A Displacement Approach to Decoding Algebraic Codes

Vadim Olshevsky* M. Amin Shokrollahi†

August 18, 2000

Abstract

Using methods originating in numerical analysis, we will develop a unified framework for derivation of efficient algorithms for decoding several classes of algebraic codes. We will demonstrate our method by accelerating Sudan's list decoding algorithm for Reed-Solomon codes [22], its generalization to algebraic-geometric codes by Shokrollahi and Wasserman [21], and the improvement of Guruswami and Sudan [9] in the case of Reed-Solomon codes.

The basic problem we attack in this paper is that of efficiently finding nonzero elements in the kernel of a *structured matrix*. The structure of such an $n \times n$- matrix allows it to be "compressed" to αn parameters for some α which is usually a constant in applications. The concept of structure is formalized using the displacement operator. It allows to perform matrix operations on the compressed version of the matrix. In particular, we can find a nontrivial element in the kernel of such a matrix in time $O(\alpha n^2)$, if it exists.

We will derive appropriate displacement operators for matrices that occur in the context of list decoding, and apply our general algorithm to them. For example, we will obtain algorithms that use $O(n^2\ell)$ and $O(n^{7/3}\ell)$ operations over the base field for list decoding of Reed-Solomon codes and algebraic-geometric codes from certain plane curves, respectively, where ℓ is the length of the list. Assuming that ℓ is constant, this gives algorithms of running time $O(n^2)$ and $O(n^{7/3})$, which is the same as the running time of conventional decoding algorithms. We will also derive efficient parallel algorithms for the above tasks.

1 Introduction

Matrices with different patterns of structure are often encountered in the context of coding theory. Examples include Hankel, Vandermonde, and Cauchy matrices which arise in the Berlekamp-Massey algorithm [3], Reed-Solomon codes, and classical Goppa codes [15], respectively. In most of these applications one is interested in a certain nonzero element in the kernel of the matrix. This problem

*Georgia State University
†Bell Labs

has been solved efficiently for each of the above cases. Although it is obvious that these algorithms make use of the structure of the underlying matrices, this exploitation is often rather implicit, and limited to the particular pattern of structure.

In this paper, we apply an alternative general method, called the method of displacement, for efficiently computing a nontrivial element in the kernel of a structured matrix. Though this method has been successfully used in other contexts such as image processing, system theory, or interpolation (see the surveys in [10, 13, 18]), its use in coding theory is novel and quite powerful, as we will see below. Its strength stems from the fact that it enables one to perform matrix operations on "compressed" versions of a structured matrix, rather than on the matrix itself. One of the main characteristics of a structured $n \times n$-matrix is that its n^2 entries are functions of only $O(n)$ parameters. In many situations, these functions are rather simple, so that it makes sense to say that the matrix can be compressed to $O(n)$ parameters.

This observation leads to algorithms with decreased running time. Before explaining this, we would like to formalize our model of computation. Here and in the sequel, an *operation* denotes one of the four fundamental arithmetic operations in the base field. The base field is often clear from the context, and is explicitly advertised should this not be the case. The *running time* of an algorithm is to be understood as the number of operations it performs.

Basic operations on matrices such as Gaussian elimination use $O(n^3)$ operations because they update the entries of the matrix $O(n)$ times. If we could modify this algorithm to run on the compressed version of the matrix, this could reduce the running time to something close to $O(n^2)$. As was realized by Morf [16, 17] and independently by Bitmead-Anderson [5], the displacement idea allows for a concise derivation of exactly such an algorithm (though limited at that time to a special "Toeplitz-like" structure). The details of the general algorithm that applies to all the above mentioned special structured matrices (Hankel, Vandermonde, Cauchy, etc.) will be carried out in the next section.

To demonstrate the power of our method, we will derive solutions to list-decoding problems concerning Reed-Solomon- and algebraic-geometric codes. Given a received word and an integer e, a list decoding algorithm returns a list of all codewords which have distance at most e from the received word. Building on a sequence of previous results [23, 4, 1], Sudan [22] was the first to invent an efficient list-decoding algorithm for Reed-Solomon-codes. This algorithm, its subsequent generalizations by Shokrollahi and Wasserman [21] to algebraic-geometric codes, and the recent extension by Guruswami and Sudan [9] are among the best decoding algorithms known in terms of the number of errors they can correct. All these algorithms run in two steps. The first step, which we call the *linear algebra* step, consists of computing a nonzero element in the kernel of a certain structured matrix. This element is then interpreted as a polynomial over an appropriate field; the second step, called the *root-finding step* then tries to find the roots of this polynomial over that field. The latter step is a subject of investigation of its own and can be solved very efficiently in many cases [2, 6, 20], so we will concentrate in this paper on the first step only. This will be done by applying our general algorithm in Sections 4 and 5. Specifically, we will for instance easily prove that the linear algebra step of decoding Reed-Solomon-codes of block length n with lists of

length ℓ can be accomplished in time $O(n^2\ell)$. A similar time bound holds also for the root-finding step, and so the overall running time of the list-decoding procedure is of this order of magnitude. This result matches that of Roth and Ruckenstein [20], though the latter has been obtained using completely different methods. Furthermore, we will design a novel $O(n^{7/3}\ell)$ algorithm for the linear algebra step of list decoding of certain algebraic-geometric-codes from plane curves of block length n with lists of length ℓ. We remark that, using other means, Høholdt and Refslund Nielsen [11] have obtained an algorithm for list decoding on Hermitian curves which solves both steps of the algorithm in [9] more efficiently.

In comparison to the existing algorithms, the method of displacement seems to have the advantage of leading to a transparent algorithm design. For instance, our method allows to design parallel algorithms for computing kernel elements for structured matrices. The design principles are the same as those of sequential algorithms, though the details are more complicated since we have to introduce new types of structure.

The paper is organized as follows. Sections 2 and 3 introduce the general framework of the displacement theory and their associated algorithm design. Sections 4, 5, and 6 apply the general framework to speed up Sudan's algorithm, its generalization by Shokrollahi-Wasserman, and the improvement of Guruswami-Sudan, respectively. The running times for these algorithms range from $O(\ell n^2)$ for Reed-Solomon codes to $O(n^{7/3}\ell)$ for AG-codes on curves from plane algebraic curves, where n is the block-length of the code and ℓ is the list-size. Section 7 introduces methods for the construction of efficient parallel algorithms for the above tasks.

2 The Displacement Structure

The main computational problem attacked in this paper is that of computing a nontrivial element in the kernel of a matrix V. We will, more precisely, distinguish between elements in the right-kernel of V, i.e., vectors x such that $Vx = 0$, and elements in the left-kernel of V, for which $xV = 0$. This computational task can be solved using Gaussian elimination. In fact, this procedure computes a PLU-decomposition for V; in other words, it computes a premutation matrix P, an invertible lower triangular matrix L, and an upper triangular matrix U such that $V = PLU$. We would like to give a succint presentation of this procedure, as this is crucial for the understanding of the algorithm we are going to present below.

Suppose that V is partitioned as $V = \left(\begin{smallmatrix} V_{11} & V_{12} \\ V_{21} & V_{22} \end{smallmatrix}\right)$, with square matrices V_{11} and V_{22}, and suppose further that V_{11} is invertible. The *Schur-complement* of V with respect to V_{11} is the matrix $V_2 := V_{22} - V_{21}V_{11}^{-2}V_{12}$, and we have the identity

$$\begin{pmatrix} V_{11} & V_{12} \\ V_{21} & V_{22} \end{pmatrix} = \begin{pmatrix} I & 0 \\ V_{21}V_{11}^{-1} & \tilde{I} \end{pmatrix} \begin{pmatrix} V_{11} & 0 \\ 0 & V_2 \end{pmatrix} \begin{pmatrix} I & V_{11}^{-1}V_{12} \\ 0 & \tilde{I} \end{pmatrix},$$

with identity matrices I and $\tilde{I}$ of appropriate sizes. This decomposition of V shows that if $V_{11} =: v$ is a 1×1-matrix, and if we know an LU-decomposition for V_2, say $V_2 = L_2U_2$, then we can compute

an LU decomposition for V as follows:

$$V = \underbrace{\begin{pmatrix} 1 & 0 \\ V_{21}/v & L_2 \end{pmatrix}}_{=:L} \cdot \underbrace{\begin{pmatrix} v & V_{21} \\ 0 & U_2 \end{pmatrix}}_{=:U}.$$

This is the basis for the Gaussian elimination algorithm, which iteratively computes Schur-complements and an LU-decomposition thereof. By using pivoting, the algorithm additionally produces a permutation matrix Q such that QV has the property that the $(1,1)$-entry of all the Schur-complements are nonzero, and hence the above procedure is applicable. Altogether, the algorithm produces a decomposition $V = PLU$, where $P = Q^{-1}$.

Much of the above discussion can be carried over to the case of structured matrices, and will lead to accelerated algorithms. To be precise, let m and n be positive integers, K be a field, and $D \in K^{m\times m}$, $A \in K^{n\times n}$. The structure of a matrix is formalized by the concept of the displacement rank: The *displacement operator* $\nabla = \nabla_{D,A}\colon K^{m\times n} \to K^{m\times n}$ is defined by $\nabla(V) := DV - VA$. The ∇-*displacement rank* of V is defined as the rank of the matrix $\nabla(V)$. If this rank is α, then $\nabla(V)$ can be written as GB with $G \in K^{m\times\alpha}$ and $B \in K^{\alpha\times n}$. The pair (G,B) is then called a ∇-*generator* for V. If α is small and D and A are sufficiently simple, then the ∇-operator allows to compress the matrix V to matrices with a total of $\alpha(m+n)$ entries. Furthermore, one can efficiently compute with the compressed form as the following lemma suggests.

Lemma 2.1. *Let the matrices in $DV - VA = GB$ be partitioned as*

$$G = (g_{11} \mid G_{21}),\ B = (b_{11} \mid B_{12}),\ D = \begin{pmatrix} d_1 & \mathbf{0} \\ \star & D_2 \end{pmatrix}, A = \begin{pmatrix} a_1 & \star \\ \mathbf{0} & A_2 \end{pmatrix}, V = \begin{pmatrix} v_{11} & V_{12} \\ V_{21} & V_{22} \end{pmatrix}.$$

Suppose that v_{11} is nonzero. Then the Schur complement $V_2 := V_{22} - v_{11}^{-1}V_{21}V_{12}$ of V satisfies the equation

$$D_2V_2 - V_2A_2 = G_2B_2,$$

where $G_2 = G_{21} - v_{11}^{-1}g_{11}V_{12}$ and $B_2 = B_{12} - v_{11}^{-1}b_{11}V_{12}$.

