFMBM9

FFMBM9 $\Lambda_t(z) \leftarrow \Lambda_{t-1}(z)$

$B_t(z) \leftarrow z \cdot B_{t-1}(z)$

$L_t \leftarrow L_{t-1}$

FFMBM10 if $2L_t = t + 1$ do step FFMBM11

FFMBM11 $\Lambda_t(z) \leftarrow \frac{\rho^{\gamma-\epsilon}\Lambda_t(z)}{g \cdot h^{\gamma \cdot N}}$

$g \leftarrow \rho$

$h \leftarrow h^{1-\gamma \cdot N} \cdot g^{\gamma}$

FFMBM12 $t \leftarrow t + 1$

FFMBM13 return $F(z) = z^{L_{t-1}} \cdot \Lambda_{t-1}(z^{-1})$

### 3.2.2 Normalizable Remainder Sequences

The fraction free Matrix Berlekamp/Massey algorithm in section 3.2.1 will always return a candidate generator for scalar sequences. However, when the input sequence has a higher dimension, then the input sequences must be a normalizable remainder sequence.

Our concept of a normalizable remainder sequence is related to the results of [30]. There the authors use a half-gcd algorithm to compute a minimal matrix generator. The half-gcd algorithm requires that the leading term of each remainder must be nonsingular so the matrix polynomial division is defined. Further, they require that the degree difference between each successive remainder polynomial is 1, so there is no gap that can exist in a gcd calculation.

Algorithm 3.2.1 enforces a similar nonsingularity requirement but removes the gap restriction. During the execution of the Algorithm 3.2.1, the degree of the candidate minimal generator is updated if step FFMBM5 is performed. Whenever a nonzero discrepancy requires a degree increase, the discrepancy must be nonsingular. Such degree increases correspond to the division step of the half-gcd algorithm of [30]. The determininant of the discrepancy is calculated using the diagonalization algorithm of [35]. If the determinant is zero, then the algorithm returns an error statement "singular sequence". Since all scalar sequences are normalizable remainder sequences, then the scalar algorithm will not return "singular sequence" for any input. If the algorithm proceeded when $\det(\Delta_t) = 0$, then $\rho$ would be assigned the value 0. Thus eventually $g$ would be assigned 0 and a division by 0 could result. Therefore, the algorithm cannot allow a discrepancy that increases the degree of $\Lambda_t$ to be singular. The use of the gap variable $\gamma$ that represents the degree gap between successive generator degrees, allows us to remove the gap condition present in [30].

Such nonsingularity requirements are not uncommon in polynomial algebra. In [2], the authors compute matrix padé systems in a fraction free manner. Their algorithm proceeds on a diagonal path from *normal* point to *normal* point. These normal points are points in the padé table that obey a nonsingular requirement. These normal points are very similar to our normalizable remainder sequences, since both ideas imply that a well defined block matrix is nonsingular. In their case, a normal point is a point where a block Sylvester matrix is nonsingular. In our situation, a normalizable remainder sequence implies that the degree of each column of the generator is equal. Further as we shall see in Section 2.3, this implies that the $L_t \times L_t$, block Hankel matrix defined by the sequence is nonsingular for all $t$ such that $L_t > 0$. In [4], the authors generalize the definition of a normal point. They then are able to compute around singularities. Unfortunately we have not been able to apply those results and make a more general fraction free Matrix Berlekamp/Massey algorithm.

## 3.3 Proof of correctness

The proof of correctness for Algorithm 3.2.1 is given in two parts. First we show that the algorithm computes a minimal matrix generator if the bound $\delta$ is correct. If the input $\delta$ is two small, then as is the case in Algorithm 2.2.1, the candidate generator may be incorrect. Second we will show that all operations performed by Algorithm 3.2.1 are integer operations and all the variables remain integral.

### 3.3.1 Minimality of the output

In Section 2.3, the correctness of the general Matrix Berlekamp/Massey algorithm for field arithmetic, Algorithm 2.2.1, was given. We will restate some of the important theorems here and give different proofs for some of the theorems that are affected by the change in the update procedure of the new algorithm. We will also restate the importance of $\delta$ and the terminating condition, which were first given in Algorithm 2.2.1.

We begin with the following definition.

**Definition 8** *Define the quantity $b_{t-1}$ at every stage $t$ of Algorithm 3.2.1 by $b_{t-1} = t + 1 - L_{t-1}$. The quantity $b_{t-1}$ is the nominal degree of the polynomial $B_{t-1}(z)$. An alternate definition is $b_t = t + 2 - L_t$.*

As stated in the definition, $b_t$ is the nominal degree of the auxiliary polynomial $B_t$. The value of $b_t$ corresponds to the value $\beta$ in the general Matrix Berlekamp/Massey algorithm of Algorithm 2.2.1. Unlike, the general algorithm, we do not need to maintain the value of $b_t$ because it is a well defined formula involving $t$ and $L_t$. In the general algorithm, $\beta$ must be maintained because the nominal degree of the auxiliary (and generator) columns are not guaranteed to be equal. Algorithm 3.2.1 has this regularity since it only works on normalizable remainder sequence. In the general scalar algorithm, the main loop continues so long as $\beta \leq \delta$. So our algorithm maintains the same termination criteria. For if $\beta > \delta$, then $2L_{t-1} < t + 1$ and so any nonsingular discrepancy will increase the degree of the generator to $\beta > \delta$.

The definition of $b_t$ also allows us to state the next lemma. The following lemma is a restatement of Lemma 9. This lemma was first used by Coppersmith as one of the conditions his version of the Matrix Berlekamp/Massey algorithm maintained [12].

**Lemma 16** *At the completion of every stage $t$, the following holds.*

- *Coeff*$(l; M(z) \cdot \Lambda_t(z)) = 0^{N\times N}$ *for all* $l$ *such that* $L_t \le l \le t$.
- *Coeff*$(l; M(z) \cdot B_t(z)) = 0^{N\times N}$ *for all* $l$ *such that* $t + 2 - L_t = b_t \le l \le t$.

**Proof.** For $t = -1$, the lemma is true by default since both ranges are empty.

Suppose $t \ge -1$ and the lemma holds at $t$. At stage $t+1$, there are three possible updates depending on the $L_t$ and $\Delta_{t+1}$. We will analyze all three updating procedures to prove the lemma.

First suppose that $2L_t < t + 2$ and $\Delta_{t+1} = 0^{N\times N}$. So $\Lambda_{t+1}(z) = \Lambda_t(z)$, $L_{t+1} = L_t$ and $B_{t+1}(z) = z \cdot B_t(z)$. So by induction *Coeff*$(l; M(z) \cdot \Lambda_{t+1}(z)) = 0^{N\times N}$ for all $L_{t+1} \le l \le t$. Further, since $\Delta_{t+1} =$ *Coeff*$(t + 1; M(z) \cdot \Lambda_{t+1}(z)) = 0^{N\times N}$, then the condition holds for $\Lambda_{t+1}(z)$. Since $B_{t+1}(z) = z \cdot B_t(z)$, then by induction we know for all $l$ such that $t+3-L_{t+1} \le l \le t+1$ we see that *Coeff*$(l; M(z)B_{t+1}(z)) =$ *Coeff*$(l-1; M(z)B_t(z)) = 0^{N\times N}$ with $t + 2 - L_{t+1} \le l - 1 \le t$. So the condition holds.

Next suppose that $2L_t < t + 2$ and $\Delta_{t+1} \ne 0^{N\times N}$ is nonsingular. So $B_{t+1}(z) = z\cdot\Lambda_t(z)\cdot\text{adj}(\Delta_{t+1})$ and $L_{t+1} = t+2-L_t$. thus by induction, for all $l$ such that $t+3-L_{t+1} = L_t + 1 \le l \le t + 1$, we know that *Coeff*$(l; M(z)B_{t+1}(z)) =$ *Coeff*$(l - 1; M(z)\Lambda_t(z)) \cdot \text{adj}(\Delta_{t+1}) = 0^{N\times N} \cdot \text{adj}(\Delta_{t+1}) = 0^{N\times N}$ with $L_t \le l - 1 \le t$. So the condition holds for $B_{t+1}(z)$. Note that *Coeff*$(t + 2; M(z)B_{t+1}(z)) = \rho \cdot I_N$. If $L_t = 0$, then $L_{t+1} = t + 2$ and so the range is empty and the condition holds for $\Lambda_{t+1}(z)$ by default. Otherwise, since $\Lambda_{t+1}(z) = \rho \cdot \Lambda_t(z) \cdot -B_t(z) \cdot \Delta_{t+1}$, then by induction, for all $l$ such that $t+2-L_t = L_{t+1} \le l \le t$ we know that *Coeff*$(l; M(z)\Lambda_{t+1}(z)) = \rho \cdot 0^{N\times N} - 0^{N\times N} \cdot \Delta_{t+1} = 0^{N\times N}$. Further since $L_t > 0$, then $B_t(z)$ is nonzero and as above *Coeff*$(t+1; M(z)B_t(z)) = \rho \cdot I_N$. So we see that *Coeff*$(t + 1; M(z)\Lambda_{t+1}(z)) = \rho \cdot \Delta_{t+1} - (\rho \cdot I_N) \cdot \Delta_{t+1} = 0^{N\times N}$. Thus the condition holds for $\Lambda_{t+1}(z)$.

Finally, suppose $2L_t \ge t + 2$. Its obvious that if we prove the condition after step 7, then the scalar adjustment of step 11 will not affect the condition. In step 7, we see that $\Lambda_{t+1}(z) = \rho \cdot \Lambda_t(z) - B_t(z) \cdot \Delta_{t+1}$ and $L_{t+1} = L_t$. So as in the previous case, induction implies that for all $l$ such that $L_t = L_{t+1} \le l \le t$ we know that *Coeff*$(l; M(z)\Lambda_{t+1}(z)) = \rho \cdot 0^{N\times N} - 0^{N\times N} \cdot \Delta t + 1 = 0^{N\times N}$. Further, as in the previous case, we see that *Coeff*$(t + 1; M(z)\Lambda_{t+1}(z)) = \rho \cdot \Delta_{t+1} - (\rho \cdot I_N) \cdot \Delta_{t+1} = 0^{N\times N}$. So the condition holds for $\Lambda_{t+1}(z)$. Like the first case, we have $B_{t+1}(z) = z \cdot B_t(z)$. So the proof of the condition is the same and the condition holds for $B_{t+1}(z)$.

Thus the conditions hold at stage $t+1$ for every possible update, and so by induction the conditions hold at every stage. ■

Since Lemma 16 holds for our fraction free Matrix Berlekamp/Massey algorithm, then the subsequent lemmas from Section 2.3 also hold for the algorithm. Thus, Theorem 4, Lemma 12, Lemma 13 and Theorem 5 imply that Algorithm 3.2.1 has the following output options. The algorithm returns a minimal matrix generator for a prefix of the matrix sequence $\{M_k\}_{k=0}^{\infty}$. This generator is valid for the given $\delta$. If the generator does not generate the sequence past the processed prefix, then the given $\delta$ was too small for the sequence and the minimal generator of the sequence has degree larger than $\delta$. Finally, if the algorithm returns "singular sequence" then this is a certificate that the sequence is not a normalizable remainder sequence. Theorem 5 of also states that the fraction free Matrix Berlekamp/Massey algorithm will process at most a prefix of $2\delta$ elements of the sequence.

