Lecture 7

The Displacement Method I

7.1. Motivation

The decoding problem for various families of linear error-correcting codes can be reduced to the problem of solving a system of linear equations. For the case of Reed-Solomon codes that we have already studied, the Berlekamp-Welch decoder can be seen as a reduction of the decoding problem to solving a homogeneous system of n linear equations in $n + 1$ unknowns, where n is the block length of the code. Note that any homogeneous system of linear equations is satisfied by the trivial zero solution. Indeed for obtaining the closest codeword to the received sequence we need a nontrivial solution, which always exists, as the number of unknown is greater than the number of equations. Moreover, to compute the decoded message, another system of linear equations has to be solved using the obtained codeword.

A similar situation arises for the case of Sudan and Guruswami-Sudan list-decoders, where list decoding is reduced to the problem of solving a system of homogeneous linear equations for which a nontrivial solution is sought. Moreover, the number of the resulting equations (and unknowns) is linear with respect to the block length of the code, provided that the multiplicity parameter for the latter decoder is a constant.

Any system of m linear equations in n unknowns can be solved using the Gaussian elimination algorithm which requires $O(m^2n)$ arithmetic operations, or $O(n^3)$ when $m = O(n)$. In particular, we get a cubic upper bound on the running time of the aforementioned decoding algorithms.

The cubic running time of the Gaussian elimination algorithm is more than enough for many applications in theoretical computer science. However, this might still be too much in practice, when the performance of a real-time

Lecture by Amin Shokrollahi
Scribes by Mahdi Cheraghchi

system has to be optimized or the cost of a hard disk controller is to be minimized. On the other hand, the Berlekamp-Welch algorithm in its original form performs much better than the general Gaussian elimination process, and has the additional property that its running time depends on the number of errors. That is, the less corrupted the received word is, the faster the decoding algorithm performs. This motivates the following question: How can we *tune* the Gaussian elimination algorithm in a way that it performs better in our special cases than its full generality?

One possible approach is to improve the Gaussian elimination algorithm so that it performs faster in solving general equations. Indeed such improvements exist, but to our current knowledge, it is not known whether they are optimal. However, despite such improvements, one may wonder if there is a faster way of solving the linear equations of *our interest*. In the sequel, we will see that the answer is *yes*.

7.2. The Displacement Structure

How can we obtain faster algorithms for solving the special families of linear equations arising in our coding-theoretic application? A key observation is that the linear equations we are dealing with have special structures. For the case of Reed-Solomon codes the matrices defining the linear equations consist of blocks of Vandermonde matrices, possibly multiplied by certain diagonal matrices.

Recall the structure of an $m \times n$ Vandermonde matrix:

$$V_{m,n} := \begin{pmatrix} 1 & x_1 & \cdots & x_1^{n-1} \\ 1 & x_2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & \cdots & x_m^{n-1} \end{pmatrix}.$$

Note than an $m \times n$ Vandermonde matrix can be fully described by the vector $(x_1, \ldots, x_m)$, rather than all the mn entries. This means that Vandermonde matrices have a particular structure that allows them to be represented in a dramatically *compressed* form.

Various other structured families of matrices arise in coding theory (particularly in decoding algebraic codes). Of such matrices we can mention Hankel and Cauchy matrices:

$$\mathrm{Hankel}_{m,n} := \begin{pmatrix} x_1 & x_2 & \cdots & x_n \\ x_2 & x_3 & \cdots & x_{n+1} \\ \vdots & \vdots & \ddots & \vdots \\ x_m & x_{m+1} & \cdots & x_{m+n-1} \end{pmatrix},$$

$$\text{Cauchy}_{m,n} := \begin{pmatrix} \frac{1}{x_1+y_1} & \frac{1}{x_1+y_2} & \cdots & \frac{1}{x_1+y_n} \\ \frac{1}{x_2+y_1} & \frac{1}{x_2+y_2} & \cdots & \frac{1}{x_2+y_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{1}{x_m+y_1} & \frac{1}{x_m+y_2} & \cdots & \frac{1}{x_m+y_n} \end{pmatrix},$$

which are completely specified by the vectors $(x_1, \ldots, x_{m+n-1})$ and $(x_1, \ldots, x_m, y_1, \ldots, y_n)$, respectively.

The second key observation is that low rank matrices already have compressed representations. In particular, it is a folklore fact that any $m \times n$ matrix of rank r can be represented as the multiplication of an $m \times r$ matrix by an $r \times n$ matrix. Consequently, all low rank matrices allow compact representations using $r(m+n)$ field elements rather than the usual mn, the difference of which could be significant.