A proof can be found in [19, Lemma 3.1], see also [8]. If $v_{11} = 0$, we need to use *partial pivoting* [18, Sect. 3.5]. If V has a nonzero entry in its first column, say at position $(k,1)$, then we can consider PV instead of P, where P is the matrix corresponding to interchanging rows 1 and k. The displacement equation for PV is then given by $(PDP^\top)(PV) - PVA = PGB$. In order to apply the previous lemma, we need to ensure that $PDP^\top$ is lower triangular. But since P can be any transposition, this implies that D is a diagonal matrix.

It is possible to use the above lemma to design an algorithm for computing a PLU-decomposition of the matrix V, see [19]. Note that it is trivial to compute a nonzero element in the kernel of V once a PLU-decomposition is known, since the kernel of V and that of U conincide, and a nonzero element in the latter can be found using backward substitution.

Algorithm 2.2. *On input a diagonal matrix $D \in K^{m\times m}$, an upper triangular matrix $A \in K^{n\times n}$, and a $\nabla_{D,A}$-generator (G, B) for $V \in K^{m\times n}$ in $DV - VA = GB$, the algorithm outputs a permutation matrix P, a lower triangular matrix $L \in K^{m\times m}$, and an upper triangular matrix $U \in K^{m\times n}$, such that $V = PLU$.*

(1) *Recover from the generator the first column of V.*

(2) *Determine the position, say $(k, 1)$, of a nonzero entry of V. If it does not exist, then set the first column of L equal to $[1, \mathbf{0}]^\top$ and the first row of U equal to the first row of V, and go to Step (4). Otherwise interchange the first and the k-th diagonal entries of A and the first and the k-th rows of G and call P_1 the permutation matrix corresponding to this transposition.*

(3) *Recover from the generator the first row of $P_1V =: \left(\begin{smallmatrix} v_{11} & V_{12} \\ V_{21} & V_{22} \end{smallmatrix}\right)$. Store $[1, V_{12}/v_{11}]^\top$ as the first column of L and $[v_{11}, V_{12}]$ as the first row of U.*

(4) *If $v_{11} \neq 0$, compute by Lemma 2.1 a generator of the Schur complement V_2 of P_1V. If $v_{11} = 0$, then set $V_2 := V_{22}$, $G_2 := G_{21}$, and $B_2 := B_{12}$.*

(5) *Proceed recursively with V_2 which is now represented by its generator (G_2, B_2) to finally obtain the factorization $V = PLU$, where $P = P_1 \cdots P_\mu$ with P_k being the permutation used at the k-th step of the recursion and $\mu = \min\{m, n\}$.*

The correctness of the above algorithm and its running time depend on steps (1) and (3). Note that it may not be possible to recover the first row and column of V from the matrices D, A, G, B. We will explore these issues later in Section 7 when we deal with parallel algorithms.

For now, we focus on developing a more memory efficient version of this algorithm for certain displacement structures. For this, the following result will be crucial.

Lemma 2.3. *Let $V \in K^{m\times n}$ be partitioned as*

$$V = \begin{pmatrix} V_{11} & V_{12} \\ V_{21} & V_{22} \end{pmatrix},$$

for some $1 \leq i < m$, where $V_{11} \in K^{i\times i}$, $V_{12} \in K^{i\times(n-i)}$, $V_{21} \in K^{(m-i)\times i}$, and $V_{22} \in K^{(m-i)\times(m-i)}$, and suppose that V_{11} is invertible. Further, let $\tilde{V} := \left(\begin{smallmatrix} V \\ I_n \end{smallmatrix}\right)$, where I_n is the $n \times n$-identity matrix, and denote by $x = (x_1, \ldots, x_{m+n-i})^\top$ the first column of the Schur complement of $\tilde{V}$ with respect to V_{11}. Suppose that $x_1 = \cdots = x_{m-i} = 0$. Then the vector $(x_{m-i+1}, \ldots, x_m, 1, 0, \ldots, 0)^\top$ is in the right kernel of the matrix V.

Proof. The Schur complement of $\tilde{V}$ with respect to V_{11} is easily seen to be

$$\begin{pmatrix} V_{22} - V_{21}V_{11}^{-1}V_{12} \\ -V_{11}^{-1}V_{12} \\ I_{n-i} \end{pmatrix}.$$

Note that

$$V \cdot \begin{pmatrix} -V_{11}^{-1}V_{12} \\ I_{n-i} \end{pmatrix} = \begin{pmatrix} 0 \\ V_{22} - V_{11}^{-1}V_{12} \end{pmatrix}.$$

By assumption, the first column of the matrix on the right is zero, which implies the assertion. □

Suppose now that V has low displacement structure with respect to $\nabla_{D,Z}$ and suppose further that A is a matrix such that $A - Z$ has low rank:

$$DV - VZ = G_1 B_1, \quad A - Z = G_2 B_2.$$

Then we have

$$\begin{pmatrix} V & 0 \\ 0 & A \end{pmatrix} \cdot \begin{pmatrix} V \\ I_n \end{pmatrix} - \begin{pmatrix} V \\ I_n \end{pmatrix} Z = \begin{pmatrix} G_1 & 0 \\ 0 & G_2 \end{pmatrix} \cdot \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}. \tag{1}$$

Algorithm 2.2 can now be customized in the following way.

Algorithm 2.4. *On input a diagonal matrix $D \in K^{m\times m}$, an upper triangular matrix $Z \in K^{n\times n}$, a matrix $A \in K^{n\times n}$, and a $\nabla_{C,Z}$-generator (G, B) for $W = \binom{V}{I_n} \in K^{(m+n)\times n}$ in $CW - WZ = GB$, where $C = \begin{pmatrix} D & 0 \\ 0 & A \end{pmatrix}$, the algorithm outputs a vector $x = (x_1, \ldots, x_i, 1, 0, \ldots, 0)^\top$ such that $Vx = 0$.*

(0) *Set $i := 0$.*

(1) *Recover from the generators the first column of W.*

(2) *Determine the position, say $(k, 1)$, of a nonzero entry of W. If k does not exist, or $k > m - i$, then go to Step (6). Otherwise interchange the first and the k-th diagonal entries of D, the first and the k-th rows of G, the first and the k-th entries of the first column of W, denoting the new matrix by W again.*

(3) *Recover from the generators the first row of W.*

(4) *Using the first row and the first column of W, compute by Lemma 2.1 a generator of the Schur complement W_2 of W.*

(5) *Proceed recursively with V_2 which is now represented by its generator (G_2, B_2), increase i by 1, and go back to Step (1).*

(6) *Output the vector $(x_1, \ldots, x_i, 1, 0, \ldots, 0)^\top$ where $x_1, \ldots, x_i$ are the $m - i + 1$-st through m-th entries of the first column of W.*

As was pointed out before, the correctness of the above algorithm and its running time depend on steps (1) and (3). Note that it may not be possible to recover the first row and column of W from the matrices D, A, G, B, Z. In fact, recovery from these data alone is only possible if $\nabla = \nabla_{C,Z}$

is an isomorphism. For simplicity we assume in the following that this is the case. In the general case one has to augment the (D, A, G, B, Z) by more data corresponding to the kernel of ∇, see [18, Sect. 5].

If ∇ is an isomorphism, then the algorithm has the correct output. Indeed, using Lemma 2.1, we see that at step (5), W_2 is the Schur-complement of the matrix PW with respect to the prinicipal $i \times i$-minor, where P is a permutation matrix (obtained from the pivoting transpositions in Step (2)). Once the algorithm reaches Step (6), the conditions of Lemma 2.3 are satisfied and the algorithm outputs the correct vector.

The running time of the algorithm is summarized in the following.

Lemma 2.5. *Suppose that steps* (1) *and* (3) *of Algorithm 2.4 run in time* $O(\alpha m)$ *and* $O(\alpha n)$*, respectively. Then the total running time of that algorithm is* $O(\alpha mn)$*, where* α *is the displacement rank of* V *with respect to* $\nabla_{D,A}$*.*

PROOF. The proof is obvious once one realizes that Step (4) runs in time $O(\alpha(m+n))$, and that the algorithm is performed recursively at most $\min\{m, n\}$ times. □

3 Computing the First Row and the First Column

A major ingredient of Algorithm 2.4 is that of computing the first row and the first column of a matrix from its generators. The computational cost of this step depends greatly on the underlying displacement structure. In this section, we will present a solution to this problem in a special case, namely, in the case where the matrix $W \in K^{(m+n)\times n}$ has the displacement structure

$$\begin{pmatrix} D & 0 \\ 0 & A \end{pmatrix} W - WZ = GB \tag{2}$$

where D is diagonal and $Z, A \in K^{n\times n}$ are given by

$$Z := \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ 0 & 0 & 0 & \cdots & 0 & 0 \end{pmatrix}, \quad A := \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & 0 & \cdots & 0 & 0 \end{pmatrix}. \tag{3}$$

Z is called the *upper shift matrix* of format n. This case will be prototypical for our later applications. For the sake of simplicity, we will assume that all the diagonal entries of D are nonzero.

Algorithm 3.1. *On input a diagonal matrix* D *with diagonal entries* $x_1, \ldots, x_m$*, all of them nonzero, a matrix* $G = (G_{ij}) \in K^{m\times\alpha}$*, and a matrix* $B = (B_{ij}) \in K^{\alpha\times n}$*, this algorithm computes the first row and the first column of the matrix* $W \in K^{(m+n)\times n}$ *which satisfies the displacement equation (2).*

(1) *Compute the first row* $(r_1, \ldots, r_n)$ *and the first column* $(\gamma_1, \ldots, \gamma_m, \gamma_{m+1}, \ldots, \gamma_{m+n})^\top$ *of* GB.

(2) *For* $i = 1, \ldots, m$ *set* $v_i := \gamma_i / x_i$.

(3) *For* $i = m+1, \ldots, m+n-1$ *set* $v_{i+1} := \gamma_i$.

(4) *Set* $v_{m+1} := \gamma_{m+n}$.

(5) *Set* $c_1 := v_1$. *For* $i = 2, \ldots, n$ *set* $c_i := (r_i + c_{i-1})/x_i$.

