Fraction-Free Algorithms for Linear and Polynomial Equations

George C Nakos[1], Peter R Turner[1] and Robert M Williams[2]
1. Mathematics Department, US Naval Academy, Annapolis, MD 21402 and
2. Naval Air Warfare Center Aircraft Division, Patuxent River Naval Air Station, MD

Abstract

This paper extends the ideas behind Bareiss's fraction-free Gauss elimination algorithm in a number of directions. First, in the realm of linear algebra, algorithms are presented for fraction-free LU "factorization" of a matrix and for fraction-free algorithms for both forward and back substitution. These algorithms are valid not just for integer computation but also for any matrix system where the entries are taken from a unique factorization domain such as a polynomial ring. The second part of the paper introduces the application of the fraction-free formulation to resultant algorithms for solving systems of polynomial equations. In particular, the use of fraction-free polynomial arithmetic and triangularization algorithms in computing the Dixon resultant of a polynomial system is discussed.

1 Introduction

Recent work in various applications including robot control and threat analysis has resulted in renewed importance in developing efficient algorithms for the solution of systems of polynomial equations. There are techniques based on computing the resultant of the system. The Dixon resultant [5], [9], [13] is one such resultant which is obtained from the original polynomial system. The use of the Dixon resultant breaks down into two steps: the computation of the Dixon matrix and then finding its determinant, the Dixon polynomial. This process is summarized in Section 3. It is described in much greater detail in [13].

These resultant approaches to polynomial equations involve significant linear algebra subproblems which are *not* standard *numerical* linear algebra problems. The "arithmetic" that is needed is usually algebraic in nature and must be handled *exactly*. In the context of the Dixon resultant, for example the arithmetic is taking place in a polynomial ring. All the usual difficulties of integer arithmetic re-emerge here — only much exaggerated. The problem of range growth manifests itself both in increased polynomial degrees and in the practicalities of representing such polynomials efficiently. The solution of polynomial systems therefore brings with it the requirement for efficient techniques for the linear algebra subproblems in an exact (or *symbolic*) computing environment. It is easy to modify the standard Gauss elimination, GE, algorithm to avoid all division operations but this leads to a very rapid growth in the dynamic range. This was discussed in [15] and, in the context of adaptive beamforming, in [10] and [11]. One way of reducing this effect dramatically is the use of *fraction-free* (or *integer-preserving*) algorithms.

Perhaps the first description of such an algorithm was due to Bareiss [1]. A simplified version of Bareiss's algorithm was rediscovered in Turner [14] using a comparison of floating-point and integer GE to identify certain factors generated by the elimination process. This is especially useful for resultant computation. These algorithms remain valid in any *unique factorization domain*, UFD, such a ring of polynomials over any field — including, in particular, real or complex polynomials. One of the principal advantages of this algorithm is that the final matrix entry at the conclusion of the elimination *is* the determinant of the original matrix.

Section 2 briefly reviews the ideas of fraction-free Gauss elimination and then extends these ideas to a diagonalization, or Gauss-Jordan, algorithm. In numerical linear algebra, it is well-known that the LU factorization is much more efficient than simple GE in case multiple systems are to be solved with the same coefficient matrix. It is not initially obvious how to achieve this in a fraction-free setting. However, if the demand for a strict *factorization* is dropped in favor of achieving the same effect, then a simplification of the fraction-free GE algorithm achieves this. There are appropriate forward and back substitution algorithms to accompany this "factorization". These are all described, with examples, in Section 2.

In Section 3 we show how to combine the fraction-free algorithms with the Dixon resultant elimination to triangularize a system of polynomial equations. Often the solution set of the resulting "triangular system" *contains* all the solutions of the original system.

In Section 4 we point to some further investigations that should be pursued. In particular, the fraction-free GE algorithm generates a large number (approximately $N^2/2$) of minors of the original matrix. In the case of

the Dixon matrix these should yield much extra information as to whether roots of the polynomial system exist. One interesting possibility is that this extra information may be sufficient to eliminate the extra factors which are usually part of the Dixon polynomial.

2 Fraction-free algorithms for linear systems

This section is concerned with the fundamental ideas of fraction-free "arithmetic" for solving linear systems. We begin with a brief review of the fraction free Gauss elimination algorithm and then extend the ideas to the Gauss-Jordan algorithm and to a version of LU factorization. These algorithms are applicable in any unique factorization domain (UFD) and so we shall assume throughout this section that all matrix and vector entries are elements of a ring $\mathcal{R}$ which is a UFD. Typical examples are the integers $\mathbf{Z}$, and spaces of real or complex polynomials $\mathbf{R}[x]$, $\mathbf{C}[z]$. A *unique factorization domain* is an integral domain with one important additional property: every element (apart from zero and ± 1) can be uniquely written as a product of "primes". In $\mathbf{Z}$, these primes are just the conventional prime numbers; in $\mathbf{R}[x]$ the "primes" comprise the nonzero constants and monic irreducible polynomials; in $\mathbf{C}[z]$ they correspond to nonzero constants and the monic linear polynomials $z + \alpha$.