The above observations lead us to a suitable formalization of particular *compressed representations* of the structured matrices. Of course the matrices of our interest do not necessarily have low ranks, and in fact for all interesting cases they have maximal ranks. However, this is not the end of the story, if we can find a bijective mapping to transform a structured family to the space of low rank matrices. As it turns out, such mappings exist, that we shall call *displacement operators*, and indeed there are several ways of defining them. The following definition of the displacement operator is well-suited for our purposes:

Definition 7.1. A displacement operator on the space of $m \times n$ matrices over a field $\mathbb{F}$ (i.e., $\mathbb{F}^{m \times n}$) is a bijective linear mapping (isomorphism) $\nabla \colon \mathbb{F}^{m \times n} \to \mathbb{F}^{m \times n}$ defined as $\nabla(X) = AX - XB$, for fixed matrices $A \in \mathbb{F}^{m \times m}$ and $B \in \mathbb{F}^{n \times n}$. For a particular choice of the matrices A and B, we will denote the displacement operator by $\nabla_{A,B}$.

A displacement operator defined as above is only interesting when the rank r of the image $\nabla_{A,B}(X)$ (denoted as the *displacement rank*) is guaranteed to be considerably lower than m and n, provided that X is chosen from a structured family. In that case, since the matrices A and B are fixed, we can use the low rank matrix decomposition discussed above to obtain a compact representation of size $r(m+n)$ for the family X belongs to. That is, X can be represented as a pair of matrices of dimension $m \times r$ and $r \times n$ that we will identify as the *generators* of X.

Now let us design a displacement operator to capture the structure of the Vandermonde matrices. We only need to find appropriate choices for the matrices A and B corresponding to the particular family of our interest. Consider the diagonal $m \times m$ matrix $\Delta_m := \text{diag}(1/x_1, \ldots, 1/x_m)$ and the $n \times n$ *shift* matrix defined as

$$Z_n := \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ 0 & 0 & 0 & \cdots & 0 \end{pmatrix},$$

as our choices of A and B, respectively. The action of the diagonal matrix is a multiplication of the i^{th} row by the reciprocal of x_i, and the action of the shift matrix (acting from the right) is precisely a shift of the entries of the matrix under the action by one column to the right, filling the first column with zeros. For Vandermonde matrices, these two actions yield *almost* the same results, modulo the first column. In particular we have

$$\Delta_m V_{m,n} - V_{m,n} Z_n = \begin{pmatrix} 1/x_1 & 0 & 0 & \cdots & 0 \\ 1/x_2 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1/x_m & 0 & 0 & \cdots & 0 \end{pmatrix}_{m \times n},$$

which is a rank one matrix, and can be written as

$$\begin{pmatrix} 1/x_1 \\ 1/x_2 \\ \vdots \\ 1/x_m \end{pmatrix} \cdot \underbrace{(1, 0, \ldots, 0)}_{n}.$$

This precisely captures the compact representation of the Vandermonde matrices we had informally discussed.

Now we turn to a more complicated example of the family arising from Sudan's list-decoding algorithm. Recall that the matrix defining the system of linear equations in this case is of the form

$$X := \left(D^{\ell} V_{n,e} \mid D^{\ell-1} V_{n,e+(k-1)} \mid \cdots \mid D V_{n,e+(\ell-1)(k-1)} \mid V_{n,e+\ell(k-1)} \right),$$

for a diagonal matrix D chosen to be $D := \mathrm{diag}(y_1, \ldots, y_n)$, certain integers e and k depending on the designated parameters of the code, and $n \times e + i(k-1)$ (for $i = 0, \ldots, \ell$) Vandermonde matrices $V_{n,e+i(k-1)}$ as already defined.

In this case, our choices of the matrices A and B will be similar to the case of Vandermonde matrices. Note that for a matrix of the form $D^i V_{n,t}$, exactly the same choice of $A := \Delta_n$ and $B := Z_t$ as before gives us the compact form

$$\begin{pmatrix} y_1^i/x_1 \\ y_2^i/x_2 \\ \vdots \\ y_n^i/x_n \end{pmatrix} \cdot \underbrace{(1, 0, \ldots, 0)}_{t}.$$