(4) *Output row* $(c_1, \ldots, c_n)$ *and column* $(v_1, \ldots, v_m, v_{m+1}, \ldots, v_{m+n})^\top$.

Proposition 3.2. *The above algorithm correctly computes its output with* $O(\alpha(m+n))$ *operations in the field* K.

Proof. Correctness of the algorithm follows immediately from the following observation: let $(c_1, \ldots, c_n)$ and $(c_1, v_2, \ldots, v_m, v_{m+1}, \ldots, v_{m+n})^\top$ be the first row and the first column of W, respectively. Then

$$\begin{pmatrix} D & 0 \\ 0 & A \end{pmatrix} W - WZ = \begin{pmatrix} x_1c_1 & x_1c_2 - c_1 & \cdots & x_1c_n - c_{n-1} \\ x_2v_2 & \star & \cdots & \star \\ \vdots & \vdots & \cdots & \vdots \\ x_mv_m & \star & \cdots & \star \\ v_{m+2} & \star & \cdots & \star \\ \vdots & \vdots & \cdots & \vdots \\ v_{m+n} & \star & \cdots & \star \\ v_{m+1} & \star & \cdots & \star \end{pmatrix},$$

where the $\star$'s denote elements in K. As for the running time, observe first that computing the first row and first column of GB requires at most $\alpha(m+n) + \alpha n$ operations over K. Step (2) can be done using m operations, steps (3) and (4) are free of charge, and Step (5) requires $2n$ operations. □

We can now combine all the results obtained so far to develop an efficient algorithm for computing a nonzero element in the kernel of a matrix $V \in K^{m \times n}$ given indirectly as a solution to the displacement equation

$$DV - VZ = GB. \tag{4}$$

Here, D is a diagonal matrix, Z is an upper-shift matrix of format n, $G \in K^{m \times \alpha}$, and $B \in K^{\alpha \times n}$. Since we will use Algorithm 2.4, we need to have a displacement structure for $W = \binom{V}{I_n}$. It is given by

$$\begin{pmatrix} D & 0 \\ 0 & A \end{pmatrix} \binom{V}{I_n} - \binom{V}{I_n} Z = \binom{G}{C} \cdot B, \tag{5}$$

where A is defined as in (3) and $C = A - Z$. We assume that none of the diagonal entries of D are zero.

Algorithm 3.3. *On input integers m, n, $m < n$, a diagonal matrix $D \in K^{m\times m}$ all of whose diagonal entries $x_1, \ldots, x_m$ are nonzero, a matrix $G \in K^{m\times\alpha}$, and a matrix $B \in K^{\alpha\times n}$, the algorithm computes a nonzero vector $v \in K^n$ in the kernel of the matrix V given by the displacement equation (4).*

(1) *Set $G_1 := \binom{G}{A-Z}$, $B_1 := B$, where A, G, and B are given in (3), (7), and (8) respectively.*

(2) *For $i = 1, \ldots, m$ do*

(a) *Let D_i be the diagonal matrix with diagonal entries $x_i, \ldots, x_m$. Use Algorithm 3.1 with input D_i, G_i, and B_i to compute the column $c = (c_1, \ldots, c_{m+n-i})^\top$.*

(b) *If $c_1 = \cdots = c_{m-i} = 0$, then* **output** $(c_{m-i+1}, \ldots, c_n, 1, 0, \ldots, 0)^\top$ *and* **stop**. *Otherwise, find an integer $k \le m - i$ such that $c_k \neq 0$. Interchange rows k and 1 of c and of G_i, interchange the first and the kth diagonal entries of D_i.*

(c) *Use Algorithm 3.1 with input D_i, G_i and B_i to compute the row $r = (r_1, \ldots, r_{n-i})$.*

(d) *For $j = 2, \ldots, m + n - i$ replace row j of G_i by $-c_j/c_1$ times the first row plus the j-th row. Set G_{i+1} as the matrix formed from G_i by deleting the first row.*

(e) *For $j = 2, \ldots, n - i$ replace the j-th column of B_i by $-r_j/c_1$ times the first column plus the j-th column. Set B_{i+1} as the matrix formed from B_i by deleting the first column.*

Theorem 3.4. *The above algorithm correctly computes its output with $O(\alpha mn)$ operations over the field K.*

Proof. The correctness of the algorithm follows from Lemma 2.3 and Proposition 3.2. The analysis of the running time of the algorithm is straight-forward: Step (2a) runs in time $O(\alpha(m + n - i))$ by Proposition 3.2. The same is true for steps (2d) and (2e). Hence, in the worst case, the running time will be $\sum_{i=1}^{n} O(\alpha(m + n - i)) = O(\alpha mn)$, since $m < n$. □

The memory requirements for this algorithm can be reduced by observing that the last $n-i$ rows of the matrix G_i are always known. As a result, the matrix G_i needs to store only $m(\ell+1)$ elements (instead of $(m+n)(\ell+1)$). Further, it is easy to see that for the vector c computed in Step (2a), we have $c_{m+1} = 1, c_{m+2} = \cdots = c_{m+n-i} = 0$, hence, we do not need to store these results. Combining these observations, one can easily design an algorithm that needs storage for two matrices of sizes $m \times (\ell + 1)$ and $(\ell + 1) \times n$, respectively, and storage for three vectors of size n.

4 The Algorithm of Sudan

In [22] Sudan describes an algorithm for list decoding of RS-codes which we will briefly describe here. Let $\mathbb{F}_q[x]_{<k}$ denote the space of polynomials over $\mathbb{F}_q$ of degree less than k, and let $x_1, \ldots, x_n$ denote distinct elements of $\mathbb{F}_q$, where $k \le n$. The image of the morphism $\gamma\colon \mathbb{F}_q[x]_{<k} \to \mathbb{F}_q^n$ mapping a polynomial f to the vector $v := (f(x_1), \ldots, f(x_n))$ is a linear code over $\mathbb{F}_q$ of dimension k and

minimum distance $n-k+1$. Suppose the vector v is sent over a communication channel and the vector $u := (y_1, \ldots, y_n)$ is received. If the Hamming distance between u and v is at most $(n-k)/2$, then conventional decoding algorithms like the Berlekamp-Massey algorithm can decode u to the correct codeword v. If the number e of errors is larger than $(n-k)/2$, then Sudan's algorithm compiles a list of at most $\ell = \ell(e)$ codewords which contains v. The algorithm consists of two steps. Let $b := \lfloor n/(\ell+1) + \ell(k-1)/2 + 1 \rfloor$. The first step uses Lagrange interpolation to compute a bivariate polynomial $F = \sum_{i=0}^{\ell} F_i(x) y^i$ with $\deg F_i < b - i(k-1)$ and such that $F(x_t, y_t) = 0$ for all $t = 1, \ldots, n$. The second step computes the roots of F which are of the form $y - g(x)$, $g \in \mathbb{F}_q[x]_{<k}$, and outputs those g such that $\gamma(g)$ and u have distance at most e. The relationship between ℓ and e is given by $e \leq n - b$. There are efficient algorithms for solving the second step using Hensel lifting [2, 6, 20]. In the following we will concentrate on the problem of efficiently computing the polynomial F.

This polynomial corresponds to a nonzero element in the kernel of a matrix V with a repetitive structure which we will describe below. Let $d_0, \ldots, d_\ell \geq 0$ be defined by $d_i := b - i(k-1)$ for $0 \leq i < \ell$, and $d_\ell = n + 1 - \sum_{i=0}^{\ell-1} d_i$. Let

$$V := \begin{pmatrix} 1 & x_1 & \cdots & x_1^{d_0-1} & y_1 & y_1 x_1 & \cdots & y_1 x_1^{d_1-1} & \cdots & y_1^\ell & y_1^\ell x_1 & \cdots & y_1^\ell x_1^{d_\ell-1} \\ 1 & x_2 & \cdots & x_2^{d_0-1} & y_2 & y_2 x_2 & \cdots & y_2 x_2^{d_1-1} & \cdots & y_2^\ell & y_2^\ell x_2 & \cdots & y_2^\ell x_2^{d_\ell-1} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & \cdots & x_n^{d_0-1} & y_n & y_n x_n & \cdots & y_n x_n^{d_1-1} & \cdots & y_n^\ell & y_n^\ell x_n & \cdots & y_n^\ell x_n^{d_\ell-1} \end{pmatrix}, \quad (6)$$

and let $v = (v_{00}, v_{01}, \ldots, v_{0,d_0-1}, \ldots, v_{\ell,0}, v_{\ell,1}, \ldots, v_{\ell,d_\ell-1}) \in \mathbb{F}_q^{n+1}$ be a nonzero vector such that $V \cdot v = 0$. Then the polynomial $F(x,y) = \sum_{i=0}^{\ell} F_i(x) y^i$ where $F_i(x) = v_{i,0} + v_{i,1} x + \cdots + v_{i,d_i-1} x^{d_i-1}$ satisfies $F(x_t, y_t) = 0$ for $t = 1, \ldots, n$. The task at hand is thus to compute a nonzero element in the kernel of V. Using the displacement approach we can easily compute such a vector in time $O(n^2 \ell)$. For constant list-size ℓ we thus obtain an algorithm whose running time is quadratic in n.

Let us first prove that V has displacement rank at most $\ell + 1$ with respect to suitable matrices. Let D be the diagonal matrix with diagonal entries $1/x_1, \ldots, 1/x_n$. Define

$$G := \begin{pmatrix} 1/x_1 & y_1/x_1 - x_1^{d_0-1} & \cdots & y_1^{\ell-1}(y_1/x_1 - x_1^{d_{\ell-1}-1}) \\ 1/x_2 & y_2/x_2 - x_2^{d_0-1} & \cdots & y_2^{r-2}(y_2/x_2 - x_2^{d_{\ell-1}-1}) \\ \vdots & \vdots & \ddots & \vdots \\ 1/x_n & y_n/x_n - x_n^{d_0-1} & \cdots & y_n^{r-2}(y_n/x_n - x_n^{d_{\ell-1}-1}) \end{pmatrix}. \quad (7)$$

and

$$B := \left(\begin{array}{ccccc|ccccc|c|ccccc|ccccc} 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 & 0 & 0 & \cdots & 0 & & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \cdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 & & 0 & 0 & 0 & \cdots & 0 & 1 & 0 & 0 & \cdots & 0 \\ \underbrace{\qquad\qquad\qquad\qquad\qquad}_{d_0} & & & & & \underbrace{\qquad\qquad\qquad\qquad\qquad}_{d_1} & & & & & & \underbrace{\qquad\qquad\qquad\qquad\qquad}_{d_{\ell-1}} & & & & & \underbrace{\qquad\qquad\qquad\qquad\qquad}_{d_\ell} & & & & \end{array} \right). \tag{8}$$

Then, one easily proves that

$$DV - VZ = GB,$$

where Z is the upper shift matrix of format $n+1$ defined in (3). Hence, V has displacement rank at most $\ell+1$ with respect to $\nabla_{D,Z}$, and we can use Algorithm 3.3 to compute a nonzero element in the kernel of V.