Both Algorithm 2.2.1 and Algorithm 3.2.1 have error outputs, but the error outputs of each algorithm are not the same. As previously stated, if Algorithm 3.2.1 return "singular sequence", then the algorithm has diagnosed that the sequence is not a normalizable remainder sequence. This condition is not a requirement of the general matrix Berlekamp/Massey algorithm since that algorithm computes a minimal matrix generator for arbitrary matrix sequences. Therefore the general algorithm does not produce the "singular sequence" output. The general algorithm's error output is "insufficient bound". Algorithm 3.2.1 does not produce this error because the $\delta$ of the algorithm is different than the $\delta$ of the general algorithm, which we will denote as $\bar{\delta}$. The two values are related by the formula $\bar{\delta} = N\delta$. In the general algorithm, the terminating value $\bar{\delta}$ is a condition not on the degree of the generator but the determinantal degree. Further, during an update, the determinantal degree could increase past $\bar{\delta}$ while the loop condition had been satisfied. This is not possible in Algorithm 3.2.1 since every column of the generator has the same nominal degree, meaning the determinantal degree will always be $N \cdot L_t$ at every stage $t$. The enforcement of the sequence being normalizable, means that Algorithm 3.2.1 would return "insufficient bound" only if $N \cdot L_t > N \cdot \delta$, which implies that $L_t > \delta$. Assuming that the degree was increased at stage $t$, then this implies that $L_t = t+1-L_{t-1} > \delta$. This condition violates the loop condition and so the algorithm would have ended before the update. Thus Algorithm 3.2.1 cannot diagnose if $\delta$ is too small, but will instead compute a generator that is valid for the given $\delta$ and a certain prefix of the matrix sequence.

### 3.3.2 Integrality

Having established that Algorithm 3.2.1 computes a minimal matrix generator, we are left to show that every intermediate value remains integral. To do this, we will show that $h$ and $\Lambda_{t-1}(0)$ are determinants of integral matrices, generalizing the subresultant based identities of [22] to matrix polynomials by giving a direct argument. Being determinants obviously implies integrality of those values, but by applying Cramer's rule, this implies the integrality of every intermediate value.

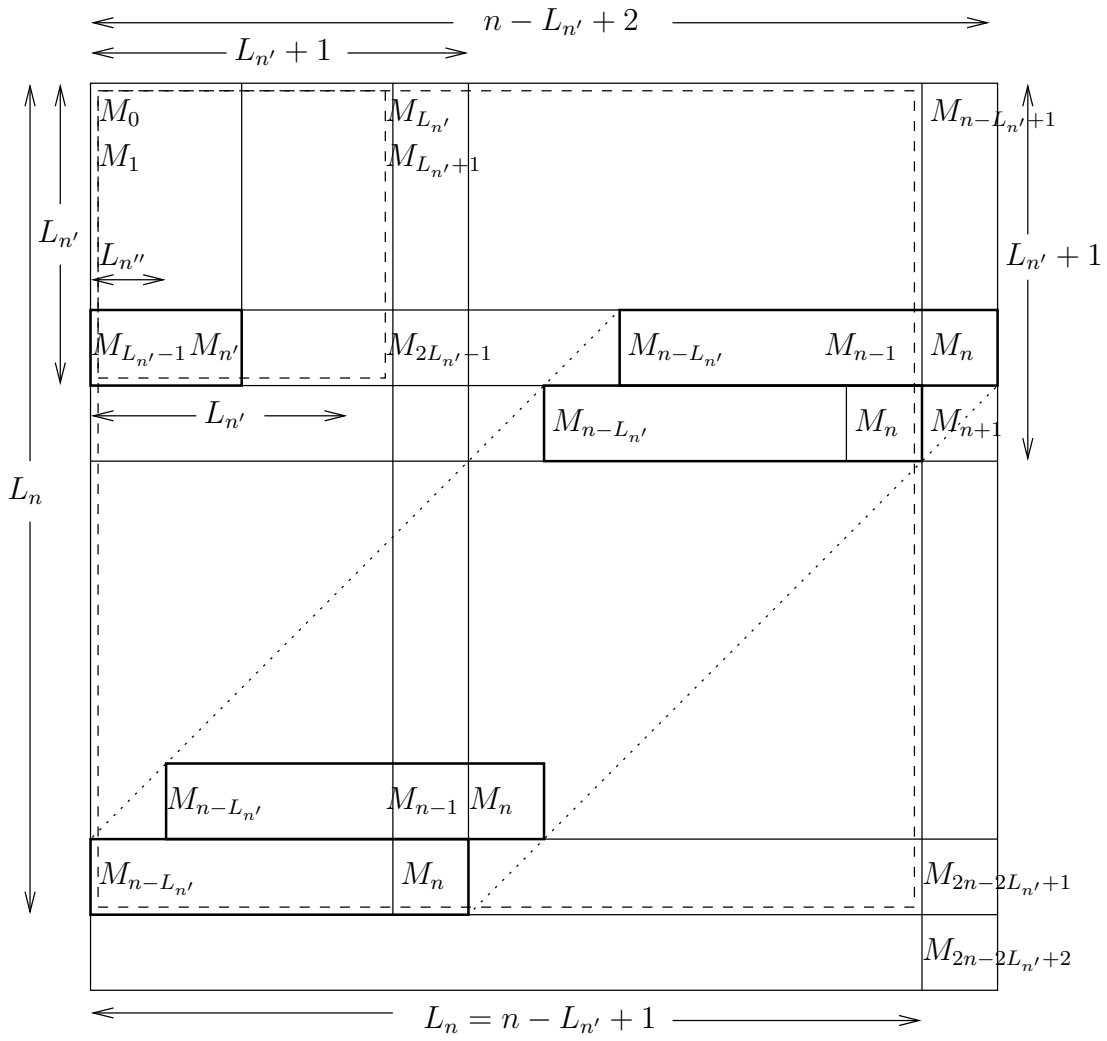


Figure 3.1: Berlekamp/Massey algorithm

Figure 3.1 follows the execution as in [28,Fig. 1]. For $t = n$ a non-zero discrepancy

$\Delta_n$ has caused execution of Step FFMBM5. The generator degree has changed last at $t = n'$, i.e., $2L_{n-1} = 2L_{n'} \leq n$. So $L_n$ is increased to $L_n = n + 1 - L_{n'} > L_{n'}$, and subsequent iterations execute either Step FFMBM7 or Step FFMBM8 until $t = 2L_n - 1 = 2n - 2L_{n'} + 1 > n$ without changing $L_t$, i.e., "complete the square." Additionally, at $t = 2L_n - 1$ the divisions of Step FFMBM11 adjust the generator polynomial and the auxiliary scalars.

**Lemma 17** *In Algorithm 3.2.1 at the end of Step* FFMBM*11, before Step* FFMBM*12 that increments $t$, the following statements are always true.*

1. *The scalar $h = (-1)^{N \cdot \chi}$ times the determinant of the the principal $(NL_t) \times (NL_t)$ block Hankel matrix $H_{L_t}$ generated by $\{M_k\}_{k=0}^{\infty}$, where $\chi = \sum_{\gamma'} \lfloor \gamma'/2 \rfloor$ with the summation taken over each value of $\gamma'$ that has been computed in Step* FFMBM*5 so far. In addition, that determinant is non-zero.*

2. *$\Lambda_t(0) = h\, I_N$.*

**Proof.** We will use induction to prove the lemma. Subscripted quantities $\rho_i$, $\gamma_i$, $\epsilon_i$, etc. refer, like $L_i$, to the values these variables have at the end of the iteration for $t = i$ at Step FFMBM12 before $t$ is incremented. If the sequence is all zero matrices, Step FFMBM11 is never executed and the trivial identity matrix generator is returned. Thus the lemma is true.

Assume now that the algorithm encounters its first non-zero $M_k$, where $k \geq 0$. Then $t = k$, $\Delta_k = M_k$ and Step FFMBM5 is executed. If $M_k$ is non-singular, the algorithm continues. In Figure 3.1 we then have $L_{n'} = 0$, $n = k$, and $L_{2n+1} = L_n = n + 1$, and $\gamma_n = L_n - L_{n'} = n + 1$. Step FFMBM11 occurs at $t = 2n + 1$ and the $(N(n+1)) \times (N(n+1))$ block Hankel matrix has the form

$$H_{L_{2n+1}} = \begin{bmatrix} 0^{N\times N} & 0^{N\times N} & \cdots & 0^{N\times N} & M_k \\ 0^{N\times N} & 0^{N\times N} & \iddots & M_k & \\ \vdots & \iddots & \iddots & & \\ 0^{N\times N} & M_k & & \ast & \\ M_k & & & & \end{bmatrix},$$

whose determinant is $(-1)^{N\lfloor (n+1)/2 \rfloor} \rho_n^{n+1} = (-1)^{N \cdot \chi_{2n+1}} h_{2n+1}$, because $\lfloor (n+1)/2 \rfloor$ block row exchanges put the matrix in block upper triangular form, where each block row exchange

needs $N$ row exchanges. Since $B_t(z)$ is always a multiple of $z$, $\Lambda(0)$ has been multiplied by $\rho_n$ in Step FFMBM7 exactly $\epsilon_{2n+1}$ times, and subsequently in Step FFMBM11 an additional $\gamma_n - \epsilon_{2n+1}$ times, yielding $\Lambda_{2n+1}(0) = \rho_n^{\gamma} I_N = h_{2n+1} I_N$.

Assume that the lemma is true for $t = j' = 2L_{n'} - 1$, the last time the end of Step FFMBM11 was reached (see again Figure 3.1). We suppose that the first next non-zero, non-singular discrepancy is found at iteration $t = n > 2L_{n'} - 1$. Then the square is completed at iteration $t = j = 2n - 2L_{n'} + 1 > n$. We subscript the variable $h$ by $j'$ and $j$, depending for which $t$ we have completed Step FFMBM11. Let $\Lambda_{j'}(z) = \sum_{\kappa=0}^{L_{n'}} \lambda_{j',\kappa} z^{\kappa}$, where $\lambda_{j',\kappa} \in D^{N \times N}$. When post-multiplying the $(NL_n) \times (NL_n)$ block Hankel matrix $H_{L_n}$ (see Figure 3.1) by the $(L_n N) \times (L_n N)$ block upper triangular matrix

$$U_{L_n} = \begin{bmatrix} I_N & 0 & 0 & \dots & 0 & \lambda_{j',L_{n'}} & 0 & \dots \\ 0 & I_N & 0 & \dots & 0 & \lambda_{j',L_{n'}-1} & \lambda_{j',L_{n'}} & \dots \\ \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \ddots & I_N & 0 & \lambda_{j',2} & \lambda_{j',3} & \dots \\ 0 & 0 & 0 & \ddots & I_N & \lambda_{j',1} & \lambda_{j',2} & \dots \\ 0 & 0 & 0 & \ddots & 0 & \lambda_{j',0} & \lambda_{j',1} & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & \lambda_{j',0} \end{bmatrix}$$

(there are $\gamma_n = L_n - L_{n'}$ block columns containing matrix coefficients $\lambda_{j',\kappa}$), we get

$$\begin{bmatrix} H_{L_{n'}} & 0 & 0 & 0 \\ 0 & 0 & 0 & \Delta_n \\ 0 & \vdots & \iddots & \\ 0 & \Delta_n & & \ast \end{bmatrix}.$$

As in the induction step, this matrix requires $\lfloor \gamma_n/2 \rfloor$ block row exchanges for the matrix to be transformed to block upper triangular form, where each block row exchange requires $N$ row exchanges. By hypothesis $\det(\lambda_{j',0}) = h_{j'}^N$. Therefore

$$\det(H_{L_n}) = \frac{\det(H_{L_{n'}})\ (-1)^{N\,\lfloor \gamma_n/2 \rfloor}\ \det(\Delta_n)^{\gamma_n}}{\det(U_{L_n})} = (-1)^{N\,\lfloor \gamma_n/2 \rfloor} \frac{(-1)^{N \cdot \chi_{j'}} h_{j'} \rho_n^{\gamma_n}}{h_{j'}^{\gamma_n N}} = (-1)^{N \cdot \chi_j} h_j,$$

which establishes Part 1.

