On Implementing Signature-based Gröbner Basis Algorithms Using Linear Algebraic Routines from M4RI

Yao Sun, Dongdai Lin
SKLOIS, Institute of Information Engineering, CAS, (China)

Dingkang Wang
KLMM, Academy of Mathematics and Systems Science, CAS, (China)

dwang@mmrc.iss.ac.cn

Abstract

Gröbner bases, proposed by Buchberger in 1965 [5], have been proven to be very useful in many aspects of algebra. Faugère introduced the concept of signatures for polynomials and presented the famous F5 algorithm [9]. Since then, signature-based algorithms have been widely investigated, and several variants of F5 have been presented, including F5C [7], extended F5 [13], F5 with revised criterion [4], and RB [8]. Gao et al. proposed another signature based algorithm G2V [11] in a different way from F5, and GVW[12] is an extended version of G2V. The authors also studied generalized criteria and signature-based algorithms in solvable polynomial algebra in [15, 16].

For implementations of signature-based algorithms, Roune and Stillman efficiently implemented GVW and AP without using linear algebra [14]. Faugère mentioned a matrix F5 in [10]. An F5 algorithm in F4 style was described in more detail by Albrecht and Perry [1].

In signature-based algorithms, each polynomial is assigned a signature, and polynomials can only be reduced by polynomials with smaller signatures. When implementing signature-based algorithms using linear algebra, such as [1], rows of the constructed matrices are also assigned with signatures. These matrices can only be eliminated from one side due to the constraints of signatures, i.e. rows can only be reduced by rows with smaller signatures. However, most public libraries on linear algebra do not provide routines for such one-side elimination, such that most public libraries cannot be applied to the implementations of signature-based algorithms directly.

In this talk, we present a method of rotating rows during the elimination, to ensure rows with larger signatures can always be reduced by rows with smaller signatures. Our method only needs to revise the swapping procedure during the elimination, and can be easily applied to most public libraries on linear algebra. We have applied our method to the M4RI package [3], and implemented the GVW algorithm by using the modified routines over the finite field $GF(2)$. Due to the efficient routines modified from M4RI, our implemented GVW algorithm is more efficient than some of Gröbner basis implementations on public softwares.

Our method of rotating rows can be illustrated by the following example. Let $A$ be a matrix with entries in $F_2$. Assume $A$ has the following form:

$$
\begin{pmatrix}
S_1 & 0 & 1 & * & * & * \\
S_2 & 0 & 0 & 1 & * & * \\
S_3 & 0 & 1 & * & * & * \\
S_4 & 0 & 1 & * & * & * \\
S_5 & 1 & * & * & * & * \\
S_6 & 1 & * & * & * & * 
\end{pmatrix}
$$

where “*” may be 1 or 0, $S_i$ is the signature of each row, and we assume $S_1 \prec_s S_2 \prec_s \cdots \prec_s S_6$.

To reduce $A$ to row-echelon form, we first find the pivot entry in the first column. We must search the pivot entry from top to bottom (i.e. from lower signatures to higher signatures). Then we find the entry at row 5 and col 1 is a pivot. If we use general methods of elimination, we need to swap row 1 and row 5 directly, and clear entries at column 1 by the row with signature $S_5$. Next, when doing elimination in the second column, the row with signature $S_4$ is selected as pivot row, and needs to eliminate other rows. However, this will leads to errors in signature-based algorithms, because the row with signature $S_1$ has a smaller signature than
S₄ and cannot be eliminated by the row with signature S₄. So we cannot swap row 1 and row 5 directly.

To make further eliminations correct, we swap row 1 and row 5 in a special manner. First, we pick up the row 5 with signature S₅. Second, we move rows 4, 3, 2, and 1 to rows 5, 4, 3, and 2 respectively. At last, we put the row with signature S₅ at row 1. After this swap, matrix A becomes the following form.

\[
\begin{bmatrix}
1 & * & * & * & * & * \\
0 & 1 & * & * & * & * \\
0 & 0 & 1 & * & * & * \\
0 & 0 & 1 & * & * & * \\
0 & 1 & * & * & * & * \\
1 & * & * & * & * & * \\
\end{bmatrix}
\]

Next, we use the row with S₅ to clear all entries at column 1 below this row, and then column 1 is done. For column 2, we find pivots from rows with S₁, ..., S₄ and S₆, and repeat the above processes. Elimination terminates when the matrix becomes an upper triangular form.

This swap makes eliminations correct in signature-based algorithm for the following reasons. On one hand, since pivot rows (e.g. row of S₅) are finding from low signatures to high signatures, rows with smaller signature (e.g. rows of S₁, ..., S₄) cannot be reduced by pivot rows (e.g. row of S₅). On the other hand, after swaps, rows below pivot rows (e.g. rows of S₁, ..., S₄ and S₆) are still in an increasing order on signatures.

Using this special swap, the echelon form of A is in an upper triangular form, such that divide-and-conquer methods of PLE decomposition [2] can be used, and hence, the eliminations can be speeded up significantly.