Algorithm 4.1. *On input $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbb{F}_q^2$ and integers $d_0, d_1, \ldots, d_\ell$ such that $\sum_{i=0}^{\ell} d_i = n+1$ and such that none of the x_i is zero, this algorithm computes a a polynomial $F(x,y) = \sum_{i=0}^{\ell} F_i(x) y^i$ such that $F(x_i, y_i) = 0$ for $i = 1, \ldots, n$.*

(1) *Run Algorithm 3.3 on input D, G, B, where D is the diagonal matrix with entries $1/x_1, \ldots, 1/x_n$, and G, B given in (7) and (8), respectively. Let $v = (v_{00}, v_{01}, \ldots, v_{0,d_0-1}, \ldots, v_{\ell,0}, v_{\ell,1}, \ldots, v_{\ell,d_\ell-1})$ denote the output.*

(2) *Output $F(x,y) = \sum_{i=0}^{\ell} F_i(x) y^i$ where $F_i(x) = v_{i,0} + v_{i,1}x + \cdots + v_{i,d_i-1}x^{d_i-1}$.*

Theorem 3.4 immediately implies the following result.

Theorem 4.2. *Algorithm 4.1 correctly computes its output with $O(n^2\ell)$ operations over the field K.*

There are various methods to extend the algorithm to the case where one of the x_i is zero. We will sketch one of these methods. Without loss of generality, assume that $x_1 = 0$. In this case, deletion of the first row of the matrix V yields another matrix $\tilde{V}$ of the form given in (6) in which none of the x_i is zero. The matrix V has then the following displacement structure

$$\begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1/x_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1/x_n \end{pmatrix} V - VZ = \begin{pmatrix} 0_{1\times(\ell+1)} & 1 \\ G & 0_{(n-1)\times 1} \end{pmatrix} \cdot \begin{pmatrix} B \\ x \end{pmatrix},$$

where x is the vector

$$x = \Big(\underbrace{1, -1, 0, \ldots, 0}_{d_0}, \underbrace{y_1, -y_1, 0, \ldots, 0}_{d_1}, \cdots, \underbrace{y_1^\ell, -y_1^\ell, 0, \ldots, 0}_{d_\ell} \Big),$$

G is the matrix obtained from the one given in (7) by deleting the first row, and $0_{a\times b}$ denotes the $a \times b$-matrix consisting of zeros. We can now run Algorithm 3.3 on the new set of matrices to obtain a nonzero element in the kernel of V.

We finish this section with an example. Suppose that C is the Reed-Solomon code of dimension 4 given as the set of evaluations of polynomials of degree at most 3 over the field $\mathbb{F}_{31}$ at the points $x_1 = 1, x_2 = 2, \ldots, x_{30} = 30$. This code can correct up to 15 errors with lists of size at most 2. In this case, $\ell = 2$, $n = 30$, $k = 4$, $b = \lfloor n/(\ell+1) + \ell k/2 + 1 \rfloor = 14$, $d_0 = 14$, $d_1 = 11$, and $d_2 = 8$.

Suppose that we have received the vector

$$y = (3, 13, 0, 6, 7, 24, 19, 25, 1, 17, 19, 5, 10, 0, 19, 2, 4, 23, 28, 23, 29, 7, 8, 12, 27, 24, 15, 6, 22, 30)$$

Then the matrices G and B given in (7) and (8) are as follows:

$$G = \begin{pmatrix} 1 & 16 & 21 & 8 & 25 & 26 & 9 & 4 & 7 & 28 & 17 & 13 & 12 & 20 & 29 & 2 & 11 & 19 & 18 & 14 & 3 & 24 & 27 & 22 & 5 & 6 & 23 & 10 & 15 & 30 \\ 2 & 14 & 7 & 15 & 15 & 29 & 28 & 22 & 20 & 2 & 23 & 17 & 16 & 3 & 28 & 0 & 10 & 14 & 25 & 2 & 3 & 0 & 15 & 4 & 17 & 25 & 6 & 22 & 28 & 2 \\ 6 & 25 & 0 & 3 & 12 & 23 & 15 & 26 & 2 & 9 & 28 & 14 & 3 & 0 & 3 & 6 & 14 & 16 & 20 & 6 & 22 & 25 & 15 & 16 & 25 & 19 & 14 & 24 & 15 & 0 \end{pmatrix}^\top,$$

$$B = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

The matrix D of Algorithm 4.1 is the diagonal matrix with diagonal entries

$$(1, 16, 21, 8, 25, 26, 9, 4, 7, 28, 17, 13, 12, 20, 29, 2, 11, 19, 18, 14, 3, 24, 27, 22, 5, 6, 23, 10, 15, 30).$$

In the first round of Algorithm 3.3, the first column computed in step 2(a) is

$$(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)^\top.$$

The first row, computed at step 2(c), is

$$(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 9, 9, 9, 9, 9, 9).$$

The loop in step 2 of Algorithm 3.3 is performed 29 times. The kernel element computed at the end equals

$$(12, 5, 4, 9, 20, 19, 28, 26, 23, 18, 11, 16, 14, 25, 26, 15, 14, 25, 1, 25, 29, 27, 9, 17, 6, 24, 30, 7, 20, 1, 0).$$

This element corresponds to the bivariate polynomial

$$\begin{aligned} F(x,y) = {} & 12 + 5\,x + 4\,x^2 + 9\,x^3 + 20\,x^4 + 19\,x^5 + 28\,x^6 + 26\,x^7 + 23\,x^8 + 18\,x^9 + 11\,x^{10} \\ & + 16\,x^{11} + 14\,x^{12} + 25\,x^{13} + \\ & \left(26 + 15\,x + 14\,x^2 + 25\,x^3 + x^4 + 25\,x^5 + 29\,x^6 + 27\,x^7 + 9\,x^8 + 17\,x^9 + 6\,x^{10}\right) y + \\ & \left(24 + 30\,x + 7\,x^2 + 20\,x^3 + x^4\right) y^2. \end{aligned}$$

Applying the algorithm of, e.g., [6], we obtain

$$F(x,y) \equiv \Big(y-(1+x+x^3)\Big)\Big((y-16)+(9y-1)x+(6-y)x^2+(29+6y)x^3+(22y+28)x^4\Big) \bmod x^5,$$

which shows that there is at most one polynomial, $f(x) = 1+x+x^3$, which satisfies $F(x, f(x)) = 0$. A further calculation shows that this is indeed the case. Hence, there is only one codeword in C of distance at most 15 from the received word y, namely the codeword corresponding to f.

5 The Algorithm of Shokrollahi-Wasserman

In [21] the authors generalize Sudan's algorithm [22] to algebraic-geometric (AG-) codes. We briefly discuss this generalization. Let $\mathcal{X}$ be an irreducible algebraic curve over the finite field $\mathbb{F}_q$, let $Q, P_1, \ldots, P_n$ be distinct $\mathbb{F}_q$-rational points of $\mathcal{X}$, and let $L(\alpha Q)$ denote the linear space of the divisor αQ, i.e., the space of all functions in the function field of $\mathcal{X}$ that have only a pole of order at most α at Q. The (one-point) AG-code associated to these data is then defined as the image of the $\mathbb{F}_q$-linear map $L(\alpha Q) \to \mathbb{F}_q^n$ mapping a function f to the vector $(f(P_1), \ldots, f(P_n))$. It is well known that this linear code has dimension $k \geq \alpha - g + 1$ and minimum distance $d \geq n - \alpha$, where g denotes the genus of the curve. Suppose that we want to decode this code with a list of length at most ℓ. Let $\beta := \lfloor (n+1)/(\ell+1) + \ell\alpha/2 + g \rfloor$. Let $\varphi_1, \ldots, \varphi_t$, $t = \beta - g + 1$, be elements of $L(\beta Q)$ with strictly increasing pole orders at Q. Let $(y_1, \ldots, y_n)$ be the received word. The algorithm in [21] first finds functions $u_0, \ldots, u_\ell$ with $u_i \in L((\beta - i\alpha)Q)$ such that $\sum_i u_i(P_j) y_j^i = 0$ for all $1 \leq j \leq n$. This step is accomplished by computing a nonzero element in the kernel of the matrix

$$V := \left(\begin{array}{ccc|ccc|c|ccc} \varphi_1(P_1) & \cdots & \varphi_{s_0}(P_1) & y_1\varphi_1(P_1) & \cdots & y_1\varphi_{s_1}(P_1) & \cdots & y_1^\ell\varphi_1(P_1) & \cdots & y_1^\ell\varphi_{s_\ell}(P_1) \\ \varphi_1(P_2) & \cdots & \varphi_{s_0}(P_2) & y_2\varphi_1(P_2) & \cdots & y_2\varphi_{s_1}(P_2) & \cdots & y_2^\ell\varphi_1(P_2) & \cdots & y_2^\ell\varphi_{s_\ell}(P_2) \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \varphi_1(P_n) & \cdots & \varphi_{s_0}(P_n) & y_n\varphi_1(P_n) & \cdots & y_n\varphi_{s_1}(P_n) & \cdots & y_n^\ell\varphi_1(P_n) & \cdots & y_n^\ell\varphi_{s_\ell}(P_n) \end{array}\right), \tag{9}$$

where $s_j = \beta - j\alpha - g + 1$ for $j < \ell$ and $\sum_{i=0}^{\ell}(s_j + 1) = n + 1$. To simplify the discussions, we assume that there are two functions $\varphi, \psi \in L(\beta Q)$ such that all the φ_i are of the form $\varphi^a\psi^b$ and such that the order of poles at Q of φ is smaller than that of ψ. Further, we assume that φ does not vanish at any of the points $P_1, \ldots, P_n$. We remark that the method described below can be modified to deal with a situation in which any of the above assumptions is not valid.