For Part 2, by hypothesis $\Lambda_{j'}(0) = h_{j'} I_N$ and, similarly to the induction basis above, that coefficient gets multiplied by $\rho_{n'}$ once in Step FFMBM5 and by $\rho_n$ exactly $\epsilon$ times in Step FFMBM7 before Step FFMBM11. Therefore

$$\Lambda_j(0) = \frac{\rho_n^{\gamma_n - \epsilon_j} \rho_n^{\epsilon_j} \rho_{n'}}{\rho_{n'} h_{j'}^{\gamma_n N}} \Lambda_{j'}(0) = \frac{\rho_n^{\gamma_n}}{h_{j'}^{\gamma_n N}} h_{j'} I_n = h_j I_n.$$

■

We now show that every intermediate value is integral at every stage of Algorithm 3.2.1.

**Theorem 7** *The quantities $\Lambda_t(z)$, $B_t(z)$, $\rho$, $\Delta_t$, $g$ and $h$ of Algorithm 3.2.1 are integral for all $t \geq 0$.*

**Proof.** We continue the notation from Lemma 17. We will again use induction on $t$ to prove the theorem. We only need to worry about the computations at $t = 2n - 1$, where $n$ is a stage that step FFMBM5 is performed, since this is the time that step FFMBM11 is performed. All eliminations in steps FFMBM5 and FFMBM7 contain no divisions. So if the variables are integral at the completion of stage $2n-1$, then at every other step, the values must remain integral. Further this implies that $\Delta_t$ and $\mathrm{adj}(\Delta_t)$ are always integral if $\Lambda_t(z)$ is always integral. Thus $g$ and $\rho$ are always integral since they are $\det(\Delta_t)$ for some $t$ at every stage. If the sequence is all zero matrices then no update is ever performed, so the theorem is true by initialization.

Let $M_k$ be the first nonzero discrepancy where $k \geq 0$. At $t = 2k - 1$, Step 11 is performed. Since $g_t = h_t = 1$, then there is no division and so all of the values are integral.

Assume that the theorem is true for $t = j' = 2L_{n'} - 1$, the last time the end of Step FFMBM11 was reached (see again Figure 3.1). We suppose that the first next non-zero, non-singular discrepancy is found at iteration $t = n > 2L_{n'} - 1$. Then the square is completed at iteration $t = j = 2n - 2L_{n'} + 1 > n$. Since $B_j(z) = z^{\gamma_t + 1} \cdot \Lambda_{n-1}(z) \cdot \mathrm{adj}(\Delta_n)$, then the induction hypothesis proves that $B_j(z)$ is integral. Before the division in step FFMBM11, we know by induction that $\Lambda$ is integral. Lemma 17 says that $h_j$ is integral since it only differs from the determinant of the $L_{j+1} \times L_{j+1}$ block Hankel matrix $H_j$ defined in Lemma 17 by a sign. Also $h_j$ is nonzero since $h_j$ and $\rho_j$ are nonzero. Let $\hat{H}$ be the $L_j \times L_j + 1$ rectangular block Hankel matrix defined by $\{M_k\}_{k=0}^{\infty}$ (see Figure 3.1). Lemma 9 implies that the $L_j + 1$ block coefficient vector $\lambda^k$ , defined by the $k$th column of $\Lambda_j$ is a nullspace

vector of $\hat{H}$. Since $\Lambda(0) = \pm\det(H_j) \cdot I_N$, then the last $N$ rows of $\lambda^k = \pm\det(H_j) \cdot e^k$ where $e^k$ is the $k$th standard basis vector. Thus the first $L_j \cdot N$ entries of $\lambda^k$ are equal to $\pm\det(H_j) \cdot H_j^{-1} \cdot b^k$ where $b^k$ is the $k$th column of the $L_j + 1$ block column of $\hat{H}$ (see Figure 3.1). Thus Cramer's rule implies that $\lambda^k$ is integral for $1 \leq k \leq N$. So all of the coefficients of $\Lambda_j(z)$ are integral at the completion of stage $j$. Therefore the theorem holds at every stage $t$ of Algorithm 3.2.1. ■

## 3.4 Block Hankel Systems With Arbitrary Right Sides

Figure 3.2 follows the execution of the Matrix Berlekamp/Massey algorithm as a block Hankel matrix solver with arbitrary right side $Y$. If we assume that the finite sequence of matrices defining the block Hankel matrix is a normalizable remainder sequence, then the Matrix Berlekamp/Massey algorithm can be extended to a linear solver with the addition of one block (solution) vector. The solution block column vector $\widetilde{\Lambda}$ is produced for each non-singular leading principal block Hankel coefficient matrix. When Algorithm 3.2.1 increases the generator degree in Step 5, then the next blocks $Y_{L_{n'}}, \ldots, Y_{L_n-1}$ can be solved for like the corresponding $M$ blocks. In Figure 3.2, the columns containing $Y$ are not actually a part of the block Hankel matrix but are there to illustrate when each block of $Y$ is processed. Note the $\widetilde{\Lambda}_n$ is $\widetilde{\Lambda}_{L_{n'}-1}$ padded at the bottom by $L_n - L_{n'}$ zero blocks. A block discrepancy $\widetilde{\Delta}_t$ at $Y_t$, $L'_n \leq t \leq L_n - 1$, of the current solution $\widetilde{\Lambda}_{t-1}$ for $Y_0, \ldots, Y_{t-1}$ is removed by $\widetilde{\Lambda}_t \leftarrow \rho \cdot \widetilde{\Lambda}_{t-1} - \text{adj. coeff. col. vector}(B_{t-1}) \cdot \widetilde{\Delta}_t$. The fraction free division at Step 11 is the same, with a separate count for zero discrepancies.

## 3.5 Examples and Conclusions

### 3.5.1 Scalar Example

We give two simple scalar examples to illustrate how the improved pseudo-division implementation in our algorithm results in smaller intermediate values than that of the fraction free Berlekamp/Massey algorithm of [22]. The first sequence shows how our algorithm builds the pseudo-division step-by-step and how that affects the coefficient size. Let the sequence $\{a_k\}_{k=0}^{\infty}$ be defined as:

$$0, 0, 0, 5, 5, 10, 15, 25, 40, 65, 105, 172, 275, 445, 720, 1165, \ldots,$$

a variation of the Fibonacci sequence. This Fibonacci sequence has an error at $a_{11}$. This error will be corrected if we make $\delta$ large enough. By inputting $\delta = 14$, Algorithm 3.2.1 will correct the error by ignoring the first 11 sequence elements. The following table gives the value of $\Lambda_t(0)$ for both algorithms.

Table 3.1: Example 1

| Iteration | GKL | Our Algorithm | Iteration | GKL | Our Algorithm |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 15 | ≈–7.6e13 | ≈6.1e12 |
| 2 | 1 | 1 | 16 | 10000 | 10000 |
| 3 | 1 | 1 | 17 | ≈9.5e12 | ≈1.25e7 |
| 4 | –625 | 1 | 18 | 760000 | –760000 |
| 5 | –625 | 5 | 19 | ≈2.3e18 | ≈5.7e11 |
| 6 | –625 | 25 | 20 | 4040000 | –4040000 |
| 7 | –625 | 125 | 21 | ≈–4.2e20 | ≈1.6e13 |
| 8 | 625 | 625 | 22 | –25881600 | –25881600 |
| 9 | 625 | 625 | 23 | ≈1.1e23 | ≈6.6e14 |
| 10 | 625 | 625 | 24 | 166722816 | 166722816 |
| 11 | 625 | 625 | 25 | ≈–1.1e25 | ≈2.7e16 |
| 12 | ≈–7.6e13 | 3125 | 26 | –430074880 | 430074880 |
| 13 | ≈–7.6e13 | 3906250 | 27 | ≈1.8e22 | ≈1.8e17 |
| 14 | ≈–7.6e13 | ≈4.8e9 | 28 | 102400 | –102400 |

Notice that the $|\Lambda_t(0)|$ of Algorithm 3.2.1 is smaller or equal to the corresponding value of [22]. The iterations where the two algorithms have the same value are the iterations where the Step 11 is executed or iterations where a zero discrepancy occurs. In these steps, the value of $|\Lambda_t(0)|$ is a known determinant and so the values must be equal. In every other step, when the pseudo-division of the algorithm in [22] is being performed, our algorithm is smaller. This difference is also true for the other coefficients of $\Lambda_t$. The polynomial computed by our algorithm is $-102400z^{14} + 102400z^{13} + 102400z^{12}$. The algorithm of [22] computes the negative polynomial of our algorithm.

The second example shows how the improved pseudo-division of [25] allows us to skip over zero discrepancies during the pseudo-division in our algorithm. We define the sequence $\{a_k\}_{k=0}^{\infty}$ to be defined as:

$$0, 0, 0, 0, 0, 5, 0, 0, 0, 0, 5, 0, 0, 0, 0, 10, 0, 0, 0, 0, 15, \ldots,$$

another variation of the Fibonacci sequence. This variation of the Fibonacci sequence is the standard sequence, multiplied by 5 with four zeroes inserted between each Fibonacci

sequence element. These zeroes lead to zero discrepancies during the pseudo-division of both our algorithm and the algorithm of [22]. The following table gives the value of $\Lambda_t(0)$ for both algorithms.

Table 3.2: Example 2

| Iteration | GKL | Our Algorithm | Iteration | GKL | Our Algorithm |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 11 | –15625 | 5 |
| 2 | 1 | 1 | 12 | 15625 | 15625 |
| 3 | 1 | 1 | 13 | 15625 | 15625 |
| 4 | 1 | 1 | 14 | 15625 | 15625 |
| 5 | 1 | 1 | 15 | 15625 | 15625 |
| 6 | –15625 | 1 | 16 | ≈–2.9e25 | 78125 |
| 7 | –15625 | 1 | 17 | ≈–2.9e25 | 78125 |
| 8 | –15625 | 1 | 18 | ≈–2.9e25 | 78125 |
| 9 | –15625 | 1 | 19 | ≈–2.9e25 | 78125 |
| 10 | –15625 | 1 | 20 | 9765625 | 9765625 |

As the table shows, our algorithm builds the pseudo-division step by step and and skips over zero discrepancies during the pseudo-division steps. This makes the intermediate values much smaller. As in the previous case, the two algorithms have the same coefficients during the iterations where $\Lambda_t(0)$ is a known determinant. The polynomial computed by both algorithms for this sequence is $9765625z^{10} - 9765625z^5 - 9765625$.

### 3.5.2 Scalar Monic Minimal Generators

During the design of the scalar version of Algorithm 3.2.1, the following theorem was discovered. As the algorithm was tested on various sequences, a pattern emerged. For every integer sequence, the minimal generator $F$, computed by Algorithm 3.2.1 had a very intriguing content. Every coefficient of the generator was divisible by the leading term of $F$. Therefore, by dividing $F$ by its leading term, we will call it $h$, then $F/h$ is monic and so it must be the unique minimal generator of the sequence denoted by $f_{\min}$. Thus since $h$ exactly divides every coefficient of $F$, then the unique minimal generator is integral. The two examples in the previous section both exhibit this behavior. So we have the following theorem, a generalization of a result by Pierre Fatou in [20,page 368].