Here, our matrix X consists of blocks of such matrices. To extract the meat from X, we need to devise a displacement operator which operates properly on each block. A moment of thought gives us the operator $\nabla(X) := \Delta_n X - XZ$, where Z is a block diagonal matrix where each block is the shift operator of the proper size. More specifically, the block diagonal entries of Z are chosen to be $Z_e, Z_{e+(k-1)}, \ldots, Z_{e+\ell(k-1)}$. This operator gives us a

rank $\ell + 1$ matrix that can be decomposed as

$$\Delta_n X - XZ = \begin{pmatrix} y_1^\ell/x_1 & y_1^{\ell-1}/x_1 & \cdots & 1/x_1 \\ y_2^\ell/x_2 & y_2^{\ell-1}/x_2 & \cdots & 1/x_2 \\ \vdots & \vdots & \ddots & \vdots \\ y_n^\ell/x_n & y_n^{\ell-1}/x_n & \cdots & 1/x_n \end{pmatrix} \cdot H,$$

where H consists of $\ell + 1$ adjacent blocks where the i^{th} block is an $(\ell + 1) \times (e + (i - 1)(k - 1))$ matrix that has the i^{th} unit vector in its first column and zero elsewhere. For instance, for the case $e = 3$, $\ell = 2$, and $k = 4$ we will have

$$H = \left(\begin{array}{ccc|cccccc|cccccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right).$$

7.3. Gaussian Elimination Revisited

Along with the displacement operator, the displacement method introduces the machinery needed to mimic the Gaussian elimination algorithm by directly working on the generators of the matrix defining the linear equations. As it turns out, this approach leads to a total running time of $O(rm(m + n))$ as opposed to $O(m^2 n)$ complexity of the Gaussian elimination algorithm.

At this point, it is worthwhile to have a quick review of the way the Gaussian elimination algorithm works. Informally, the algorithm is based on the observation that a system of linear equation $Ax = b$ is easier to solve when the matrix A is triangular, in which case an assignment satisfying the equations can be quickly identified using backward substitutions. In fact Gaussian elimination is a way of transforming an arbitrary linear system into such a form.

Another way to look at the Gaussian elimination algorithm is that it computes a decomposition of the input matrix A in the form $A := PLU$ where L and U are lower and upper triangular matrices, respectively, and P is a permutation matrix (that is, a square matrix with entries in $\{0, 1\}$ such that there is a single entry with value 1 in each row and column) which defines the appropriate permutations of the equations needed to obtain a solution. Note that having such a decomposition at our disposal, it is straightforward to obtain a solution for the equation $Ax = PLUx = b$, provided that one exists:

1. Apply the permutation P^{-1} to the entries of b to obtain b'.
2. Solve the equation $Ly = b'$ for y.
3. Solve $Ux = y$ to obtain x.

Note that inverting a permutation is quite easy, and as the matrices L and U are triangular, solving the equations involved in the last two steps is much easier than the general case. Hence, the main complexity of the Gaussian elimination algorithm is due to the computation of a PLU decomposition for the matrix defining the equations. Now let us rephrase these observation more precisely.

Let $A \in \mathbb{F}^{m\times n}$ and A_{11} be an invertible square submatrix of A. Hence up to a permutation, we can regard A as $\left(\begin{smallmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{smallmatrix}\right)$. Define the *Schur complement* of A with respect to A_{11} as $A_2 := A_{22} - A_{21}A_{11}^{-1}A_{12}$. Then it is easy to verify the decomposition

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \underbrace{\begin{pmatrix} I & \mathbf{0} \\ A_{21}A_{11}^{-1} & \tilde{I} \end{pmatrix}}_{A_\ell} \underbrace{\begin{pmatrix} A_{11} & \mathbf{0} \\ \mathbf{0} & A_2 \end{pmatrix}}_{A'} \underbrace{\begin{pmatrix} I & A_{11}^{-1}A_{12} \\ \mathbf{0} & \tilde{I} \end{pmatrix}}_{A_r}, \tag{7.1}$$

where I and $\tilde{I}$ are identity matrices of appropriate sizes.

The above decomposition is the main ingredient of the Gaussian elimination algorithm. Suppose that we are to find a nonzero element in the right kernel of the given matrix A (that is, a nonzero solution to the equation $Ax = \mathbf{0}$), and without loss of generality assume that no column of A is identically zero (otherwise the problem is trivial). First, the Gaussian elimination algorithm performs a *pivoting* step (by permuting the rows of A if necessary) to ensure that the entry a_{11} at the top-left corner is nonzero. Then it applies (7.1) by computing the Schur complement of A with respect to a_{11}.

Note that the matrices A_ℓ and A_r on the left and right hand side of the decomposition are triangular. Then the algorithm continues by recursively applying the decomposition to the matrix A' in the middle to finally obtain a decomposition $A := PLU$, where P is the permutation matrix accumulating all the permutations performed for pivoting and L and U are lower and upper triangular matrices, respectively (simply because multiplication of triangular matrices are triangular as well). Note that considering the PLU decomposition, $Ax = \mathbf{0}$ if and only if $Ux = \mathbf{0}$, since P is a permutation matrix and L is square lower triangular with ones on the main diagonal, both having a trivial right kernel. As U is triangular, finding a nonzero solution for $Ux = \mathbf{0}$ (and hence for $Ax = \mathbf{0}$) is now easily possible.