Let d be the order of poles of φ at Q. Then it is easily seen that any element of $L(\beta Q)$ is of the form $\varphi^a\psi^b$, where $0 \leq b < d$. Now we divide each of the blocks of the matrix V into subblocks in

the following way: by changing the order of the columns, we write the t-th block of V in the form

$$\left(\begin{array}{ccc|c|ccc} \varphi(P_1)^0\psi(P_1)^0y_1^t & \cdots & \varphi(P_1)^{a_0}\psi(P_1)^0y_1^t & \cdots & \varphi(P_1)^0\psi(P_1)^sy_1^t & \cdots & \varphi(P_1)^{a_s}\psi(P_1)^sy_1^t \\ \varphi(P_2)^0\psi(P_2)^0y_2^t & \cdots & \varphi(P_2)^{a_0}\psi(P_2)^0y_2^t & \cdots & \varphi(P_2)^0\psi(P_2)^sy_2^t & \cdots & \varphi(P_2)^{a_s}\psi(P_2)^sy_2^t \\ \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ \varphi(P_n)^0\psi(P_n)^0y_n^t & \cdots & \varphi(P_n)^{a_0}\psi(P_n)^0y_n^t & \cdots & \varphi(P_n)^0\psi(P_n)^sy_n^t & \cdots & \varphi(P_n)^{a_s}\psi(P_n)^sy_n^t \end{array}\right).$$

By the above remarks, $s < d$. Let now D be the diagonal matrix $\mathrm{diag}[\varphi]_{1,n}$, and let A be the upper shift matrix of format $n+1$. Then in a similar way as in the last section one can construct a matrix $G \in \mathbb{F}_q^{n\times d\ell}$ and a matrix $B \in \mathbb{F}_q^{d\ell\times m}$ such that $DV - VZ = GB$, where Z is the upper shift matrix of format $n+1$. Using Algorithm 3.3 we then obtain the following result.

Theorem 5.1. *Using Algorithm 3.3, one can compute a nonzero element in the kernel of the matrix V of (9) with $O(n^2d\ell)$ operations over the base field.*

A final estimate of the running time of this algorithm depends on the relationship between d and n. This depends on the specific structure of the curve and the divisor used in the definition. For instance, if the curve has a nonsingular plane model of the form $G(x, y) = 0$, and G is a monic polynomial of degree d in the variable y, and if we choose for Q the common pole of the functions x and y, then the above conditions are all valid (see also [12]). As an example, consider the case of a Hermitian curve defined over $\mathbb{F}_{q^2}$ by the equation $x^{q+1} = y^q + y$. In this case, the parameter d equals $q + 1$, which $n = q^3$. As a result, the algorithm uses $O(n^{7/3}\ell)$ $\mathbb{F}_q$-operations. Note that for $\ell = 1$, i.e., for unique decoding, this is exactly the running time of the algorithm presented in [12].

6 The Guruswami-Sudan Algorithm

In [9] the authors describe an extension of algorithms presented in [22] and [21] in which they use a variant of Hermite interpolation rather than a Lagrange interpolation. In the case of Reed-Solomon codes, the input to the algorithm is a set of n points (x_i, y_i) in the affine plane over $\mathbb{F}_q$, and parameters r, k, and ℓ. Let β be the smallest integer such that $(\ell+1)\beta > \binom{\ell+1}{2}k + \binom{r+1}{2}n$. The output of the algorithm is a nonzero bivariate polynomial $G = \sum_{i=0}^{\ell} G_i(x)y^i$ such that $\deg(G_i) < \beta - i(k-1)$, $G(x_i, y_i) = 0$ for all $i \le n$, and $G(x + x_i, y + y_i)$ does not contain any monomial of degree less than r. Such a polynomial G corresponds to a nonzero element in the kernel of a certain matrix V which we will describe below. Let $t \le n$ be a fixed positive integer. For $0 \le i < r$ and $0 \le j \le \ell$ let $V_{ij}^t \in \mathbb{F}_q^{(r-i)\times(\beta-j(k-1))}$ be the matrix having (μ, ν)-entry equal to $\binom{\nu}{\mu}x_t^{\nu-\mu}$, where $0 \le \mu < r - i$ and $0 \le \nu < \beta - j(k-1)$. Now define the matrix V_t as a block matrix with r block rows and $\ell+1$ block

columns, with block entry (i,j) equal to $\binom{j}{i} y_t^{j-i} V_{ij}^t$. The matrix V then equals

$$V = \begin{pmatrix} V_1 \\ V_2 \\ \vdots \\ V_n \end{pmatrix}. \tag{10}$$

V has $m := \binom{r+1}{2} n$ rows and $s := (\ell+1)\beta - \binom{\ell+1}{2}(k-1)$ columns (and $s > m$). In the following we will show that V has a displacement structure with respect to suitable matrices.

Let J_i^t denote the $i \times i$-Jordan block having x_t in its main diagonal and 1's on its lower subdiagonal. Let J^t be the block diagonal matrix with block diagonal entries $J_r^t, \ldots, J_1^t$. Let J be the block diagonal matrix with block diagonal entries $J^1, \ldots, J^n$:

$$J_i^t := \underbrace{\begin{pmatrix} x_t & 0 & 0 & \cdots & 0 & 0 \\ 1 & x_t & 0 & \cdots & 0 & 0 \\ 0 & 1 & x_t & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & x_t \end{pmatrix}}_{i}, \quad J^t := \underbrace{\begin{pmatrix} J_r^t & 0 & \cdots & 0 \\ 0 & J_{r-1}^t & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & J_1^t \end{pmatrix}}_{\binom{r+1}{2}}, \quad J := \underbrace{\begin{pmatrix} J^1 & 0 & \cdots & 0 \\ 0 & J^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & J^n \end{pmatrix}}_{n\binom{r+1}{2}}. \tag{11}$$

Then, a quick calculation shows that

$$JV - VZ_s^\top = HU \tag{12}$$

where Z_s is the upper shift matrix of format s, and H and U are described as follows: for $1 \le t \le n$

let

$$H_t := \left(\begin{array}{ccccc}
b_{0,0}^{d_0,0} - b_{1,0}^{0,0} & b_{1,0}^{d_1,0} - b_{2,0}^{0,0} & \cdots & b_{\ell-1,0}^{d_{\ell-1},0} - b_{\ell,0}^{0,0} & b_{\ell,0}^{d_\ell,0} \\
b_{0,0}^{d_0,1} - b_{1,0}^{0,1} & b_{1,0}^{d_1,1} - b_{2,0}^{0,1} & \cdots & b_{\ell-1,0}^{d_{\ell-1},1} - b_{\ell,0}^{0,1} & b_{\ell,0}^{d_\ell,1} \\
\vdots & \vdots & \cdots & \vdots & \vdots \\
b_{0,0}^{d_0,r-1} - b_{1,0}^{0,r-1} & b_{1,0}^{d_1,r-1} - b_{2,0}^{0,r-1} & \cdots & b_{\ell-1,0}^{d_{\ell-1},r-1} - b_{\ell,0}^{0,r-1} & b_{\ell,0}^{d_\ell,r-1} \\
\hline
b_{0,1}^{d_0,0} - b_{1,1}^{0,0} & b_{1,1}^{d_1,0} - b_{2,1}^{0,0} & \cdots & b_{\ell-1,1}^{d_{\ell-1},0} - b_{\ell,1}^{0,0} & b_{\ell,1}^{d_\ell,0} \\
b_{0,1}^{d_0,1} - b_{1,1}^{0,1} & b_{1,1}^{d_1,1} - b_{2,1}^{0,1} & \cdots & b_{\ell-1,1}^{d_{\ell-1},1} - b_{\ell,1}^{0,1} & b_{\ell,1}^{d_\ell,1} \\
\vdots & \vdots & \cdots & \vdots & \vdots \\
b_{0,1}^{d_0,r-2} - b_{1,1}^{0,r-2} & b_{1,1}^{d_1,r-2} - b_{2,1}^{0,r-2} & \cdots & b_{\ell-1,1}^{d_{\ell-1},r-2} - b_{\ell,1}^{0,r-2} & b_{\ell,1}^{d_\ell,r-2} \\
\hline
\vdots & \vdots & \ddots & \vdots & \vdots \\
\hline
b_{0,r-11}^{d_0,0} - b_{1,r-1}^{0,0} & b_{1,r-1}^{d_1,0} - b_{2,r-1}^{0,0} & \cdots & b_{\ell-1,r-1}^{d_{\ell-1},0} - b_{\ell,r-1}^{0,0} & b_{\ell,r-11}^{d_\ell,0}
\end{array}\right) \in \mathbb{F}_q^{\binom{r+1}{2}\times(\ell+1)}, \quad (13)$$

where $b_{c,d}^{e,f} := \binom{c}{d}\binom{e}{f} y_t^{c-d} x_t^{e-f}$. Then

$$H = \begin{pmatrix} H_1 \\ H_2 \\ \vdots \\ H_n \end{pmatrix}, \quad (14)$$

and

$$U := \begin{pmatrix} \underbrace{\begin{matrix} 0 & 0 & 0 & \cdots & 1 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \end{matrix}}_{d_0} & \underbrace{\begin{matrix} 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 1 \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \end{matrix}}_{d_1} & \cdots & \underbrace{\begin{matrix} 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ 0 & 0 & 0 & \cdots & 0 \end{matrix}}_{d_{\ell-1}} & \underbrace{\begin{matrix} 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 1 \end{matrix}}_{d_\ell} \end{pmatrix}. \tag{15}$$

Unfortunately, we cannot apply our general results of Section 2 to this situation directly, since J is not a diagonal matrix and $Z_s^\top$ is not upper triangular. To remedy the situation we proceed as follows. Let $\mathbb{F}$ be a suitable extension of $\mathbb{F}_q$ which contains s nonzero pairwise distinct elements $\epsilon_1, \ldots, \epsilon_s$. Let W be the $s \times s$-Vandermonde matrix having (i,j)-entry ϵ_i^{j-1} and let Δ be the diagonal matrix with diagonal entries $1/\epsilon_1, \ldots, 1/\epsilon_s$. Then we have the following displacement structure for W

$$\Delta W - W Z = EF, \tag{16}$$

where $E = (1/\epsilon_1, 1/\epsilon_2, \ldots, 1/\epsilon_s)^\top$ and $F = (1, 0, \ldots, 0)$. Transposing both sides of (12) we obtain the equation $Z_s V^\top - V^\top J^\top = -F^\top E^\top$. Multiplying both sides of this equation with W from the left, multiplying both sides of (16) with $V^\top$ from the right, and adding the equations gives the displacement equation for the matrix $WV^\top$:

$$\Delta W V^\top - W V^\top J^\top = E F V^\top - W U^\top H^\top = \left(E \mid -WU^\top \right) \cdot \begin{pmatrix} FV^\top \\ H^\top \end{pmatrix} =: GB. \tag{17}$$