**Theorem 8** *If $\{a_k\}_{k=0}^{\infty}$ is a linearly generated sequence with $a_i \in D$ where $D$ is a unique factorization domain, then the monic minimal generator $f_{\min} \in D[z]$.*

**Proof.** Let $F$ be the field of quotients of $D$ and let $a(z) \in D[[z]]$ be the power series $a(z) = \sum_{i \geq 0} a_i z^i$. Without loss of generality, we assume that the unique monic minimal generator of $\{a_k\}_{k=0}^{\infty}$, $f_{\min}$, is such that $f_{\min}(0) \neq 0$. Otherwise, $f_{\min}$ is divisible by $z^i$ for some $i > 0$. This factor of $z^i$ skips over the first $i$ sequence elements in the linear generation. As such we can divide $f_{\min}$ by $z^i$ and skip the necessary sequence elements to continue the proof.

Since $\{a_k\}_{k=0}^{\infty}$ is linearly generated, then there exists two polynomials $A(z), B(z) \in F[z]$ such that $\frac{A(z)}{B(z)} = a(z)$ and $A(z), B(z)$ are relatively prime. This is a known result of linearly generated (linearly recurrent) sequences and rational functions. Further, we know that $B(z)$ is a minimal recurrence of $\{a_k\}_{k=0}^{\infty}$ and so the reverse polynomial of $B(z)$ is a minimal generator of $\{a_k\}_{k=0}^{\infty}$. By clearing out denominators, we can assume that $A(z), B(z) \in D[z]$ and $A, B$ remain relatively prime. So $A(z) = B(z)a(z)$. We now proceed to show that $B$ is the unique minimal recurrence.

Suppose there exists $d \in D$, $d$ not a unit, such that $d$ is a common divisor of the coefficients of $B(z)$. Then since $A(z) = B(z)a(z)$, we see that $d$ is a common divisor of all the coefficients of $A(z)$, contradicting the relative primeness of $A$ and $B$. So no such $d$ can exist.

Since $A$ and $B$ are relatively prime, there exists $U(z), V(z) \in D[z]$ such that $UA + VB = \alpha \in D \setminus \{0\}$. Thus the rational function $G = \frac{\alpha}{B(z)} = U(z)\frac{A(z)}{B(z)} + V(z) = U(z)a(z) + V(z) \in D[[z]]$. If $c \in D \setminus \{0\}$ is a common divisor of the coefficients of $G$, then $B(z)G(z) = \alpha$ implies that $c \mid \alpha$. Therefore we can clear out the common divisors of $G$ and call it $G^*$. Let $\alpha^* = B(z)G^*(z)$.

Finally we show that $B(0) = 1$. Suppose there exists $p \in D$ a prime such that $p \mid B(0)$. Then $B(z)G^*(z) = \alpha^*$ implies that $B(0)G^*(0) = \alpha^*$ and so $p \mid \alpha^*$. Now consider $\overline{B}(z)\overline{G^*}(z) = 0 \in D/pD[[z]]$. Since $D$ is a unique factorization domain, then $D/pD$ is an integral domain and so $D/pD[[z]]$ is also an integral domain. So $\overline{B} = 0$ or $\overline{G^*} = 0$. Thus $p$ is a common divisor of the coefficients of $B$ or $G^*$. This is a contradiction from the previous paragraphs and so there is no $p \in D$ that divides $B(0)$ and so $B(0)$ is a unit of $D$. Therefore we can assume that $B(0) = 1$.

So $a(z) = \frac{A(z)}{B(z)}$ and $B(0) = 1$. Further $B$ is a minimal recurrence of $\{a_k\}_{k=0}^{\infty}$ and $B \in D[z]$. So the reverse of $B$ is $f_{\min}$ and $f_{\min} \in D[z]$. ■

Theorem 8 allows us to compute the unique minimal generator of a scalar sequence in a unique factorization domain using Algorithm 3.2.1. By performing a final division by

$h$ before returning the generator, we can compute $\pm f_{\min}$. This result does not extend to general matrix sequences. A counterexample is the following sequence:

$$a_0 = \begin{bmatrix} 3 & 2 \\ 0 & 1 \end{bmatrix}, a_1 = \begin{bmatrix} 3 & 2 \\ 0 & 1 \end{bmatrix}, a_2 = \begin{bmatrix} 6 & 4 \\ 0 & -2 \end{bmatrix}, a_i = \begin{bmatrix} 3 \cdot Fibo_i & 2 \cdot Fibo_i \\ 0 & -(a_{i-1})_{2,2} - (a_{i-2})_{2,2} \end{bmatrix}.$$

This sequence has a unique right minimal matrix generator given by:

$$f(z) = \begin{bmatrix} z^2 - z - 1 & -\frac{4}{3}z - \frac{4}{3} \\ 0 & z^2 + z + 1 \end{bmatrix}.$$

We know that $f$ is the unique right minimal matrix generator because it is in (descending degree) column Popov form. Algorithm 3.2.1 confirms that this sequence is a normal remainder sequence.

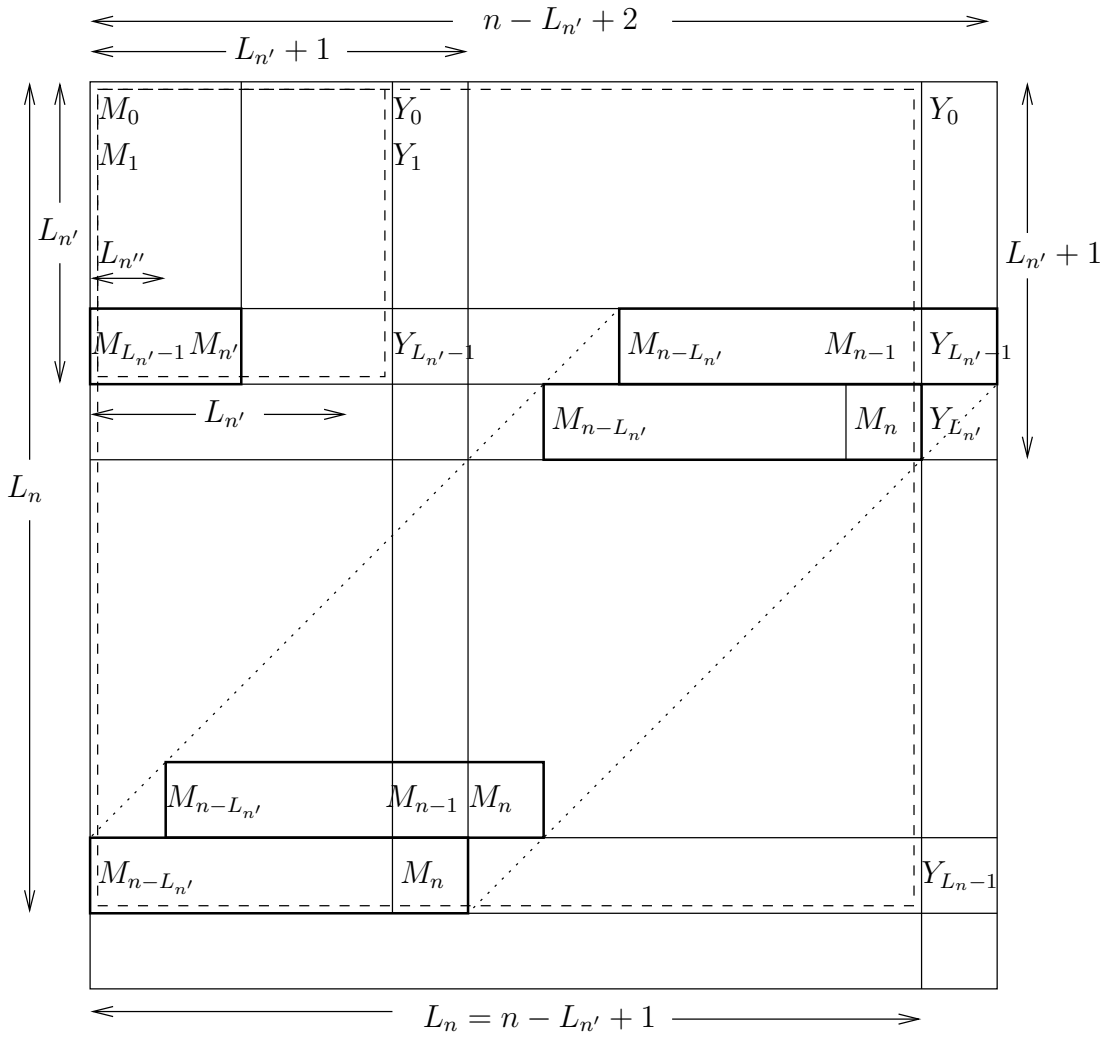


Figure 3.2: Block-Hankel Linear System Solver

# Chapter 4

# Implementing the Matrix Berlekamp/Massey Algorithm

Algorithm 2.2.1 presents interesting design choices for any implementation. We will describe two implementations, a Maple implementation and a C++ implementation in the LinBox library. The Maple implementation is straightforward using various native packages and so will be briefly described. The LinBox implementation is a more generic design, to take advantage of features of both C++ and LinBox. A more detailed description will be given to this implementation. The design choices intrinsic to the algorithm include representation of matrix sequences, matrix polynomial representation, field arithmetic implementation and matrix operation implementation. A discussion of each of these choices is given for each implementation. Finally a performance comparison of the two implementations is given.

## 4.1 Maple Implementation

The computer algebra system Maple includes many built in tools that make the implementation of Algorithm 2.2.1 straight forward. Field arithmetic can be performed over the finite field $\mathbb{Z}_p$ using modular arithmetic and rational arithmetic is done to arbitrary precision. This requires two versions of the implementation, one rational and one for finite fields. The *LinearAlgebra* package contains all of the necessary matrix operations for Algorithm 2.2.2. As in the description of Algorithm 2.2.1, both the sequence and the intermediate generator can be represented as a matrix polynomial natively. The *Matrix-*

*PolynomialAlgebra* package contains all of the necessary functions required by the algorithm for matrix polynomials. Maple makes the choice of data structures for the sequence and the polynomial simple, since it hides and handles the underlying structure.

The benefits of Maple are offset by computational shortcomings that are inherent in the system. Since the implementation is not compiled and requires the invocation of the Maple kernel, there is natural overhead when using it. More pressing is the lack of fast and dedicated field arithmetic algorithms for working over finite fields. The finite field implementation uses modular arithmetic over a prime field in a two step process. First, the matrix operations are performed over the integers and then every entry is reduced mod the given prime. It would be preferable that the elements of the matrices have a field structure that would perform the arithmetic more efficiently. This would also allow more general fields of finite character to be used, something that is lacking in the current Maple implementation.

## 4.2 LinBox Implementation

LinBox is a C++ template library that provides generic implementations of black box linear algebra algorithms [16]. The library was developed by a consortium of universities in Canada, France and the USA. The goal is to supply "efficient black box solutions for a variety of problems including linear equations and matrix normal forms with the guiding design principle of re-usability" [16]. The library uses the C++ template instantiation mechanism to compile code for the most efficient ways of performing the arithmetic in the various entry fields [16; 17; 38]. The algorithms and matrix implementations are designed for use over finite fields and so LinBox has specialized implementations for finite field arithmetic. The implementation of the Matrix Berlekamp/Massey algorithm will take advantage of these finite field implementations.