7.4. Finding a PLU Decomposition from the Generators

As mentioned before, the displacement method provides the means to apply the Gaussian elimination algorithm (or in other words, to compute a PLU decomposition) on structured matrices by *directly* working on their generators. The following lemma plays a key role in realization of such a framework:

Lemma 7.2. *Let $A \in \mathbb{F}^{m\times n}$ admit a displacement structure $\Delta A - AZ = GH$ for some $\Delta \in \mathbb{F}^{m\times m}$, $Z \in \mathbb{F}^{n\times n}$,*

$G \in \mathbb{F}^{m\times r}$, and $H \in \mathbb{F}^{r\times n}$. Suppose that the matrices involved can be partitioned in the following ways:

$$A = \begin{pmatrix} a_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad \Delta = \begin{pmatrix} \delta_{11} & \mathbf{0} \\ \star & \Delta_2 \end{pmatrix}, \quad Z = \begin{pmatrix} z_{11} & \star \\ \mathbf{0} & Z_2 \end{pmatrix}, \quad G = (G_1 \mid G')^\top, \quad H = (H_1 \mid H'),$$

$$a_{11}, \delta_{11}, z_{11} \in \mathbb{F}, \quad a_{11} \neq 0, \quad G_1, H_1 \in \mathbb{F}^{r\times 1}, \quad G' \in \mathbb{F}^{r\times(m-1)}, \quad H' \in \mathbb{F}^{r\times(n-1)}.$$

Then the Schur complement of A with respect to a_{11}, denoted by A_2, satisfies the displacement structure $\Delta_2 A_2 - A_2 Z_2 = G_2 H_2$, for some $G_2 \in \mathbb{F}^{(m-1)\times r}$ and $H_2 \in \mathbb{F}^{r\times(n-1)}$ which are independent of A_{22}.

Proof. Applying the decomposition (7.1) gives us

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & \mathbf{0} \\ a_{11}^{-1}A_{21} & I \end{pmatrix}}_{A_\ell} \underbrace{\begin{pmatrix} a_{11} & \mathbf{0} \\ \mathbf{0} & A_2 \end{pmatrix}}_{A'} \underbrace{\begin{pmatrix} 1 & a_{11}^{-1}A_{12} \\ \mathbf{0} & I \end{pmatrix}}_{A_r}.$$

Note that the matrices A_r and A_ℓ are atomic triangular matrices, i.e., all their nonzero off-diagonal entries lie on a certain (in this case the first) row or column. In particular, their inverses are in the form

$$A_\ell^{-1} = \begin{pmatrix} 1 & \mathbf{0} \\ -a_{11}^{-1}A_{21} & I \end{pmatrix}, \qquad A_r^{-1} = \begin{pmatrix} 1 & -a_{11}^{-1}A_{12} \\ \mathbf{0} & I \end{pmatrix}.$$

Combining the decomposition with the assumption $\Delta A - AZ = GH$, we get

$$\Delta A_\ell A' A_r - A_\ell A' A_r Z = GH.$$

Now we multiply the both sides from left by A_ℓ^{-1} and from right by A_r^{-1} to obtain

$$A_\ell^{-1}\Delta A_\ell A' - A' A_r Z A_r^{-1} = A_\ell^{-1} G H A_r^{-1}. \tag{7.2}$$

Observe that

$$A_\ell^{-1}\Delta A_\ell = \begin{pmatrix} 1 & \mathbf{0} \\ a_{11}^{-1}A_{21} & I \end{pmatrix} \begin{pmatrix} \delta_{11} & \mathbf{0} \\ \star & \Delta_2 \end{pmatrix} \begin{pmatrix} 1 & \mathbf{0} \\ -a_{11}^{-1}A_{21} & I \end{pmatrix},$$

which is identical to Δ except for the first column. Similarly, $A_r Z A_r^{-1}$ is the same as Z modulo the first row. Then the lemma immediately follows from (7.2), by restricting the equality to the minors obtained by removal of the first rows and columns. □

In the light of the above lemma it is now more clear how to mimic the Gaussian elimination process to find a nonzero element in the right kernel of a given matrix when we are only given access to its generators. In fact in order to do so, we need one more tool: An efficient means of *extracting* a particular row or column of a matrix from its

generators. We will discuss this problem in the next section, but for now take it for granted that such a possibility exists.