Written out explicitly, we have

$$G = \begin{pmatrix} \epsilon_1^{D_0-1} & -\epsilon_1^{D_1-1} & -\epsilon_1^{D_2-1} & \cdots & -\epsilon_1^{D_\ell-1} \\ \epsilon_2^{D_0-1} & -\epsilon_2^{D_1-1} & -\epsilon_2^{D_2-1} & \cdots & -\epsilon_2^{D_\ell-1} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ \epsilon_s^{D_0-1} & -\epsilon_s^{D_1-1} & -\epsilon_s^{D_2-1} & \cdots & -\epsilon_s^{D_\ell-1} \end{pmatrix}, \tag{18}$$

$$B = \begin{pmatrix} \overbrace{1 \;\; 0 \;\; \cdots \;\; 0}^{\binom{r+1}{2}} & \overbrace{1 \;\; 0 \;\; \cdots \;\; 0}^{\binom{r+1}{2}} & \cdots & \overbrace{1 \;\; 0 \;\; \cdots \;\; 0}^{\binom{r+1}{2}} \\ H_1^\top & H_2^\top & \cdots & H_n^\top \end{pmatrix}, \tag{19, 20}$$

where $D_0 := 0$, and $D_i := d_0 + \cdots + d_{i-1}$ for $i > 0$, and H_i as defined in (13).

The matrix $WV^\top$ has the right displacement structure, since Δ is diagonal and $J^\top$ is upper triangular. However, since we are interested in an element in the right kernel of V, we need to compute an element in the *left* kernel of $WV^\top$. This computation can be done using similar techniques as above by considering an appropriate displacement structure for the matrix $(WV^\top \mid I)$. However, certain complications arise in this case due to the fact that the displacement operator for this matrix which maintains a low displacement rank is not an isomorphism. Therefore, we defer the design of that algorithm to the appendix. Using that procedure, we obtain the following overall algorithm.

Algorithm 6.1. *On input $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbb{F}_q^2$ where none of the x_i is zero, a positive integer r, and integers $d_0, d_1, \ldots, d_\ell$ such that $\sum_{i=0}^{\ell}(d_i + 1) =: s > \binom{r+1}{2}n$, this algorithm computes a polynomial $F(x, y) = \sum_{i=0}^{\ell} F_i(x)y^i$ such that $\deg(F_i) < d_i$, $F(x_t, y_t) = 0$ for $t = 1, \ldots, n$, and such that $F(x + x_t, y + y_t)$ does not have any monomial of degree less than r.*

(1) *Let $d := \lceil \log_q(s + n + 1) \rceil$ and construct the finite field $\mathbb{F}_{q^d}$.*

(2) *Choose s pairwise distinct nonzero elements $\epsilon_1, \ldots, \epsilon_s$ in $\mathbb{F}_{q^d}$ such that $\epsilon_i \neq x_j$ for $i \neq j$.*

(3) *Run Algorithm* **??** *on input*

$$D, J^\top, G, B, \underbrace{(1, 1, \ldots, 1)}_{s\text{times}},$$

where D is the diagonal matrix with diagonal entries $1/\epsilon_1, \ldots, 1/\epsilon_s$, J is the matrix defined in (11), G is the matrix defined in (18), and B is the matrix defined in (20). Let w denote the output.

(4) *Compute $v := w \cdot W$ where W is the Vandermonde matrix with (i, j)-entry equal to ϵ_i^{j-1}. Let $v = (v_{00}, v_{01}, \ldots, v_{0,d_0-1}, \ldots, v_{\ell,0}, v_{\ell,1}, \ldots, v_{\ell,d_\ell-1})$ denote the output.*

(5) *Output $F(x, y) = \sum_{i=0}^{\ell} F_i(x)y^i$ where $F_i(x) = v_{i,0} + v_{i,1}x + \cdots + v_{i,d_i-1}x^{d_i-1}$.*

Theorem 6.2. *Algorithm 6.1 correctly computes its output with $O(s^2\ell)$ operations over $\mathbb{F}_{q^d}$, or, equivalently, $O(s^2\ell \log_q(s)^2)$ operations over $\mathbb{F}_q$.*

Proof. The finite field $\mathbb{F}_{q^d}$ can be constructed with a randomized algorithm with an expected number of $O(d^3 \log(d) \log(dq))$ operations over the field $\mathbb{F}_q$ [7, Th. 14.42]. This cost is negligible compared to the cost of the subsequent steps. By Lemma **??**, Step (3) requires $O(\ell s^2)$ operations over $\mathbb{F}_{q^d}$. Steps (4), (5), and (6) each require $O(s^2)$ operations over $\mathbb{F}_{q^d}$. Each $\mathbb{F}_{q^d}$-operation can be simulated with $O(d^2)$ operations over $\mathbb{F}_q$. The result follows. □

7 Parallel Algorithms

The aim of this section is to show how the displacement method can be used to obtain parallel algorithms for computing a nontrivial element in the kernel of a structured matrix. The model of parallel computation that we use is a PRAM (Parallel Random Access Machine). A PRAM consists of several independent sequential processors, each with its own private memory, communicating with one another through a global memory. In one unit of time, each processor can read one global or local memory location, execute a single random access operation, and write into one global or local memory location. We assume in the following the each processor in the PRAM is capable of performing operations over $\mathbb{F}_q$ in a single step. The *running time* of a PRAM is the number of steps it performs to finish its computation.

We will use the following standard facts: one can multiply $m \times \alpha$- with $\alpha \times k$-matrices on a PRAM in $O(\alpha k)$ time on m processors. This is because each of the m processors performs independently a vector-matrix multiplication of size $1 \times \alpha$ and $\alpha \times k$. Further, one can solve an upper triangular $m \times m$-system of equations in time $O(m)$ on m processors. There are various algorithms for this problem. One is given in [14, Sect. 2.4.3]. As a result, it is easily seen that Algorithm 2.2 can customized to run in parallel, if steps (1) and (3) can be performed in parallel. More precisely, if these steps can be performed on m processors in time $O(\alpha)$, then the whole algorithm can be customized to run in time $O(\alpha(m+n))$ on m processors.

However, one cannot always perform steps (1) and (3) in parallel. Whether or not this is possible depends heavily on the displacement structure used. For instance, it is easy to see that the recovery algorithm given in Section 3 cannot be parallelized (at least in an obvious way). The reason for this is that for obtaining the ith entry of the first row one needs to know the value of the $(i-1)$st entry. To obtain the nth entry one thus needs to know the value of all the previous entries. It is therefore not possible to perform this step in time $O(\alpha)$ even if the number of processors is unbounded!

A closer look at this problem reveals the following: if V has a displacement structure of the type $DV - V\Delta = GB$ with *diagonal* matrices D and Δ, then one can recover the first row and the first column of V in parallel, if the nonzero entries of D and Δ are pairwise distinct. The algorithm for this task is rather simple and is given below.

The problem with this approach is that the matrices V we are studying do not necessarily have low displacement structure with respect to a pair of diagonal matrices. This problem can be fixed in certain cases. For instance, suppose that $V \in \mathbb{F}_q^{m \times n}$ has low displacement rank with respect to D and Z_n, where D is diagonal and Z_n is the upper shift matrix of format n:

$$DV - VZ_n = G_1 B_1. \tag{21}$$

Let W be a Vandermonde matrix whose (i,j)-entry is ϵ_i^{j-1}, where $\epsilon_1, \ldots, \epsilon_n$ are elements in $\mathbb{F}_q$ that are pairwise distinct and different from the diagonal entries of D. Then we have the following displacement structure for W

$$\Delta W - W Z_n^\top = G_2 B_2$$

where $G_2 = (\epsilon_1^n, \ldots, \epsilon_n^n)^\top$ and $B_2 = (0, 0, \ldots, 0, 1)$. Transposing this equation and multiplying with V from the left gives

$$VZ_nW^\top - W^\top\Delta = (VB_2^\top)G_2. \tag{22}$$

Multiplying (21) with W from the right, and adding that to the previous equation gives

$$DVW^\top - VW^\top\Delta = \left(G_1 | VB_2^\top\right)\begin{pmatrix} B_1W^\top \\ G_2 \end{pmatrix} =: GB. \tag{23}$$

This is now the correct displacement structure, but for the wrong matrix $VW^\top$. However, if w is a nonzero vector in the kernel of $VW^\top$, then $v := W^\top w$ is in the kernel of V. Moreover, since W is invertible, v is nonzero, hence is the desired solution.

The rest of this section deals with putting these ideas into effective use. We will first start by designing an algorithm that computes the first row and the first column of the matrix V that has a displacement structure with respect to two diagonal matrices.