The design of the LinBox implementation involves two distinct abstractions. The first is how to represent matrix sequences and the second is the implementation of the actual algorithm. These two seemingly disparate parts have been combined into an overall abstract data type. The implementation is patterned after the Standard Template Library, or STL [34]. The STL provides generic implementations of common abstract data types such as lists, maps and vectors. These containers have associated objects called iterators that provide access to the elements in the container. The implementation of matrix sequences

*BM_Seq* is modeled after an STL list. The algorithm is implemented as a specialized iterator, *BM_iterator*, that will perform each iteration of Algorithm 2.2.1 one step at a time.

In the Standard Template Library, there are a few basic categories of objects. Two of these are *containers* that hold elements of a type with some structure and *iterators* that point to objects in a container. The container is an abstraction of abstract data types such as lists, vectors and sets. Each container maintains its data in its own way with its own access methods. The containers provide varied methods for insertion and deletion. The iterators are abstractions of pointers, allowing for easier reassignment of the object that the iterator points to. Depending on the container, the iterator moves across the elements in the container in different ways [34].

### 4.2.1 BM_Seq description

The *BM_Seq* object interface is patterned after the STL list container but it is not a subclass of the list container. The declaration of the *BM_Seq* class is given in Figure 4.1. The class is templated by the template parameter *_Field*, which defines the LinBox field implementation of which the elements of the matrices will be a member. The template parameter is integrated into the *BM_Seq* by the typedef of *Field*. This type assignment is done so that the template parameter can be access in the form *DenseMatrix<_Field>::Field*. Without this typedef, the template parameter *_Field* would not be accessible to the *BM_iterator*. The last three typedef's shown in Figure 4.1 are types common to most STL containers. The *value_type* is the type of objects contained in the container. In *BM_Seq* the value type is the type of matrix in the sequence. This is defined by the *Field* template parameter and is defined to be a LinBox implementation of a dense matrix with entries in the given field, *DenseMatrix<Field>*. The typedef for *const_iterator* provides the only available iterator for *BM_Seq* and is a constant iterator of a list of dense matrices. Unlike normal STL containers, *BM_Seq* only contains one type of iterator, a constant iterator. This decision was made for the same reason that *BM_Seq* is not sub-classed from the list container, but only maintains a list as a private member, as seen in Figure 4.2. The desire for this design of *BM_Seq* is so that we can control the access to the elements in the *BM_Seq*.

The STL list container includes bi-directional forward and reverse iterators in both const and non constant varieties. The constant iterators move across a container but similar to a constant pointer, do not allow the contents of element that they point to to be altered.

```
template<class _Field>
class BM_Seq {

  public:

    typedef _Field Field;
    typedef DenseMatrix<Field> value_type;
    typedef typename std::list<value_type>::const_iterator
        const_iterator;
    typedef int size_type;
```

Figure 4.1: BM_Seq Declaration

We wanted the *BM_Seq* to have a very stringent interface that controlled how the elements are accessed and how they are added. The hope is to make a insertion safe container and a specialized iterator that is also safe [21]. The design of *BM_Seq* includes only what we felt a matrix sequence needs and removes all of the unnecessary functionality of a list by encapsulating it as a private member. Matrices cannot be altered once they are added to the sequence. This will ensure that any *BM_iterator* of the *BM_Seq* is always valid. For similar reasons, matrices can only be added at the end of the sequence. Thus unlike the standard list container, we do not allow front or middle insertion. Finally by encapsulating the list instead of inheriting from it, we can rewrite the size member function so that it is constant time. The size member function for all STL containers is only guaranteed to be linear time, and the list container is one of the non-constant time containers.

```
  private:
    Field& F;
    std::list<value_type > seq;
    size_type _size;
    size_t row, col;
```

Figure 4.2: BM_Seq private members

The constructors for *BM_Seq*, shown in Figure 4.3, are based on constructors of

the list container but reflect the necessities of *BM_Seq*. The first two constructors create an empty sequence while defining the field and matrix dimensions of sequence. The second constructor is a specialization that creates a square matrix sequence and so only one of the matrix dimensions is needed. These two constructors should be considered the "default" constructors of *BM_Seq*. The third constructor creates a sequence with a length $n$ prefix of the matrix $M$. The field and dimension members are obtained from the matrix $M$. The final constructor is a copy constructor. No default constructor or assignment operator is provided. The *DenseMatrix* class has no default constructor because it contains a reference member that cannot be constructed by default or overwritten by assignment. Therefore *BM_Seq* cannot have either of these as well.

```
public:
  BM_Seq(Field& _F, size_t r, size_t c) : F(_F) {
  /* code omitted */
  }
  BM_Seq(Field& _F, size_t r) : F(_F){
  /* code omitted */
  }
  BM_Seq(int n, value_type& M) : F(M.field()),  seq(n, M), _size(n){
  /* code omitted */
  }

  BM_Seq(const BM_Seq<Field>& S) : F(S.F), seq(S.seq), _size(S._size),
      row(S.row), col(S.col){}
```

Figure 4.3: BM_Seq constructors

The member functions of *BM_Seq*, shown in Figures 4.4 and 4.5 reflect its connection to both STL and LinBox. The *field()*, *rowdim()*, and *coldim()* functions are common member functions found in all LinBox matrix classes. They are presented so that other algorithms and classes can have access to this information in the common LinBox interface. The member functions *begin()*, *end()*, *push_back()* and *size()* are taken from the STL list container. In an STL list, the *begin()* member function returns an iterator pointing the beginning of the list, and so does ours. The main difference is that our member function will only return a *const_iterator* since as stated before, we wish to protect the matrices in

the sequence from being changed after they have been added to the sequence. The *end()* function returns an iterator that points one past the end of the sequence (or underlying list in this case). This function should only be used to create an iterator for boolean tests designed to find the end of the sequence or list. Dereferencing or decrementing the *end()* iterator is not suggested and the results of such actions are not guaranteed to be safe. The STL list provides a reverse_iterator and related functions to quickly access the end of a list. However, it was decided that *BM_Seq* should be a forward only container and so the only way to access the last matrix in the sequence is to start at the beginning and increment to the last. This restriction does not mean that the *const_iterator* is one directional, it is a bi-directional iterator, it just means that access to the last element is not provided by a member function.

```
public:
  Field& field(){ return F; }
  size_t rowdim(){ return row; }
  size_t coldim(){ return col; }
```

Figure 4.4: BM_Seq member functions

The *push_back* member function is the only insertion function for *BM_Seq*. The STL list container has many insertion routines. Besides the *push_back* insertion at the end of a list, there are front and middle insertion routines. We did not want to allow these insertion methods to *BM_Seq* so the specialized iterator, *BM_iterator*, remains valid and "safe" on insertion. In the Standard Template Library, the iterators of a container are not guaranteed to remain valid after an insertion or deletion routine. That this is true for deletion is straight forward since an iterator may point to the element that has been removed. The reason insertion corrupts iterators is because the underlying storage of the container may change and so the iterator becomes invalid. For example, a container may always store its data in a contiguous block of memory and the insertion of just a single element could force a new larger memory allocation and copy. We wanted to make *BM_iterator* safe, so that even after an insertion, any *BM_iterator* is still valid. Since adding a matrix to the end of a sequence just adds more information to Algorithm 2.2.1, then the *push_back* operation does not corrupt the *BM_iterator*. Any front or middle insertion routine could invalidate the generator that the *BM_iterator* has computed.

```
public:
  const_iterator begin() const { return seq.begin(); }
  const_iterator end() const { return seq.end(); }
  void push_back(const value_type &M){
  /* code omitted */
  }
  bool operator==(const BM_Seq<Field>& l){
  /* code omitted */
  }
  bool operator!=(const BM_Seq<Field>& l){
  /* code omitted */
  }
  size_type size(){ return _size; }
```

Figure 4.5: BM_Seq member functions continued

The final members of *BM_Seq* are the *BM_iterator* class definition and the member functions related to *BM_iterator*. These are given in Figure 4.6. The member type *BM_iterator* is the specialized iterator that implements Algorithm 2.2.1. A detailed description of *BM_iterator* will be given in Section 4.2.2. In STL, the iterators of a container are encapsulated types that can only be accessed through the container. Iterators are also initialized by calling the member functions *begin* and *end* of the container. So as our *BM_Seq* has a *begin* and *end* initialization function for the *const_iterator*s, so too *BM_Seq* has two initialization member functions for *BM_iterator*. The *BM_begin()* member function will create a new initialized *BM_iterator*. This *BM_iterator* points to the first element in the sequence in the sense that the newly constructed *BM_iterator* initializes all of the variables in the initialization of Algorithm 2.2.1. So when *BM_begin* returns the new *BM_iterator*, the iterator is ready to process the first element of the sequence, ie perform Algorithm 2.2.1 for $t = 0$. The member function *BM_end()* returns a *BM_iterator* that points one past the last element of the sequence. It does not process the elements of the sequence by performing Algorithm 2.2.1 it just sets an private pointer and state member to the end of the sequence. The *BM_end* iterator should only be used for boolean tests or other iterators,however the definition of *BM_iterator* will provide better state tests that will

be described in Section 4.2.2.

```
public:
  class BM_iterator {
  /* code omitted */
  }
  //return an initialized BM_iterator
  typename BM_Seq<Field>::BM_iterator BM_begin(){
    return typename BM_Seq<Field>::BM_iterator(*this);
  }
  //return an initialized BM_iterator that points to one past the end
      of the sequence
  typename BM_Seq<Field>::BM_iterator BM_end(){
    return typename BM_Seq<Field>::BM_iterator(*this, _size);
  }
```

Figure 4.6: BM_Seq BM_iterator member

### 4.2.2 BM_iterator description

In the Standard Template Library, iterators are a common object interface that act as pointers to elements in a container. The iterators are encapsulated classes so that an iterator of a list of integers is not the same type as an iterator of a vector of integers or a list of floats. Iterators have only a few basic operations. The dereference operator, "*" returns the element that the iterator points to. The increment operator, "++" advances the iterator to the next element in the container. These are the only operations for "forward" iterators. As the name implies, forward iterators can only move in one direction. Bi-directional iterators include both an increment and decrement operation. The decrement operator, "–", moves the iterator to the element before the current element. To understand this, visualize the elements in the container written from left to right. Then the increment operator moves the iterator one to the right and the decrement operator moves the iterator one to the left.

The *BM_iterator* is a forward iterator that implements Algorithm 2.2.1. The iterator points to the $n$th matrix in the *BM_Seq* in the sense that the *BM_iterator* will

have executed Algorithm 2.2.1 for all $t$ such that $0 \leq t \leq n-2$. Thus the *BM_iterator* will be ready to process the matrix that it points to. The increment operator will perform one loop iteration of the algorithm, step MBM4 through step MBM9. After performing the loop iteration, the iterator points to the next matrix, meaning it is ready to process it. Dereferencing the *BM_iterator* will return the current generator, $f$. The remaining member functions of *BM_iterator* address specific needs and desired outputs of Algorithm 2.2.1.