Equipped with the required machinery, it is straightforward to compute a PLU decomposition for a matrix A, given an access to its generators. Note that a PLU decomposition easily translates to a nonzero element in the right kernel of A as well. The main idea for computation of a PLU decomposition is to apply the Gaussian elimination process as we did before! Recall that the Gaussian elimination algorithm is based on the decomposition given in (7.1), that we can still compute. This is so because we have assumed to be able to extract the first row and column of A from the generators. Moreover, the recursion step still works as by Lemma 7.2 we are also able to compute a displacement identity for the Schur complement of A without the need to extract any further entries of A. Hence the same approach still applies. Note that for Lemma 7.2 to work throughout the whole recursive steps, we need A to allow a displacement structure $A = \Delta A - AZ$ in which Δ and Z are lower and upper triangular, respectively. This is not much of a problem for our cases of interest. The following algorithm is a concrete explanation of what we just described, and also handles certain special cases:

Algorithm 7.3. COMPUTATION OF A PLU DECOMPOSITION.

Input: *Matrices $D \in \mathbb{F}^{m\times m}, Z \in \mathbb{F}^{n\times n}, G \in \mathbb{F}^{m\times r}, H \in \mathbb{F}^{r\times n}$.*

Promise: *The matrices D and Z are diagonal and upper triangular, respectively. Moreover, $DA - AZ = GH$ for some $A \in \mathbb{F}^{m\times n}$.*

Output: *Matrices $P \in \mathbb{F}^{m\times m}, L \in \mathbb{F}^{m\times m}, U \in \mathbb{F}^{m\times n}$ such that P is a permutation matrix, L and U are lower and upper triangular, respectively, and $A = PLU$.*

(0) *Initialize the matrices L and U by all-zero entries.*

(1) *Recover from the generator the first column of A, identified as A_1.*

(2) *If A_1 is the zero vector, set the first column of L to the unit vector $(1, 0, \ldots, 0)^\top$, extract the first row of A and put it on the first row of U. Then proceed recursively with the first row and column of D and Z as well as the first row and the first column of H removed. Otherwise, follow the steps below.*

(3) *Find a nonzero entry of A_1, say at the position $(k, 1)$. Let $\mathbb{P}_1 \in \mathbb{F}^{m\times m}$ be the identity matrix.*

(4) *If $k \neq 1$, then permute the first and the k^{th} rows of G and P_1.*

(5) *Extract the k^{th} column of A. Assume the partitioning $\left(\begin{smallmatrix} a_{11} & A_{12} \\ A_{21} & \star \end{smallmatrix}\right)$ for the matrix $P_1 A$, where a_{11} is in fact the nonzero entry of A at position $(k, 1)$ moved to the top left corner after the permutation.*

(6) *Store* $(1, A_{12}/a_{11})^\top$ *as the first column of* L *and* (a_{11}, A_{12}) *as the first row of* U.

(7) *Apply Lemma 7.2 to compute the generators* $G_2 \in \mathbb{F}^{(m-1)\times r}$ *and* $H_2 \in \mathbb{F}^{r\times(n-1)}$ *of the Schur complement of* $P_1 A$ *with respect to* a_{11}. *Then proceed recursively with the first row and column of* D *and* Z *removed and using the generators* G_2 *and* H_2 *just obtained. This gives us the rest of the entries of* L *and* U. *Update the permutation matrix* P *obtained from the recursion by composing it with* P_1.

Remark. The requirements of the above algorithm are rather strong. First of all, we need the matrices D and Z be triangular to satisfy the conditions of Lemma 7.2. Moreover, as in the case of the Gaussian elimination, the algorithm occasionally requires a change of coordinates (reflected as a permutation of the rows of A) to guarantee the existence of the Schur complement. However, the matrix is no longer directly accessible to us. Instead, we only have a displacement equation $DA - AZ = GH$ to exploit[1]. Note that a permutation of the rows of A implies a similar permutation on the rows of AZ, and we are also able to explicitly permute the rows of G and hence GH. However, permuting the rows of A does not necessarily imply the same effect on DA, unless A and D commute. Thus in order to keep the displacement equation satisfied for the *permuted versions* as well, the algorithm further requires D to be diagonal. Fortunately, our special applications satisfy these strong requirements.