Algorithm 7.1. *The input to the algorithm are two diagonal matrices D and Δ, and two additional matrices $G = (G_{ij}) \in \mathbb{F}_q^{m\times\alpha}$ and $B = (B_{ij}) \in \mathbb{F}_q^{\alpha\times n}$. We assume that note of the diagonal entries $x_1, \ldots, x_n$ is equal to any of the diagonal entries $z_1, \ldots, z_m$ of Δ. The output of the algorithm are the first row $(r_1, \ldots, r_m)$ and the first column $(c_1, \ldots, c_n)^\top$ of the matrix V given by the displacement structure $DV - V\Delta = GB$.*

(1) *Compute the first row $(\rho_1, \ldots, \rho_m)$ and the first column $(\gamma_1, \ldots, \gamma_n))^\top$ of GB.*

(2) *For $i = 1, \ldots, n$ set in parallel $c_i := \gamma_i/(x_i - z_1)$.*

(3) *For $i = 1, \ldots, m$ set in parallel $r_i := \rho_i/(x_1 - z_i)$.*

Proposition 7.2. *The above algorithm correctly computes its output with $O(\alpha)$ operations on a PRAM with $\max\{m, n\}$ processors.*

Proof. Correctness of the algorithm is easily obtained the same way as that of Algorithm 3.1. As for the running time, note that Step (1) can be performed in time $O(\alpha)$ on $\max\{m, n\}$ processors. Steps (2) and (3) obviously need a constant number of steps per processor. □

To come up with a memory-efficient procedure, we will customize Algorithm 2.4 rather than Algorithm 2.2. For this we need to have a displacement structure for $\binom{V}{I_n}$. This is given by the following:

$$\begin{pmatrix} D & 0 \\ 0 & \Delta \end{pmatrix}\begin{pmatrix} V \\ I_n \end{pmatrix} - \begin{pmatrix} V \\ I_n \end{pmatrix}\Delta = \begin{pmatrix} G \\ 0 \end{pmatrix}B. \tag{24}$$

Algorithm 7.3. *On input a diagonal matrix $D \in \mathbb{F}_q^{n\times n}$ with diagonal entries $x_1, \ldots, x_n$, a diagonal matrix $\Delta \in \mathbb{F}_q^{(n+1)\times(n+1)}$ with diagonal entries $z_1, \ldots, z_{n+1}$ such that the x_i and the z_j are pairwise distinct, and $z_i \neq x_j$ for $i \neq j$, a matrix $G \in \mathbb{F}_q^{n\times\alpha}$, and a matrix $B \in \mathbb{F}_q^{\alpha\times(n+1)}$, the algorithm computes a nonzero vector $v \in \mathbb{F}_q^n$ in the kernel of the matrix V given by the displacement equation $DV - V\Delta = GB$.*

(1) *Set* $G_0 := \binom{G}{0}$, $B_0 := B$.

(2) *For* $i = 0, \ldots, n-1$ *do:*

(a) *Let* D_i *be the diagonal matrix with diagonal entries* $x_{i+1}, \ldots, x_n, z_1, \ldots, z_i$, *and let* Δ_i *be the diagonal matrix with diagonal entries* $z_{i+1}, \ldots, z_{n+1}$.

(a) *Use Algorithm 7.1 to compute the first column* $(c_1, c_2, \ldots, c_{2n+1-i})^\top$ *of the matrix* V_i *given by the displacement equation*

$$\begin{pmatrix} D_i & 0 \\ 0 & \Delta \end{pmatrix} V_i - V_i \Delta_i = G_i B_i.$$

(b) *If* $c_1 = \cdots = c_{n-i} = 0$, *then* **output** *the vector* $v := (c_{n-i+1}, \ldots, c_n, 1, 0, \ldots, 0)$. *and* **Stop**. *Otherwise, find the smallest* k *such that* $c_k \neq 0$, *interchange* x_1 *and* x_k, c_1 *and* c_k, *and the first and the* k*th rows of* G_i.

(c) *Use Algorithm 7.1 again to compute in parallel the first row* $(r_1, \ldots, r_{n-i+1})$ *of the matrix* V_i *given by* $D_i V_i - V_i \Delta_i = G_i B_i$ *(note that* V_i *is not necessarily the same as in Step (2a) since* D_i *and* G_i *may not be the same).*

(d) *For* $j = 2, \ldots, 2n - i + 1$ *replace row* j *of* G_i *by* $-c_j/c_1$ *times the first row plus the* j*-th row. Set* G_{i+1} *as the matrix formed from* G_i *by deleting the first row.*

(e) *For* $j = 2, \ldots, n - i + 1$ *replace the* j*-th column of* B_i *by* $-r_j/c_1$ *times the first column plus the* j*-th column. Set* B_{i+1} *as the matrix formed from* B_i *by deleting the first column.*

Theorem 7.4. *The above algorithm correctly computes its output and uses* $O(\alpha n)$ *operations on a PRAM with* $n + 1$ *processors.*

Proof. This algorithm is a customized version of Algorithm 2.2 and its correctness follows from that of the latter. As for the running time, note that Step (2a) takes $O(\alpha)$ steps on a PRAM with $2n+1-i$ processors by Proposition 7.2 (G_i has $2n+1-i$ rows). However, a simple induction shows that the last $n+1-i$ rows of G_i are zero. As a result, we only need $n-i$ processors for this step. Step (2c) can be performed in time $O(\alpha)$ on $n+1-i$ processors by Proposition 7.2. Steps (2d) and (2e) can be performed in time $O(\alpha)$ on $n+1-i$ processors (again, the number of processors is not equal to the number of rows of G_i since the last $n+1-i$ rows of G_i are zero). At each round of the algorithm the processors can be reused, so the total number of processors equals to the maximum number at each round, i.e., it equals $n+1$. The running time is at most $O(n\alpha)$ since step (2) is performed at most n times. □

The above algorithm can now be used as a major building block for solving the 2-dimensional interpolation problem in parallel.

Algorithm 7.5. *On input* $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbb{F}_q^2$ *and integers* $d_0, d_1, \ldots, d_\ell$ *such that none of the* x_i *is zero and such that* $\sum_{i=0}^{\ell} d_i = n + 1$, *this algorithm computes a polynomial* $F(x, y) = \sum_{i=0}^{\ell} F_i(x) y^i$ *such that* $\deg(F_i) < d_i$ *for* $i = 0, \ldots, \ell$ *and* $F(x_t, y_t) = 0$ *for* $t = 1, \ldots, n$.

(1) *Let* $d := \lceil \log_q(2n+1) \rceil$ *and construct the field* $\mathbb{F}_{q^d}$. *Find* $n+1$ *elements* $\epsilon_1, \ldots, \epsilon_{n+1}$ *in* $\mathbb{F}_{q^d}$ *that are pairwise distinct and such that* $\epsilon_i \neq 1/x_j$ *for* $i \neq j$.

(2) *Set* $G := (G_1 \mid v_1)$ *where* G_1 *is the matrix given in (7) and* v_1 *is the last column of the matrix* V *given in (6).*

(3) *Compute* $\tilde{B} := B_1 W^\top$ *where* B *is the matrix given in (8) and* W *is the* $(n+1)\times(n+1)$ *Vandermonde matrix having* (i,j)*-entry* ϵ_i^{j-1}. *Set* $B := \binom{\tilde{B}}{G_2}$ *where* G_2 *is the vector* $(\epsilon_1^{n+1}, \ldots, \epsilon_{n+1}^{n+1})$.

(4) *Run Algorithm 7.3 on input* D, Δ, G, B, *where* D *is a diagonal matrix with diagonal entries* $1/x_1, \ldots, 1/x_n$, Δ *is the diagonal matrix with diagonal entries* $\epsilon_1, \ldots, \epsilon_{n+1}$, *and* G *and* B *are the matrices computed in the previous two steps. Let* w *denote the output.*

(5) *Compute in parallel the vector* $v := W^\top w$. *Let* $(v_{00}, v_{01}, \ldots, v_{0,d_0-1}, \ldots, v_{\ell,0}, v_{\ell,1}, \ldots, v_{\ell,d_\ell-1})$ *denote its components.*

(6) *Output* $F(x,y) = \sum_{i=0}^{\ell} F_i(x) y^i$ *where* $F_i(x) = v_{i,0} + v_{i,1}x + \cdots + v_{i,d_i-1}x^{d_i-1}$.

Theorem 7.6. *The above algorithm correctly computes its output with* $O(\ell n \log_q(n)^2)$ *operations on a PRAM with* $n+1$ *processors.*

Proof. The matrix $VW^\top$ satisfies the displacement equation 23 with the matrices G and B computed in steps (2) and (3). Step (4) then produces a nontrivial element in the kernel of $VW^\top$. This shows that the vector v computed in Step (5) is indeed a nontrivial element in the right kernel of V, and proves correctness of the algorithm.

The finite field $\mathbb{F}_{q^d}$ can be constructed probabilistically in time proportional to $d^3 \log(qd) \log(d)$ on one processor [7, Th. 14.42]. Arithmetic operations in $\mathbb{F}_{q^d}$ can be simulated using d^2 operations over $\mathbb{F}_q$ on each processor. In the following we thus assume that the processors on the PRAM are capable of executing one arithmetic operation in $\mathbb{F}_{q^d}$ in $O(d^2)$ steps. In Step (2) G_1 can be calculated with $O(\alpha \log(n))$ $\mathbb{F}_{q^d}$ operations on n processors (each of which computes the rows of G_1 independently). Likewise, v_1 can be computed with $O(\log(n))$ operations on n processors. Hence, computation of G requires $O(\ell \log(n))$ operations over $\mathbb{F}_{q^d}$. Given the special structure of the matrix B_1, $\tilde{B}$ can be computed with $O(n)$ $\mathbb{F}_{q^d}$ operations on $n+1$ processors. G_2 can be computed with $O(\log(n))$ operations on $n+1$ processors. Hence, steps (2) and (3) require $O(\ell \log(n) + n)$ operations on $n+1$ processors. Step (4) needs $O(\ell n)$ operations on $(n+1)$ processors by Theorem 7.4, and Step (5) needs $O(n)$ operations on $n+1$ processors. Hence, the total number of $\mathbb{F}_{q^d}$-operations of the algorithm equals $O(\ell n)$ and it can be performed on $n+1$ processors. □

References

[1] S. Ar, R. Lipton, R. Rubinfeld, and M. Sudan. Reconstructing algebraic functions from mixed data. In *Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science*, pages 503–512, 1992.

[2] D. Augot and L. Pecquet. An alternative to factorisation: a speedup for Sudan's algorithm and its generalization to algebraic-geometric codes. Preprint, 1998.

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

[4] E.R. Berlekamp. Bounded distance + 1 soft decision Reed-Solomon decoding. *IEEE Trans. Inform. Theory*, 42:704–720, 1996.

[5] R. Bitmead and B. Anderson. Asymptotically fast solution of Toeplitz and related systems of linear equations. *Linear Algebra and its Applications*, 34:103–116, 1980.

[6] S. Gao and M.A. Shokrollahi. Computing roots of polynomials over function fields of curves. Preprint, 1998.

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

[8] I. Gohberg and V. Olshevsky. Fast state-space algorithms for matrix Nehari and Nehari-Takagi interpolation problems. *Integral Equations and Operator Theory*, 20:44–83, 1994.

[9] V. Guruswami and M. Sudan. Improved decoding of Reed-Solomon and algebraic-geometric codes. In *Proceedings of the 39th IEEE Symposium on Foundations of Computer Science*, pages 28–37, 1998.

[10] G. Heinig and K. Rost. *Algebraic Methods for Toeplitz-like matrices and operators*, volume 13 of *Operator Theory.* Birkhäuser, Boston, 1984.