The declaration of *BM_iterator* shown in Figure 4.6 will be shown and discussed in more detail in this section. The first public member of *BM_iterator* is the *value_type*, shown in Figure 4.7. The iterators of an STL container always have an associated value_type. For normal iterators, its value_type is the same as the container to which it belongs. The value_type of an iterator is the type returned by the dereferencing operator. Our *BM_iterator* does not share the same value_type as *BM_Seq*. The specialized *BM_iterator* value_type is instead a STL list of *BM_Seq<Field>::value_type*. This is how we will store the current generator $f$ of Algorithm 2.2.1. We use a list instead of a vector, which may seem more natural in storing a matrix polynomial, because it is easier to expand a list by one than it is to expand a vector. Step MBM9 sometimes requires that the degree of $f$ grow. Both STL lists and vectors have the capacity to grow in size, but vectors do so in blocks of memory. This causes an error for many LinBox matrix implementations because they contain a reference member. The presence of a reference member means that the default constructor and assignment operator cause a compile time error. Using an STL list circumvents this problem and does not add any added complexity to the computation of $\Delta$ in step MBM4.

```
class BM_iterator {
  public:
    typedef std::list<typename BM_Seq<Field>::value_type> value_type;
```

Figure 4.7: BM_iterator declaration

Most of the private members of *BM_iterator*, shown in Figure 4.8 are either taken directly from Algorithm 2.2.1. The private members that do not have a direct corollary to the algorithm are present to assist in the algorithm and to make the iterator "safe". The members *t*, *delta*, *mu*, *beta* and *sigma* are the same as in the algorithm. The member *gen* is the current generator $f$ of Algorithm 2.2.1, while the vector *deg* is the vector of nominal degrees for each column of $f$. The members *row* and *col* are generalized forms of $N$ from

the algorithm, since this implementation is generalized for rectangular matrix sequences. The member *gensize* is the size of the current generator *gen* and will be used to determine when the degree of the generator has grown due to the multiplication by $z$ in step MBM9.

The private member *seq* of *BM_iterator* is the input matrix sequence $M(z)$ of Algorithm 2.2.1. As Figure 4.8 shows, this is a reference member that is contains a reference to a *BM_Seq*. Thus a *BM_iterator* of a *BM_Seq* will maintain a reference to that *BM_Seq*. Since the *BM_Seq* and any *BM_iterator* are intrinsically linked, that relationship is protected by this reference type member. By giving the *BM_iterator* a reference to its *BM_Seq*, the *BM_iterator* is able to monitor the sequence and determine when it has been altered by insertion. Normal iterators and containers do have this type of connection and so changing the container may corrupt the iterator. By allowing the *BM_iterator* to know query for details of the *BM_Seq*, then our iterator is safe upon element insertion. The *BM_iterator* members *size* and *seqel* will be used to maintain the connection between the iterator and its *BM_Seq*. The last known size of the *BM_Seq* is stored in *size* while *seqel* is an *BM_Seq* const_iterator that points at the matrix to be processed next, the $t+1$ positioned matrix in the sequence. The increment operator will test to see if the size of the sequence has changed since and will reset both the *size* and *seqel* members accordingly if the size has increased.

The execution of the main loop of Algorithm 2.2.1 is controlled by the loop condition given in step MBM2. To mimic this loop condition and provide a user of the *BM_iterator* with its information, the *BM_iterator* includes an encapsulated enumeration class called *TerminationState*, shown in Figure 4.9. The class has four states, *GeneratorUnconfirmed*, *GeneratorFound*, *DeltaExceeded* and *SequenceExceeded*. The state *GeneratorUnconfirmed* implies that the loop condition is true and the algorithm can safely perform another iteration. The state *GeneratorFound* implies that the loop condition has failed and the algorithm has found the generator of the sequence defined by the given $\delta$. If the *BM_iterator* has a state of *DeltaExceeded*, then the loop condition is false and the given $\delta$ has been proven to be too small for the *BM_Seq*. Finally, the state *SequenceExceeded* is not related to the loop condition, but indicates that all of the elements in the *BM_Seq* have been processed by the *BM_iterator* and a generator has not been found.

Figure 4.9 also shows the public member function *setDelta()*. This member function will reset the member *delta* to the desired value. When the value *delta* is set or reset, the *TerminationState* member em _state is updated as well. The private member *_state* is computed using the members *delta*, *mu*, *beta*, *sigma* and *seqel*. The value of *delta* was made

```
private:
  typedef typename BM_Seq<Field>::value_type matrix_type;
  Field& F;
  BM_Seq<Field>& seq;
  typename BM_Seq<Field>::size_type size;
  typename BM_Seq<Field>::size_type t;
  typename BM_Seq<Field>::const_iterator seqel;
  std::list<matrix_type> gen;
  std::vector<int> deg;
  int delta;
  int mu;
  int beta;
  int sigma;
  int gensize;
  size_t row, col;
```

Figure 4.8: BM_iterator private members

```
public:
  class TerminationState{
  /* code omitted */
  }
  void setDelta(int d) {
  /* code omitted */
        }
private:
  TerminationState _state;
```

Figure 4.9: BM_iterator state members

a member of *BM_iterator* and not *BM_Seq* to allow for multiple *BM_iterator*'s, each with its own *delta*. Further it makes more sense for the *BM_iterator* to be associated with the *delta* since it is the actual implementation of Algorithm 2.2.1 and $\delta$ is an input to the algorithm. The value of the *TerminationState* class contains a private member which holds its value. This value is set by the *BM_iterator*, which is declared a "friend" of *TerminationState*, as seen in Figure 4.10. A friend of any class has access to its private members and private member functions. Even though *TerminationState* is an encapsulated class, *BM_iterator* is not automatically a friend and so must be declared as such. Further an encapsulated class is not automatically a friend of its outer class meaning *BM_iterator* cannot access the private members of *BM_Seq*. This is another reason that *BM_iterator* contains a reference to its *BM_Seq*. Notice that the constructors of *TerminationState* is private, meaning only a friend, like *BM_iterator*, can construct an object of this type. Further, since *BM_iterator* is a friend, it can set the *state* value of *TerminationState* directly. The value of a *TerminationState* object can be queried through the boolean member functions.

```
class TerminationState{
  private:
    int state;
    friend class BM_iterator;
    TerminationState() : state(GeneratorUnconfirmed) {}
    TerminationState(int m) : state(m) {}
  public:
    TerminationState(const TerminationState& t) : state(t.state) {}
    TerminationState & operator=(const TerminationState & t){
    /* code omitted */
    }
    bool IsGeneratorUnconfirmed() { return state==GeneratorUnconfirmed;
       }
    bool IsGeneratorFound() {return state==GeneratorFound; }
    bool IsSequenceExceeded() { return state==SequenceExceeded; }
    bool IsDeltaExceeded() { return state==DeltaExceeded; }
};
```

Figure 4.10: BM_iterator TerminationState

There are only two constructors for *BM_iterator*. These constructors are shown in Figure 4.11. The first constructor is a standard constructor that initializes all of the members. Two of those initializations are shown because they are not straight forward. First you will see that *delta* is initialized to $-1$. There is no parameter for *delta* to be set to any other value on initialization since it is not intended for a user to create a *BM_iterator* without using the *BM_Seq* member function *BM_begin*. This member function takes no arguments, patterned to act like the STL begin, and so the *delta* should be set using *setDelta* after construction. The initial value of $-1$ is used to mimic infinity. If the *delta* is never set then the state of the *BM_iterator* will be *GeneratorUnconfirmed* until the entire sequence has been processed. When the iterator has been incremented to the end the state will change to *SequenceExceeded*.

The second initialization shown is that of $t$ and *_state*. The constructor parameter *elinit* is used to create the "end" *BM_iterator* returned by the *BM_end* member function of *BM_Seq*. To create the end *BM_iterator*, the iterator that points to the last element of the sequence, we do not want to run the algorithm. So this parameter just sets the appropriate values to look like it is at the end. The values associated with Algorithm 2.2.1, other than $t$, are initialized according to the algorithm description. This end iterator is present for completeness, so that the *BM_iterator* is patterned after a STL iterator. Further this method of setting the value of the iterator to the end iterator using a parameter of the constructor is done in the Standard Template Library. Note that if the *BM_Seq* is empty, $size = 0$, then the iterator is also the end *BM_iterator*. In practice, one should refrain from the standard iterator boolean tests and use the *_state* member to determine the state of the *BM_iterator*.

The second constructor shown in Figure 4.11 is copy constructor. No default constructor is provided just as in the case of *BM_Seq*. Further, no assignment operator is given. This is due to the reference members present in the *BM_iterator*. The presence of these reference members and the known LinBox matrix reference member problems make the default constructor and assignment operator untenable to provide.

The main action of the *BM_iterator* is to execute Algorithm 2.2.1. The heart of the algorithm is the loop of steps MBM4 to MBM9. These steps will performed my incrementing the *BM_iterator* using the "++" operator. Figure 4.12 shows the declaration of the two increment operators. The computation of $\tau$ in step MBM5 is performed by the private member function *Algorithm2dot3dot2*. This in an implementation of Algorithm 2.2.2 that

```
public:
  explicit BM_iterator(BM_Seq<Field>& s, typename BM_Seq<Field>::
      size_type elinit=0) : F(s.field()), seq(s) {
  /* code omitted */
  delta = -1;
  /* code omitted */
  t = elinit;
  if(size==0 || t==size)
    _state.state = SequenceExceeded;
  /* code omitted */
  }
  BM_iterator(const BM_Seq<Field>::BM_iterator & it) : F(it.F), seq(it
      .seq), size(it.size), t(it.t), seqel(it.seqel), gen(it.gen), deg
      (it.deg), delta(it.delta), mu(it.mu), beta(it.beta), sigma(it.
      sigma), gensize(it.gensize), row(it.row), col(it.col), _state(it
      ._state) {}
```

Figure 4.11: BM_iterator constructors

uses column operations implemented as private member functions of *BM_iterator*. The two increment operators are differentiated as the "pre" and "post" increment operators. The pre-increment operator performs the loop iteration and returns the iterator that points to the next matrix in the sequence. The post-increment operator, returns the iterator that points at the current matrix and then performs the loop iteration. The post-increment operator is the one with the parameter while the pre-increment is the void operator. This is the C++ way to differentiate the two and allow overloading both operators. Further note the different returns of the two operators. The pre-increment operator returns a reference to the incremented *BM_iterator*, and the post-increment operator returns by copy a copy of the iterator before the execution of the loop.

```
private:
  template <class Matrix>
  Matrix  Algorithm2dot3dot2(Matrix &D, std::vector<int> &d, int &mu,
      int &sigma, int &beta){
  /* code omitted */
  }
public:
  BM_iterator& operator++(){
  /* code omitted */
    return *this;
  }
  BM_iterator operator++(int){
    BM_iterator temp(*this);
  /* code omitted */
    return temp;
  }
```

Figure 4.12: BM_iterator increment operators

The increment operators execute one iteration of the main loop of Algorithm 2.2.1. They also maintain the safety of the *BM_iterator* by updating the size of the *BM_Seq* and ensuring the validity of the *seqel* member. If the *BM_Seq* has had one or more matrices inserted by the *push_back*, then *seqel* a iterator of the sequence and in turn an iterator of the list maintained by the sequence could have been corrupted. Both the pre and post-

increment operators include the if statement in Figure 4.13. This check tests if the *size* stored in the *BM_iterator* matches the size of the sequence. If it does not, then *seqel* is reset the beginning of the sequence and incremented to the appropriate matrix. The value of *size* can only be less than or equal to the actual size of sequence since matrices can only be added and not removed. The call to the *size* member function of *BM_Seq* is constant time since we did not sub-class *BM_Seq* from the STL list. Both increment operators set the *TerminationState* member *_state* after the loop iteration is executed according to the loop condition of Algorithm 2.2.1.

```
if(size < seq.size()){
  seqel = seq.begin();
  for(int i = 0; i<t; i++)
    seqel++;
  size = seq.size();
}
```

Figure 4.13: BM_iterator increment operators

The dereference operator of *BM_iterator* is overloaded. Shown in Figure 4.14, the "*" operator returns a reference to the value_type of *BM_iterator*. This is the same as STL iterators, except that the *BM_iterator* and *BM_Seq* value_type are different while the STL iterators and corresponding containers have the same value_type. The *BM_iterator* points to the most relevant data of Algorithm 2.2.1, the current generator and returns this data when dereferenced. The element of the *BM_Seq* that the *BM_iterator* points to is the matrix of the sequence that will be the next to be tested by the current generator when the discrepancy $\Delta$ is computed. This design seemed to be the most natural when it was decided to implement Algorithm 2.2.1 as a specialized iterator. Since iterators are generalizations of pointers, the data can also be accessed through the "$->$" operator. Thus this operator has been overloaded for the *BM_iterator* accordingly.