In our implementation, we modify many subroutines of mzd_ple() in M4RI library to use this swap. The new function with the special swap is called gvw_ple(). We compare the efficiency of mzd_ple() and gvw_ple() in Table 1. Examples with density ≈ 50% are generated randomly by routines from M4RI. Since the densities of matrices in Gröbner basis computations are usually very small, we also generate some randomized matrices with density ≈ 3%. The first column in Table 1 is the size of matrices, and the timings in this table are given by seconds. All matrices are generated over GF(2).

<table>
<thead>
<tr>
<th>Tests</th>
<th>density ≈ 50% mzd_ple()</th>
<th>gvw_ple()</th>
<th>density ≈ 3% mzd_ple()</th>
<th>gvw_ple()</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,000 × 10,000</td>
<td>0.378</td>
<td>0.382</td>
<td>0.345</td>
<td>0.354</td>
</tr>
<tr>
<td>10,000 × 30,000</td>
<td>1.342</td>
<td>1.301</td>
<td>1.268</td>
<td>1.262</td>
</tr>
<tr>
<td>30,000 × 10,000</td>
<td>1.432</td>
<td>1.443</td>
<td>1.403</td>
<td>1.418</td>
</tr>
<tr>
<td>30,000 × 30,000</td>
<td>7.661</td>
<td>7.655</td>
<td>7.604</td>
<td>7.577</td>
</tr>
<tr>
<td>30,000 × 60,000</td>
<td>18.684</td>
<td>18.671</td>
<td>18.651</td>
<td>18.634</td>
</tr>
<tr>
<td>60,000 × 30,000</td>
<td>19.396</td>
<td>19.296</td>
<td>19.282</td>
<td>19.298</td>
</tr>
<tr>
<td>60,000 × 60,000</td>
<td>58.373</td>
<td>58.636</td>
<td>54.509</td>
<td>54.263</td>
</tr>
<tr>
<td>60,000 × 100,000</td>
<td>123.321</td>
<td>123.298</td>
<td>119.479</td>
<td>122.523</td>
</tr>
<tr>
<td>100,000 × 60,000</td>
<td>119.991</td>
<td>118.388</td>
<td>108.565</td>
<td>108.501</td>
</tr>
<tr>
<td>100,000 × 100,000</td>
<td>266.817</td>
<td>267.191</td>
<td>237.401</td>
<td>237.560</td>
</tr>
<tr>
<td>150,000 × 150,000</td>
<td>817.682</td>
<td>817.750</td>
<td>700.032</td>
<td>700.781</td>
</tr>
</tbody>
</table>

Table 1: mzd_ple() vs gvw_ple()

From the above table, we can see the function mzd_ple() and gvw_ple() almost have the same efficiency.

In Table 2, we compare our implementation of GVW with some intrinsic implementations on public softwares, including Gröbner basis functions on Maple (version 17, setting “method = fgb”), Singular (version 3-1-6), and Magma (version 2.12-16)\(^1\), and the computing times in seconds are listed. In the column of Exam., n × n means that the input polynomial system has n polynomials with n variables. These square polynomial systems were generated by Courtois in [6]. The Computer we used is MacBook Pro with 2.6 GHz Intel Core i7, 16 GB memory.

\(^1\)Magma 2.12-16 is an old version.
<table>
<thead>
<tr>
<th>Exam.</th>
<th>Maple</th>
<th>Singular</th>
<th>Magma</th>
<th>GVW</th>
</tr>
</thead>
<tbody>
<tr>
<td>$16 \times 16$</td>
<td>4.088</td>
<td>5.210</td>
<td>0.484</td>
<td>0.560</td>
</tr>
<tr>
<td>$17 \times 17$</td>
<td>9.891</td>
<td>12.886</td>
<td>0.874</td>
<td>0.893</td>
</tr>
<tr>
<td>$18 \times 18$</td>
<td>22.340</td>
<td>31.590</td>
<td>1.513</td>
<td>1.556</td>
</tr>
<tr>
<td>$19 \times 19$</td>
<td>48.314</td>
<td>84.771</td>
<td>2.792</td>
<td>2.742</td>
</tr>
<tr>
<td>$20 \times 20$</td>
<td>107.064</td>
<td>265.325</td>
<td>5.226</td>
<td>4.676</td>
</tr>
<tr>
<td>$21 \times 21$</td>
<td>218.479</td>
<td>724.886</td>
<td>10.468</td>
<td>14.991</td>
</tr>
<tr>
<td>$22 \times 22$</td>
<td>839.067</td>
<td>$\geq 1h$</td>
<td>37.144</td>
<td>28.947</td>
</tr>
</tbody>
</table>

Table 2: Maple, Singular and Magma vs M-GVW

From the above table, we can see that, due to the efficiency of routines from M4RI, our implementation of M-GVW is more efficient than some of functions from existing public softwares. However, since the matrices in large polynomial systems become quite sparse, our implementation may not perform very good for large systems at present.

**Keywords**

Gröbner basis, linear algebra, implementation.

**References**