[11] T. Høholdt and R. Refslund Nielsen. Decoding Hermitian codes with Sudan's algorithm. Preprint, Denmark Technical University, 1999.

[12] J. Justesen, K.J. Larsen, H.E. Jensen, and T. Høholdt. Fast decoding of codes from algebraic plane curves. *IEEE Trans. Inform. Theory*, 38:111–119, 1992.

[13] T. Kailath and A.H. Sayed. Displacement structure: Theory and applications. *SIAM Review*, 37:297–386, 1995.

[14] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures : Arrays, Trees, Hypercubes.* Academic Press/Morgan Kaufmann, 1992.

[15] F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes.* North-Holland, 1988.

[16] M. Morf. *Fast algorithms for multivariable systems.* PhD thesis, Stanford University, 1974.

[17] M. Morf. Doubling algorithms for Toeplitz and related equations. In *Prceedings of IEEE Conference on Acoustics, Speech, and Signal Processing, Denver*, pages 954–959, 1980.

[18] V. Olshevsky. Pivoting for structured matrices with applications. http://www.cs.gsu.edu/~matvro, 1997.

[19] V. Olshevsky and V. Pan. A superfast state-space algorithm for tangential Nevanlinna-Pick interpolation problem. In *Proceedings of the 39th IEEE Symposium on Foundations of Computer Science*, pages 192–201, 1998.

[20] R. Roth and G. Ruckenstein. Efficient decoding of reed-solomon codes beyond half the minimum distance. In *Prceedings of 1998 International Symposium on Information Theory*, pages 56–56, 1998.

[21] M.A. Shokrollahi and H. Wasserman. List decoding of algebraic-geometric codes. 45:432–437, 1999. IEEE Trans. Inform. Theory.

[22] M. Sudan. Decoding of Reed-Solomon codes beyond the error-correction bound. *J. Compl.*, 13:180–193, 1997.

[23] L.R. Welch and E.R. Berlekamp. Error correction for algebraic block codes. U.S. Patent 4,633,470, issued Dec. 30, 1986.

8 Appendix

In this appendix we shall clarify how to implement the step 3 of the algorithm 6.1, i.e., how to find a vector in the right kernel of a matrix $R := VW^{\top}$ satisfying

$$JR - R\Delta = GB$$

(cf. with (17)). By Lemma 2.3 we could use the extended displacement equation

$$\begin{pmatrix} J & 0 \\ 0 & \Delta \end{pmatrix} \begin{pmatrix} R \\ I \end{pmatrix} - \begin{pmatrix} R \\ I \end{pmatrix} \Delta = \begin{pmatrix} G \\ 0 \end{pmatrix} B \tag{25}$$

to try to run the algorithm 2.4 to compute a vector in the right kernel of $R := VW^{\top}$. Indeed, the latter algorithm is based on Lemma 2.1 which is directly applicable in our situation because $\begin{pmatrix} J & 0 \\ 0 & \Delta \end{pmatrix}$ is lower triangular, and Δ is upper triangular (even diagonal).

However, there are several differences between (25) and (1) that do not allow one to apply the algorithm 2.4 directly. Here is the list of these differences:

1. A closer look at (25) reveals that the mapping $\nabla : \mathbb{F}^{(m+s)\times s} \longrightarrow \mathbb{F}^{(m+s)\times\alpha} \times \mathbb{F}^{\alpha\times s}$ defined by $\nabla S = \begin{pmatrix} J & 0 \\ 0 & \Delta \end{pmatrix} S - S\Delta$ is not an isomorphism, so the the generator $\{\begin{pmatrix} G \\ 0 \end{pmatrix}, B\}$ no longer contains the full information about $\begin{pmatrix} R \\ I \end{pmatrix}$ in (25). Thus, we need to introduce a new definition for a generator,
$$\{\begin{pmatrix} G \\ 0 \end{pmatrix}, B, D\} \tag{26}$$
including in it one more matrix D to completely describe our structured matrix $\begin{pmatrix} R \\ I \end{pmatrix}$.

2. Lemma 2.3 shows how to compute a vector in the kernel of R via computing the Schur complements of $\begin{pmatrix} R \\ I \end{pmatrix}$. Further, Lemma 2.1 allows us to speed-up the procedure by manipulation on two generator matrices $\{\begin{pmatrix} G \\ 0 \end{pmatrix}, B\}$. In our situation (having a new extended generator (26)) we need to specify a recursion for the third matrix in (26) as well.

3. One of the ingredients of the algorithm 2.4 is recovering the first row and the first column of a structured matrix from its generator. We need to clarify this procedure for our new extended generator in (26).

4. In what follows we provide formulas resolving the above three difficulties, and specify a modified version of the fast algorithm 2.4 for our matrix $S := \begin{pmatrix} R \\ I \end{pmatrix}$. However, the original algorithm 2.4 employes partial row pivoting to run to the completion.[1] Row pivoting was possible due to the diagonal form of the matrix D in (1). In contrast, the new matrix J in (25) is no longer diagonal, so we have to find a different pivoting strategy.

We next snow resolve the above four difficulties. Let S satisfy

$$AS - S\Delta = GB, \tag{27}$$

where we set $A := \begin{pmatrix} J & 0 \\ 0 & \Delta \end{pmatrix} \in \mathbb{F}^{(n+s)\times(n+s)}$ and $\Delta \in \mathbb{F}^{s\times s}$ is diagonal.

1. A general theory of displacement equations with kernels can be found in [**?**, **?**], and this more delicate case appears in studying *boundary* rational interpolation problems and in the design of preconditioners [**?**]. As we shall se below, we can recover from $\{G, B\}$ on the right-hand side of (27) all the enries of the $(n+s)\times s$ matrix S but s diagonal elements $\{S(n+1,1), S(n+2,2), \dots, S(n+s,s)\}$. Therefore the triple

$$\{G, B, D\} \qquad \text{with} \qquad D := \{S(n+1,1), S(n+2,2), \dots, S(n+s,s)\} \tag{28}$$

contains the full information on S, so we call it a *generator*.

It should be mentioned that the idea of generator is in compressing the information on n^2 entries of S into just $O(n)$ entries of its generator. The newly defined generator in (28) involves just s more parameters (i.e., the entries of the $1 \times s$ row D) and therefore is still a very efficient representation.

2. Each step of the algorithm 2.4 consists of two parts: (1) recovering the first row and column of S from its generator; (2) computing a generator $\{G_2, B_2, D_2\}$ for the Schur complement S_2 of S. We shall discuss the part (1) in the next item, and here we address the part (2). A closer look at lemma 2.1 reveals that it applies in our situation and the formulas for computing $\{G_2, B_2\}$ remain valid. It remains to show how to compute the row $D_2 = \{d_1^{(2)}, d_2^{(2)}, \dots, d_s^{(2)}\}$ capturing the $\{(n-1+1,1), (n-1+2,2), \dots, (n-1+s,s)\}$ elements of S_2. Since $S_2 = S_{22} - S_{21}S_{11}^{-1}S_{12}$ is the Schur complement of $S = \begin{pmatrix} S_{11} & S_{12} \\ S_{21} & S_{22} \end{pmatrix}$ hence we have the following formulas

$$d_k^{(2)} = S_{n+k,k} - S_{n+k,1}S_{1,1}^{-1}S_{1,k},$$

where all the involved entries S_{ij} of S are available: $\{S_{n+k,1}, S_{1,1}, S_{1,k}\}$ appear either in the first row or in the first column (we shall show next how to recover them from (28)), and $S_{n+k,k}$ is an entry of the matrix D in (28).

[1] Specifically, at each step the algorithm 2.4 uses row interchanges to bring a nonzero element in the (1,1) position of S (moreover, the absence of appropriate non-zero elements was the stopping criterion).

3. Now we are able to provide formulas to recover the first row and column of S in (27) from its generator in (28).

   The elements of the top row can be obtained by the formula

   $$S(1,k) = \frac{G(1,:)B(:,k)}{A(1,1) - \Delta(1,k)},$$

   where we follow the MATLAB notations and dentote by $G(1,:)$ the first row of G and by $B(:,k)$ the k-th column of B. This formula works only if $A_{1,1} \neq \Delta_{1,k}$. If $A_{1,1} = \Delta_{1,k}$ (this may happen only for $k = 1$) then $S_{1,1}$ cannot be recovered from $\{G, B\}$ and is therefore stored in the third matrix D of (28).

   Since A is a bi-diagonal matrix, the elements of the first column of S can be recovered as follows:

   $$S(1,1) = \frac{G(1,:)B(:,1)}{A(1,1) - \Delta(1,1)}, \qquad S(k,1) = \frac{[G(k,:) - A(k,k-1)G(k-1,:)]B(:,1)}{A(k,k) - \Delta(1,1)}$$

4. The above items describe several modifications to the algorithm 2.4. The modified algorithm terminates successfully if after k steps when the first $m - k$ entries of the $(m - k + s) \times s$ Schur complement are zero (cf. Lemma refle:direct)). However, we need to implement an appropriate pivoting strategy in ensure that at each step of the algorithm, the element $S(1,1)$ of the Schur complement is non-zero. Pivoting is the standard method to bring a non-zero element in the (1,1) position, however, arbitrary pivoting can destroy the structure of S and thus block further efficient steps of the algorithm. Indeed, lemma 2.1 requires that the matrix A is lower triangular and the matrix Δ in (27) is upper triangular, which can be destroyed by row/column interchanges. In our case the matrix Δ is diagonal, so any $(P^\top \Delta P)$ is diagonal as well. Therefore the permuted (SP) satisfies

   $$A(SP) - (SP)(P^\top \Delta P) = G(BP).$$

   and thus pivoting can be easily implemented in terms of operations on the generator of S. However, the above column pivoting does not remove all difficulties, and it may happen that the entire first column of S is zero although there are still non-zero entries in the first column of S. In this case one may proceed as follows. Since the top column of S is zero, we are free to make any changes in the first column of A in (27). Let us zero all the entries of $A(:,1)$ but $A(1,1)$. Then A becomes a block-diagonal matrix. Denote by Q a permutation that moves all entries up and moves the first entry to the bottom. Then (27) implies

   $$(QAQ^\top)(QS) - (QS)\Delta = (QG)B.$$

   where $(QAQ^\top)$ is still lower triangular.

   Thus applying column pivoting we bring non-zero entries in the first row to the (1,1) position, and applying row pivoting we move totally zeros to the bottom. It is clear that the algorithm now runs to a successful completion.