The final members of *BM_iterator* are data access member functions. These members, shown in Figure 4.15, allow a user to probe the *BM_iterator* for current peripheral data not contained in value_type. In normal STL iterators there is only one piece of data, the value_type that is accessed by dereferencing the iterator. The *BM_iterator* has other important data arising from the execution of Algorithm 2.2.1. So there are member functions

```
public:
  value_type& operator*(){
    return gen;
  }
  value_type* operator->(){
    return &gen;
  }
```

Figure 4.14: BM_iterator dereference operators

to find the current value of *t*, *beta*, *sigma*, *mu* and *delta*. Also there is a member function to get a copy of the *_state* member. After obtaining the state of the *BM_iterator*, the boolean member functions of *TerminationState* can then be used to find the state of algorithm. The member *GetGenerator* is not the same as deferencing the *BM_iterator*. The current generator *gen* is the implementation of the matrix polynomial $f$ of Algorithm 2.2.1. So *gen* is in reversed form and contains both generator and auxiliary columns. The *GetGenerator* member will perform step MBM10 and return only the generator that has been computed. It will return said generator in the reversed form from *gen* according to the degree *mu*. There are no checks in place, so *GetGenerator* may be used at any time regardless of the value of the *_state* member.

### 4.2.3 BM_Seq example

An example of how to use *BM_Seq* and *BM_iterator* to execute Algorithm 2.2.1 is given in Figure 4.16. The field *Modular<uint32>* is a LinBox field class over unsigned 32-bit integers. The actual field object $F$ is the field over the prime $q$. The *BM_Seq* object *seq* is templated by *Field* with row dimension $r$ and column dimension $c$. Matrices like *m0* are added to the sequence using the member function *push_back*. The *BM_iterator* object *bmit* is created using the *BM_begin* member function. The bound $\delta$ of Algorithm 2.2.1, which is also the member *delta* of *BM_iterator*, is set to be $d$ using the member function *setDelta*. After setting the bound, the state of the *BM_iterator* is obtained and stored in *check*. The while loop is controlled by *check*. If *check.IsGeneratorUnconfirmed()* returns true, then the main loop of Algorithm 2.2.1 must be executed at least one more iteration.

```
  public:
    TerminationState state() const { return _state; }

    std::vector<matrix_type> GetGenerator(){
    /* code omitted */
    }

    typename BM_Seq<Field>::size_type get_t() { return t; }
    int get_mu() { return mu; }
    int get_sigma() { return sigma; }
    int get_beta() { return beta; }
    int get_delta() { return delta; }
};
```

Figure 4.15: BM_iterator data access members

This procedure is performed by incrementing *bmit*. Here a post-increment has been used, but a pre-increment would also be fine. After the increment, *check* is updated for the next loop test. When the loop is terminated, the state variable *check* is probed to see how the loop terminated.

Note that the line in Figure 4.16 that constructs the *BM_iteratorbmit* is initialization and not assignment. That difference is important because initialization uses the copy constructor, which exists, and not the assignment operator, which does exist. In C++, declarations like that of *object1* in Figure 4.17 are always initialization [39]. The object *Class(data)* is created and then *object* is initialize with this constructed object. The construction of *object2* by passes this by using direct initialization via the constructor of *Class* defined to accept *data* as a parameter. The declaration of *object3* is not initialization. The default constructor of class *Class* is used to construct *object3* and then the value of *object1* is assigned to *object3* by the overloaded assignment operator of *Class*.

## 4.3 Comparison of implementations

We conducted three timing comparisons of the Maple implementation and the LinBox implementation. The implementations performed field operations over the finite

```
typedef Modular<uint32> Field;
Field F (q);
BM_Seq<Field> seq(F, r, c);
DenseMatrix<Field> m0(F,r,c);
seq.push_back(m0);
//Add more matrices to the sequence
BM_Seq<Field>::BM_iterator bmit = seq.BM_begin();
bmit.setDelta(d);
BM_Seq<Field>::BM_iterator::TerminationState check = bmit.state();
while(check.IsGeneratorUnconfirmed){
        bmit++;
        check = bmit.state();
}
if(check.IsBoundExceeded())
  std::cout << "The determinantal bound is incorrect" << std::endl;
if(check.IsSequenceExceeded())
  std::cout << "More matrices needed to compute generator" << std::
     endl;
if(check.IsGeneratorFound())
  vector<DenseMatrix<Field> >gen(bmit.GetGenerator());
```

Figure 4.16: Using BM_Seq and BM_iterator

```
Class object1 = Class(data);
Class object2(data);
Class object3;
object3 = object1;
```

Figure 4.17: Methods of initialization and construction

field $\mathbb{Z}_p$ for $p = 2$, $p = 101$ and $p = 2157483647 = 2^{31} - 1$. For each value of $p$, ten random linearly generated square matrix sequences of dimensions 2, 3, 4, 5, and 6 were created. The sequences were formed as bilinear projections, $\{U^T A^i V\}_{i=0}^{\infty}$. The matrices $U$, $A$ and $V$ were randomly generated in Maple, with $A$ a $60 \times 60$ square matrix. Thus the determinantal bound $\delta$ was 60 for all of the sequences. The test matrix sequences generated in Maple, were written to files so that both implementations were tested on the same sequences. The timings represent the average computation time of the Matrix Berlekamp/Massey algorithm in both implementations for the ten sequences of each dimension. The results of the comparisons are given in Table 4.1, Table 4.2 and Table 4.3.

Table 4.1: Implementation timings; $p = 2$

| | Maple | LinBox |
|---|---|---|
| Dimension | time in seconds | |
| 2 | 4.648000 | 0.062 |
| 3 | 3.677200 | 0.06 |
| 4 | 3.666800 | 0.065 |
| 5 | 3.847400 | 0.073 |
| 6 | 4.381500 | 0.082 |

Table 4.2: Implementation timings: $p = 101$

| | Maple | LinBox |
|---|---|---|
| Dimension | time in seconds | |
| 2 | 9.872000 | 0.06 |
| 3 | 7.730700 | 0.055 |
| 4 | 7.258300 | 0.058 |
| 5 | 7.501000 | 0.07 |
| 6 | 8.083400 | 0.08 |

Table 4.3: Implementation timings: $p = 2147483647 = 2^{31} - 1$

| | Maple | LinBox |
|---|---|---|
| Dimension | time in seconds | |
| 2 | 12.085900 | 0.058 |
| 3 | 9.932100 | 0.055 |
| 4 | 9.673800 | 0.06 |
| 5 | 10.406600 | 0.068 |
| 6 | 11.699800 | 0.076 |

The results of this timing comparison show two differences in the implementa-

tions. First, the LinBox implementation is far superior to the Maple implementation. The combination of a dedicated finite field arithmetic and compiled code allows the LinBox implementation to run about 70 to 150 times faster depending on the dimension of the sequence and the size of the prime. Second, notice that the times of the Maple implementation increase as the size of the prime $p$ increases while the times of the LinBox implementation are relatively constant for each prime. Here is another place where the dedicated finite field arithmetic is superior, since the LinBox implementation used the same type of field, with the same element type. Thus the arithmetic over $p = 2$ and $p = 2147483647$ has the same complexity.

Both the Maple and LinBox implementations exhibit a consequence of the termination criterion. For every $p$ and in both implementations, the average time for the dimension 2 sequences is larger than that of the dimension 3 sequences. In the Maple implementation, the dimension 2 average computation time is larger than all of the other dimensions. This occurs because when the dimension changes from 2 to 3, the algorithm does not need to see as many matrices in the sequence to find the generator. In fact the algorithm processes about 60 matrices in the dimension 2 case while processing only 40 matrices in dimension 3 case. This earlier termination offsets the costs of performing column operations on the larger matrices and so the time decreases. In fact, the smallest computation time for the Maple implementation for each $p$ is the dimension 4 test case. This is not the case in the LinBox implementation, presumably due to a simpler but higher complexity implementation of the column operations.

The LinBox implementation was also tested on a large example. A bilinear projection over $\mathbb{Z}_2$ was created from a square matrix with dimension of $10,000$ and block matrices of size $10,000 \times 100$. Thus the minimal matrix generator has determinantal degree bound $10,000$. The computation of the first four hundred sequence elements took 260 hours. The computation of the minimal matrix generator took only 4 hours.

# Chapter 5

# Summary and Conclusions

We have presented two matrix generalizations of the Berlekamp/Massey algorithm and described two implementations of the Matrix Berlekamp/Massey algorithm. We began with a full definition and description of linearly generated matrix sequences in Chapter 2. A complete classification of all linearly generated matrix sequences is given by Theorem 3. It states that all linearly generated matrix sequences can be associated to a square matrix and the bilinear projection of its matrix powers. In Chapter 2 we also give a description of the Matrix Berlekamp/Massey algorithm and a complete proof of correctness. The termination criterion of Algorithm coppersmith is definitive. Theorem 5 implies that the generator computed by the Matrix Berlekamp/Massey algorithm with the given determinantal bound $\delta$ is the minimal matrix generator for the completion of the matrix sequence prefix, defined by Lemma 12. The chapter ends with a complexity analysis of the Matrix Berlekamp/ Massey algorithm.

We present a fraction free Matrix Berlekamp/Massey algorithm in Chapter 3. The algorithm works for any scalar sequence and any normalizable remainder sequence. If the sequence is not a normalizable remainder sequence, then the algorithm may return a certificate that is not. The scalar form of Algorithm 3.2.1 maintains smaller intermediate values than the previous known algorithm. A description of the fraction free Matrix Berlekamp/Massey algorithm as a block Hankel solver is also given in Chapter 3. Theorem 8 allows us to compute the unique minimal generators of sequences in unique factorization domains such as $Z$ or $F[X]$ using our fraction free scalar algorithm. It states that any linearly generated sequence whose elements are integral has an integral monic minimal generator.

Two implementations of the Matrix Berlekamp/Massey algorithm are described in Chapter 4. The implementation in Maple is not given in detail, but the LinBox implementation is. The algorithm and matrix sequences are implemented as a connected data structure. Patterned after the Standard Template Library list container, *BM_Seq* is a implementation of a matrix sequence. It stores the matrices in a sequential order and allows new matrices to be inserted only on the end. The Matrix Berlekamp/Massey algorithm is implemented as a specialized iterator of *BM_Seq*, called *BM_iterator*. Just as STL iterators point to an element in a container, the *BM_iterator* points to the matrix in *BM_Seq* that is the next to be processed by the main loop of the Matrix Berlekamp/Massey algorithm. The main loop of the algorithm is executed by incrementing the *BM_iterator*. By implementing the algorithm in LinBox, the specialized field arithmetic of LinBox is available. A timing comparison of the Maple implementation versus the LinBox implementation (hopefully) shows the speed up due to the compiled specialized field arithmetic.