What is the running time of the algorithm? Without considering the time needed to extract a row or column of A from its generators, the algorithm takes $O(mr(m+n))$ time. That is because the computation involved in the application of Lemma 7.2 is straightforward and can be carried out using $O(r(m+n))$ operations. That is in fact the time needed for each level of recursion. As the depth of the recursion is at most m (in fact $\min\{m,n\}$), we get the desired bound $O(mr(m+n))$. This will coincide with the actual running time of the algorithm if we can manage to perform the row/column extraction phase in time $O(r(m+n))$. If r happens to be low enough, this will make our algorithm much more efficient than Gaussian elimination. However, the space requirement of the above algorithm is considerably high, as it generates a whole PLU decomposition while we are only interested in a nonzero kernel element. The following lemma will help us develop a more space efficient version of the algorithm:

Lemma 7.4. *Let* $A \in \mathbb{F}^{m\times n}$ *be partitioned as*

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix},$$

where $A_{11} \in \mathbb{F}^{i\times i}$, $A_{12} \in \mathbb{F}^{i\times(n-i)}$, $A_{21} \in \mathbb{F}^{(m-i)\times i}$, *and* $A_{22} \in \mathbb{F}^{(m-i)\times(n-i)}$, *for some* $1 \le i < \min\{m,n\}$, *and suppose that* A_{11} *is invertible. Further, let* $\tilde{A} := \left(\begin{smallmatrix} A \\ I_n \end{smallmatrix}\right)$, *where* I_n *is the* $n \times n$ *identity matrix. Denote by*

[1] Caveat: Initially the matrix A is directly accessible to us, but this is not the case for the recursive steps, when we do not explicitly compute the Schur complements.

$c := (c_1, \ldots, c_{m+n-i})^\top$ the first column of the Schur complement of $\tilde{A}$ with respect to A_{11}, and suppose that $c_1 = \cdots = c_{m-i} = 0$. Then the vector $c' := (c_{m-i+1}, \ldots, c_m, 1, \underbrace{0, \ldots, 0}_{n-i-1})^\top$ is in the right kernel of A.

Proof. Consider the partitioning of $\tilde{A}$ as

$$\tilde{A} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \\ I_i & \mathbf{0} \\ \mathbf{0} & I_{n-i} \end{pmatrix}.$$

Then by definition, the Schur complement of $\tilde{A}$ with respect to A_{11}, denoted as A_2, will be

$$A_2 = \begin{pmatrix} A_{22} \\ \mathbf{0} \\ I_{n-i} \end{pmatrix} - \begin{pmatrix} A_{21} \\ I_i \\ \mathbf{0} \end{pmatrix} \cdot A_{11}^{-1} \cdot A_{12} = \begin{pmatrix} A_{22} - A_{21}A_{11}^{-1}A_{12} \\ -A_{11}^{-1}A_{12} \\ I_{n-i} \end{pmatrix},$$

whose first column is c. It is worthwhile to remember that the first $m-i$ rows of the Schur complement of $\tilde{A}$ coincide with the Schur complement of A. The assumption about the vector c implies that the first column of $A_{22} - A_{21}A_{11}^{-1}A_{12}$ is zero. Observe that c' is the first column of the matrix $\left(\begin{smallmatrix} -A_{11}^{-1}A_{12} \\ I_{n-i} \end{smallmatrix}\right)$. On the other hand we have that

$$A \cdot \begin{pmatrix} -A_{11}^{-1}A_{12} \\ I_{n-i} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} -A_{11}^{-1}A_{12} \\ I_{n-i} \end{pmatrix} = \begin{pmatrix} \mathbf{0} \\ A_{22} - A_{21}A_{11}^{-1}A_{12} \end{pmatrix}.$$

Now restriction of both sides to the first column gives us $Ac' = 0$, as claimed. □

Note that any nonzero vector in the right kernel of a matrix defines a *linear dependence* on the columns of the matrix, and vice versa. In fact, what the above lemma says is that if A_{11} is nonsingular (implying that the leftmost i columns of A and $\tilde{A}$ are linearly independent), then the leftmost $i+1^{\text{st}}$ columns of A will be linearly dependent if the first column of the Schur complement of A with respect to A_{11} is zero.

Now using this lemma it is possible to modify the PLU decomposition algorithm so that it directly computes a nonzero kernel element, without explicitly computing a PLU decomposition first. We can apply a similar method as in Algorithm 7.5 to compute successive Schur complements until we find one whose leftmost column is zero. Then we can apply the lemma to obtain a nonzero kernel element.

However, in order to be able to apply Lemma 7.4 we need an even stronger condition than before on the structure of A. Namely, we need to work on the matrix $\tilde{A} := \left(\begin{smallmatrix} A \\ I_n \end{smallmatrix}\right)$ rather than A itself, and we need $\tilde{A}$ to have a nice displacement structure as well (so that the underlying ideas of our initial algorithm can still be applicable)! More clearly, suppose that $A \in \mathbb{F}^{m \times n}$ has low displacement rank r_A with respect to $\nabla_{D,Z}$ and there exists a matrix $B \in \mathbb{F}^{n \times n}$ such that