The future course of this research can follow many paths. A fraction free variant of the general Matrix Berlekamp/Massey algorithm is still unknown to us. A method for eliminating the nonsingularity requirements and circumventing singularities has thus far eluded us. A general fraction free Matrix Berlekamp/Massey algorithm could be found or a proof that one is not possible could also be another potential result. The LinBox implementation is another potential research project. LinBox is built on generic programming that the implementation does not fully take advantage of. Currently we only use one type of matrix in *BM_Seq* while LinBox has many matrix implementations. Also expanded constructor and insertion techniques patterned after similar STL list members can be added. Finally, the incorporation of faster linear algebra libraries such as BLAS is a potential project.

The quadratic Berlekamp/Massey algorithm and its generalizations are generally presented in the context of polynomial arithmetic. The research presented here explores these algorithms in the context of linear algebra. Further, the results show that the linear algebra context gives a more thorough explanation and illustration of how the Berlekamp/Massey algorithms operate. All of these algorithms, the scalar Berlekamp/Massey algorithm, the Matrix Berlekamp/Massey algorithm and the fraction free variations perform updates that maintain a basis of nullspace vectors of a well defined block Hankel matrix. The polynomial context just encapsulates these nullspace vectors in a compact representation that obfuscates the underlying justification of the update procedure. The connection between Berlekamp/Massey algorithms and block Hankel matrices is also vital. The dupli-

cation of the entries in the block Hankel matrices allows the algorithm to perform its degree updates at any time, in a similar manner to $\sigma$-basis computation. The underlying block Hankel matrix is encapsulated in the polynomial view by the convolution used to compute the discrepancies. Finally, unlike in many Hankel like matrix algorithms, the general Matrix Berlekamp/Massey algorithm can adjust for singularities in the submatrices of the block Hankel matrix. The linear algebra of Berlekamp/Massey algorithms for linearly generated sequences wins out over the polynomial arithmetic of the algorithms.

# Bibliography

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

[2] Beckermann, Bernhard, Cabay, Stanley, and Labahn, George. Fraction-free computation of matrix pade systems. In *International Symposium on Symbolic and Algebraic Computation*, pages 125–132, 1997. URL citeseer.ist.psu.edu/beckermann97fractionfree.html.

[3] Beckermann, B. and Labahn, G. A uniform approach for fast computation of matrix-type Padé approximants. *SIAM J. Matrix Anal. Applic.*, 15(3):804–823, July 1994.

[4] Beckermann, Bernhard and Labahn, George. Fraction-free computation of matrix rational interpolants and matrix GCDs. *SIAM J. Matrix Anal. Applic.*, 22(1):114–144, 2000.

[5] Berlekamp, E. R. *Algebraic Coding Theory.* McGraw-Hill Publ., New York, 1968.

[6] Brent, R. P., Gustavson, F. G., and Yun, D. Y. Y. Fast solution of Toeplitz systems of equations and computation of Padé approximants. *J. Algorithms*, 1:259–295, 1980.

[7] Brown, W. S. On Euclid's algorithm and the computation of polynomial greatest common divisors. *J. ACM*, 18:478–504, 1971.

[8] Brown, W. S. The subresultant PRS algorithm. *ACM Trans. Math. Software*, 4: 237–249, 1978.

[9] Brown, W. S. and Traub, J. F. On Euclid's algorithm and the theory of subresultants. *J. ACM*, 18:505–514, 1971.

[10] Cabay, S., Choi, D. K., and Labahn, G. Inverses of block Hankel and block Toeplitz matrices. *SIAM Journal of Computing*, 19:98–123, 1990.

[11] Collins, George E. Subresultants and reduced polynomial remainder sequences. *J. ACM*, 14:128–142, 1967.

[12] Coppersmith, D. Solving homogeneous linear equations over GF(2) via block Wiedemann algorithm. *Math. Comput.*, 62(205):333–350, 1994.

[13] Corless, Robert M., Jeffrey, David J., and Zhou, Wenqin. Fraction-free forms of LU and QR matrix factors. In *Proc. Transgressive Computing*, pages 443–446, Granada, 2006.

[14] Dickinson, Bradley W., Morf, Martin, and Kailath, Thomas. A minimal realization algorithm for matrix sequences. *IEEE Trans. Automatic Control*, AC-19(1):31–38, February 1974.

[15] Dornstetter, J. L. On the equivalence between Berlekamp's and Euclid's algorithms. *IEEE Trans. Inf. Theory*, IT-33(3):428–431, 1987.

[16] Dumas, J.-G., Gautier, T., Giesbrecht, M., Giorgi, P., Hovinen, B., Kaltofen, E., Saunders, B. D., Turner, W. J., and Villard, G. LinBox: A generic library for exact linear algebra. In Cohen, Arjeh M., Gao, Xiao-Shan, and Takayama, Nobuki, editors, *Proc. First Internat. Congress Math. Software ICMS 2002, Beijing, China*, pages 40–50, Singapore, 2002. World Scientific. ISBN 981-238-048-5. URL: EKbib/02/Detal02.pdf.

[17] Dumas, Jean-Guillaume, Giorgi, Pascal, and Pernet, Clément. FFPACK: Finite field linear algebra package. In Gutierrez [24], pages 119–126, 2004.

[18] Dummit, David S. and Foote, Richard M. *Abstract Algebra*. Prentice Hall, Inc., Englewood Cliffs, NJ 07632, 1991.

[19] Durbin, J. The fitting of time-series models. *Review of International Statistics Institute*, 28:229–249, 1959.

[20] Fatou, P. Séries trigonométriques et séries de Taylor. *Acta Mathematica*, 30(1):335–400, December 1906.

[21] Gamess, E., Musser, D. R., and Sánchez-Ruíz, A. J. Complete traversals and their implementation using the standard template library, 1997.

[22] Giesbrecht, Mark, Kaltofen, Erich, and Lee, Wen-shin. Algorithms for computing the sparsest shifts for polynomials via the Berlekamp/Massey algorithm. In *Proc. 2002 Internat. Symp. Symbolic Algebraic Comput. (ISSAC'02)*, pages 101–108, 2002.

[23] Giorgi, Pascal, Jeannerod, Claude-Pierre, and Villard, Gilles. On the complexity of polynomial matrix computations. In Sendra, J. R., editor, *Proc. 2003 Internat. Symp. Symbolic Algebraic Comput. (ISSAC'03)*, pages 135–142, New York, N. Y., 2003. ACM Press. ISBN 1-58113-641-2.

[24] Gutierrez, Jaime, editor. *ISSAC 2004 Proc. 2004 Internat. Symp. Symbolic Algebraic Comput.*, 2004. ACM Press. ISBN 1-58113-827-X.

[25] Hearn, Anthony C. An improved non-modular polynomial gcd algorithm. *SIGSAM Bull.*, (23):10–15, 1972. ISSN 0163-5824.

[26] Kaltofen, E. On computing determinants of matrices without divisions. In Wang, P. S., editor, *Proc. 1992 Internat. Symp. Symbolic Algebraic Comput. (ISSAC'92)*, pages 342–349, New York, N. Y., 1992. ACM Press. URL: EKbib/92/Ka92_issac.pdf.

[27] Kaltofen, E. Analysis of Coppersmith's block Wiedemann algorithm for the parallel solution of sparse linear systems. *Math. Comput.*, 64(210):777–806, 1995. URL: EKbib/95/Ka95_mathcomp.pdf.

[28] Kaltofen, Erich and Lee, Wen-shin. Early termination in sparse interpolation algorithms. *J. Symbolic Comput.*, 36(3–4):365–400, 2003. Special issue Internat. Symp. Symbolic Algebraic Comput. (ISSAC 2002). Guest editors: M. Giusti & L. M. Pardo. URL: EKbib/03/KL03.pdf.

[29] Kaltofen, E. and Saunders, B. D. On Wiedemann's method of solving sparse linear systems. In Mattson, H. F., Mora, T., and Rao, T. R. N., editors, *Proc. AAECC-9*, volume 539 of *Lect. Notes Comput. Sci.*, pages 29–38, Heidelberg, Germany, 1991. Springer Verlag. URL: EKbib/91/KaSa91.pdf.

[30] Kaltofen, Erich and Villard, Gilles. On the complexity of computing determinants. *Computational Complexity*, 13(3-4):91–130, 2004. URL: EKbib/04/KaVi04_2697263.pdf; Maple 7 worksheet URL: EKbib/04/KaVi04_2697263.mws.

[31] Levinson, N. The Wiener RMS (root-mean-square) error criterion in the filter design and prediction. *Journal of Mathematical Physics*, 25:261–278, 1947.

[32] Lobo, Austin A. *Matrix-Free Linear System Solving and Applications to Symbolic Computation*. PhD thesis, Rensselaer Polytechnic Instit., Troy, New York, December 1995.

[33] Massey, J. L. Shift-register synthesis and BCH decoding. *IEEE Trans. Inf. Theory*, IT-15:122–127, 1969.

[34] Musser, D. R. and Saini, A. *STL Tutorial and Reference Guide C++ Programming with the Standard Template Library*. Addison-Wesley Publ. Comp., Reading, Massachusetts, 1996.

[35] Nakos, George C., Turner, Peter R., and Williams, Robert M. Fraction-free algorithms for linear and polynomial equations. *SIGSAM Bull.*, 31(3):11–19, 1997. ISSN 0163-5824.

[36] Popov, V. M. Some properties of control systems with irreducible matrix transfer functions. In *Lecture Notes in Mathematics*, volume 144, pages 169–180. Springer Verlag, Berlin, 1970.

[37] Rissanen, J. Realizations of matrix sequences. Technical Report RJ-1032, IBM T. J. Watson Research Center, Yorktown Heights, New York, USA, 1972. As cited in [14].

[38] Saunders, David and Wan, Zhendong. Smith normal form of dense integer matrices, fast algorithms into practice. In Gutierrez [24], pages 274–281, 2004.

[39] Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley, third edition, 1997. ISBN 0-201-88954-4.

[40] Sugiyama, Y., Kasahara, M., Hirasawa, S., and Namekawa, T. A method for solving key equation for decoding Goppa codes. *Information & Control*, 27:87–99, 1975.

[41] Thomé, E. Subquadratic computation of vector generating polynomials and improvements of the block Wiedemann method. *J. Symbolic Comput.*, 33(5):757–775, May 2002.

[42] Turner, William J. *Black box linear algebra with the LINBOX library.* PhD thesis, North Carolina State Univ., Raleigh, North Carolina, August 2002. 193 pages.

[43] Van Barel, M. and Bultheel, A. The computation of non-perfect Padé-Hermite approximants. *Numerical Algorithms*, 1:285–304, 1991.

[44] Van Barel, M. and Bultheel, A. A general module theoretic framework for vector M-Padé and matrix rational interpolation. *Numerical Algorithms*, 3:451–462, 1992.

[45] Villard, G. Further analysis of Coppersmith's block Wiedemann algorithm for the solution of sparse linear systems. In Küchlin, W., editor, *ISSAC 97 Proc. 1997 Internat. Symp. Symbolic Algebraic Comput.*, pages 32–39, New York, N. Y., 1997. ACM Press. ISBN 0-89791-875-4.

[46] Villard, G. A study of Coppersmith's block Wiedemann algorithm using matrix polynomials. Rapport de Recherche 975 IM, Institut d'Informatique et de Mathématiques Appliquées de Grenoble, www.imag.fr, April 1997.

[47] Wiedemann, D. Solving sparse linear equations over finite fields. *IEEE Trans. Inf. Theory*, IT-32:54–62, 1986.

[48] Zhou, Wenqin and Jeffrey, David J. Fraction-free matrix factors: new forms for LU and QR factors. *Frontiers of Computer Science in China*, 2(1):67–80, 2008. ISSN 1673-7350.