$B - Z$ has low (nonzero) rank r_B. This means that we have

$$DA - AZ = G_1 H_1, \qquad B - Z = G_2 H_2$$

for some $G_1 \in \mathbb{F}^{m \times r_A}$, $H_1 \in \mathbb{F}^{r_A \times n}$, $G_2 \in \mathbb{F}^{n \times r_B}$, and $H_2 \in \mathbb{F}^{r_B \times n}$. Then $\tilde{A}$ will admit the following decomposition:

$$\begin{pmatrix} D & \mathbf{0} \\ \mathbf{0} & B \end{pmatrix} \cdot \begin{pmatrix} A \\ I_n \end{pmatrix} - \begin{pmatrix} A \\ I_n \end{pmatrix} \cdot Z = \begin{pmatrix} G_1 & \mathbf{0} \\ \mathbf{0} & G_2 \end{pmatrix} \cdot \begin{pmatrix} H_1 \\ H_2 \end{pmatrix},$$

which satisfies our requirements.

Now having this further assumption we are able to slightly modify Algorithm 7.5 so that it directly computes a kernel element. Here is a more detailed account:

Algorithm 7.5. COMPUTATION OF A NONZERO RIGHT KERNEL ELEMENT.

Input: *Matrices* $D \in \mathbb{F}^{m \times m}, B \in \mathbb{F}^{n \times n}, Z \in \mathbb{F}^{n \times n}, G_1 \in \mathbb{F}^{m \times r}, H_1 \in \mathbb{F}^{r \times n}, G_2 \in \mathbb{F}^{n \times r'}, H_2 \in \mathbb{F}^{r' \times n}$.

Promise: *The matrices D, Z, and B are diagonal, upper triangular, and lower triangular, respectively, and $m < n$. Moreover, $D' := \left(\begin{smallmatrix} D & \mathbf{0} \\ \mathbf{0} & B \end{smallmatrix}\right) \in \mathbb{F}^{(m+n)\times(m+n)}$ and $\tilde{A} := \left(\begin{smallmatrix} A \\ I_n \end{smallmatrix}\right) \in \mathbb{F}^{(m+n)\times n}$ have the property that $D'\tilde{A} - \tilde{A}Z = GH$ for some $A \in \mathbb{F}^{m \times n}$, where $G := \left(\begin{smallmatrix} G_1 & \mathbf{0} \\ \mathbf{0} & G_2 \end{smallmatrix}\right)$, and $H := \left(\begin{smallmatrix} H_1 \\ H_2 \end{smallmatrix}\right)$.*

Output: *A nonzero vector $x \in \mathbb{F}^{n \times 1}$ such that $Ax = \mathbf{0}$.*

(0) *Let $i := 0$. Initialize the vector x with the unit vector $(1, 0, \ldots, 0)^\top$.*

(1) *Recover from the generator the first column of $\tilde{A}$, identified as $\tilde{A}_1$.*

(2) *Find the first occurrence of a nonzero entry in $\tilde{A}_1$, say at the k^{th} position. If no such entry exists or $k > m - i$, then return.*

(3) *Extract the k^{th} row of $\tilde{A}$. If $k \neq 1$, then permute the first and the k^{th} rows of G. Denote the matrix $\tilde{A}$ with its first and the k^{th} rows interchanged by $\tilde{A}$ again.*

(4) *Now assume the partitioning $\left(\begin{smallmatrix} \tilde{a}_{11} & \tilde{A}_{12} \\ \tilde{A}_{21} & \star \end{smallmatrix}\right)$ on the matrix $\tilde{A}$, where $\tilde{a}_{11}$ is the nonzero entry at the top left position.*

(5) *By applying Lemma 7.2, replace the generators G and H by generators of the Schur complement of $\tilde{A}$ with respect to $\tilde{a}_{11}$. Remove the first rows and columns of D' and Z, increase i by one and go to step* (1).

Provided efficient means of row/column extraction from the generators, the running time of this modified version of the algorithm is $O(rmn)$, as before. But this algorithm is more space efficient.

Remark 1. Note that the algorithm goes into at most m iterations, meaning that at most it attempts to compute the Schur complement with respect to the upper $m \times m$ submatrix of $\tilde{A}$, but not a larger one. In particular, such a submatrix never *reaches* the area corresponding to B. Thus in this case, Lemma 7.2 does not enforce any conditions on the matrix B. In particular B need not be triangular. Moreover observe that D being diagonal is a sufficient condition for the matrices D' and $\tilde{A}$ to commute, because of the identity component of $\tilde{A}$.

7.5. Row and Column Extraction

Recall that a major subroutine needed by our algorithms is that of efficiently computing the first row and column of a matrix that is implicitly given by its generators. Giving a general answer to this problem is difficult, as it highly depends on the particular displacement structure of the matrix. However, efficient algorithms for certain special cases exist, and here we only consider one prototypical case.

Suppose that the matrix $A \in \mathbb{F}^{(m+n)\times n}$ has the displacement structure

$$\begin{pmatrix} D & \mathbf{0} \\ \mathbf{0} & B \end{pmatrix} A - AZ = GH, \tag{7.3}$$

where $D \in \mathbb{F}^{m\times m}$ is a diagonal matrix given by $\mathrm{diag}(d_1, \ldots, d_m)$ and $Z, B \in \mathbb{F}^{n\times n}$ are shift and rotation (circular shift) matrices, respectively. That is,

$$Z := \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ 0 & 0 & 0 & \cdots & 0 & 0 \end{pmatrix}, \quad B := \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 \\ 0 & 0 & 0 & \cdots & 0 & 1 \\ 1 & 0 & 0 & \cdots & 0 & 0 \end{pmatrix}. \tag{7.4}$$

For convenience, we also assume that $\prod_{i=1}^{m} d_i \neq 0$, that is, all diagonal entries of D are nonzero. Note that the matrix B given as above is not triangular. However, by Remark 1 this is not problematic. Denote the first row and column of A by $x := (x_1, \ldots, x_n)$ and $y := (y_1 = x_1, y_2, \ldots, y_{m+n})^\top$, respectively. Considering our particular

choices of the matrices D, Z, and B, we can write GH as

$$GH = \begin{pmatrix} D & \mathbf{0} \\ \mathbf{0} & B \end{pmatrix} A - AZ = \begin{pmatrix} d_1x_1 & d_1x_2 - x_1 & \cdots & d_1x_n - x_{n-1} \\ d_2y_2 & \cdots & \cdots & \cdots \\ \vdots & \cdots & \cdots & \cdots \\ d_my_m & & & \\ y_{m+2} & \cdots & \cdots & \cdots \\ \vdots & \cdots & \cdots & \cdots \\ y_{m+n} & & & \\ y_{m+1} & \cdots & \cdots & \cdots \end{pmatrix}.$$

This observation leads to the following algorithm for reconstruction of the first row and column of A (namely, the vectors x and y):

Algorithm 7.6. RECOVERY OF THE FIRST ROW AND COLUMN FROM GENERATORS.

Input: *Matrices $D \in \mathbb{F}^{m\times m}, G \in \mathbb{F}^{(m+n)\times r}, H \in \mathbb{F}^{r\times n}$.*

Promise: *The matrix D is diagonal. Moreover, D, G, and H satisfy the displacement structure* (7.3)*, for some matrix $A \in \mathbb{F}^{(m+n)\times n}$ and matrices Z and B as in* (7.4)*.*

Output: *The first row and the first column of A (i.e., the vectors x and y).*

(1) *Compute the first row $(r_1, \ldots, r_n)$ and the first column $(c_1, \ldots, c_{m+n})^\top$ of the product GH.*

(2) *For $i := 1, \ldots, m$ set $y_i := c_i/d_i$. Set $x_1 := y_1$.*

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

(4) *Set $y_{m+1} := c_{m+n}$.*

(5) *For $i := 2, \ldots, n$ set $x_i := (r_i + x_{i-1})/d_1$.*

(6) *Output row $(x_1, \ldots, x_n)$ and column $(y_1, \ldots, y_{m+n})^\top$.*

The running time of this algorithm is dominated by the first step, when the first row and the first column of the product GH is computed (Recall that G is an $(m+n) \times r$ matrix and the dimension of H is $r \times n$). Using the straightforward algorithm for doing so requires at most $r(m+n)$ operations for computation the first column and at most rn operations for that of the first row. This makes the total running time of the algorithm to be at most $O(r(m+n))$.

Combining this algorithm with the ones described in the last section gives us a full procedure to compute a nonzero vector in the right kernel of an $m \times n$ matrix with an overall running time of $O(r(m+n))$, where r is the displacement rank of the matrix. Indeed the given matrix has to satisfy certain conditions required by our algorithms, but fortunately this is the case for our particular decoding problems. When r is sufficiently small (and in particular, constant), we can get a major improvement upon the familiar Gaussian elimination algorithm whose running time is $O(m^2 n)$.

Further Reading

[1] V. Olshevksy and A. Shokrollahi. The displacement method in coding theory. Preprint available online at `http://algo.epfl.ch/contents/output/pubs/structmat_3.ps.gz`.

[2] V. Olshevksy and A. Shokrollahi. A displacement structure approach to decoding algebraic geometric codes. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing (STOC'99)*, pages 235–244, 1999.

[3] T. Kailath and A. H. Sayed. *Fast Reliable Algorithms for Matrices With Structure (Advances in Design and Control)*. Society for Industrial and Applied Math, 1999.