Any nonsingular $n \times n$ linear system of equations in $\mathcal{R}$ has a unique solution in the associated field of fractions which we shall denote by $\mathcal{Q}$. In the case where $\mathcal{R}$ is $\mathbf{Z}$, this field of fractions is just $\mathbf{Q}$, the set of rational numbers; for $\mathbf{R}[x]$, $\mathbf{C}[z]$ the corresponding fields consist of the rational functions over $\mathbf{R}$ or $\mathbf{C}$. For computation in a UFD $\mathcal{R}$ it is desirable to remove any unwanted factors as soon as possible to avoid unnecessary demands on storage and to simplify any subsequent "arithmetic". Keeping this *dynamic range* growth to a minimum, not only simplifies the computation itself but also makes the results of the computation much easier to use — for example in order to determine whether a particular system has a solution in $\mathcal{R}$ itself. For simplicity all our examples in this section use the integers $\mathbf{Z}$ as the underlying UFD. For the special context of integer computation, these ideas can be combined with the use of *Residue Number System*, RNS arithmetic [16]

2.1 Fraction-free Gauss elimination

We present a brief review of the basic fraction-free linear algebra routine, Bareiss's [1] "integer-preserving" Gauss elimination, sometimes known as the Gauss-Bareiss algorithm. The actual form of the algorithm we present is a simplified version which is described in more detail in [14]. This simplified form was derived from a comparison of *divisionless* integer GE with the usual floating-point (or *real*) arithmetic algorithm.

The essence of the algorithm can be seen by considering a 3×3 matrix. From this analysis it is observed that the result of divisionless GE on an initial matrix

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

is an upper triangular matrix

$$\begin{bmatrix} a & b & c \\ & ae - db & af - dc \\ & & a * \det A \end{bmatrix}$$

The key observations here are that the final element has the factor a and that if this factor is removed the diagonal will then consist precisely of the principal minors of increasing size. Furthermore with a larger matrix, *every* element below or to the right of the $3,3$ position would have this factor a at this stage since the computation in every such position is essentially the same. This factor is therefore removable from the entire active matrix. The same comment applies at each subsequent stage except that the common factor "moves" down the diagonal. The presence of these factors and their removal is apparent in the following algorithm written in its conventional three-loop form.

Algorithm 1 Fraction-Free Gauss Elimination (FFGE)

Input $n \times n$ matrix A and right-hand side vector $\mathbf{b}$
Compute

```
for i = 1 : n − 1
  for j = i + 1 : n
    b_j := a_ii b_j − a_ji b_i
    if i > 1 then b_j := b_j / a_{i−1,i−1}
    for k = i + 1 : n
      a_jk := a_ii a_jk − a_ji a_ik
      if i > 1 then a_jk := a_jk / a_{i−1,i−1}
    a_ji := 0
```

Output (modified) matrix A and right-hand side $\mathbf{b}$ for back substitution

Remark It is apparent that FFGE is not *division*-free. It is nonetheless *fraction-free* because of the known factors described earlier.

Example 1 *FFGE was applied to the following very simple system.*

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 2 & 3 & 4 \\ 3 & 3 & 3 & 4 \\ 9 & 8 & 7 & 6 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 10 \\ 11 \\ 13 \\ 30 \end{bmatrix}$$

The results of the three stages of elimination are, respectively

$$U = \begin{bmatrix} 1 & 2 & 3 & 4 & 10 \\ 0 & -2 & -3 & -4 & -9 \\ 0 & -3 & -6 & -8 & -17 \\ 0 & -10 & -20 & -30 & -60 \end{bmatrix}$$

$$U = \begin{bmatrix} 1 & 2 & 3 & 4 & 10 \\ 0 & -2 & -3 & -4 & -9 \\ 0 & 0 & 3 & 4 & 7 \\ 0 & 0 & 10 & 20 & 30 \end{bmatrix}$$

$$U = \begin{bmatrix} 1 & 2 & 3 & 4 & 10 \\ 0 & -2 & -3 & -4 & -9 \\ 0 & 0 & 3 & 4 & 7 \\ 0 & 0 & 0 & -10 & -10 \end{bmatrix}$$

Inspection of the final system immediately yields the solution $(1,1,1,1)^t$ for this example. Further, it is easy to check that the diagonal of the (first four columns of the) upper triangle contains the determinants of the principal minors of A. The reason why this algorithm works is precisely because of the *exact* factors which are generated so that the divisions will always produce exact integer results.

FFGE yields additional information. Temporarily, let A denote the *augmented* matrix including any right-hand side columns. The elements u_{jk} of the matrix U returned by the above algorithm are *all* determinants of minors of A. Specifically, u_{jk} is the determinant of the minor formed using rows $1, 2, \ldots, j$ and columns $1, \ldots, j-1, k$ of A. These additional minors are of potential interest and value in a numerical setting, but in a more general algebraic setting, much more information may be gleaned from them. For example, the Dixon resultant for a polynomial system is the determinant of a matrix of polynomial terms. This determinant must vanish identically, if the original system has a solution. These minors could perhaps be used to determine whether a subsystem has a solution and therefore to place conditions on parameters which necessarily must be satisfied for solutions of the full system. These issues are discussed further later.

Remarks

FFGE makes no allowance for pivoting. Therefore if, for example, the 2×2 principal minor is singular, then the process would fail after the first stage of the elimination. Both [1] and [14] advocate a simple pivoting strategy that is used only when necessary: if the pivot element is 0, then perform a row-interchange with the first (lower) row which has a nonzero entry in the pivot position. This is very easily incorporaetd into FFGE.

The removal of the common factors necessarily reduces the dynamic range requirement. A useful indication of the savings achieved can be obtained from a simple comparison of the final values for a_{nn} by the Algorithm FFGE and by the *divisionless* algorithm. This comparison was introduced in [16]. A simple characterization of this is obtained in terms of the matrix elements in the fraction-free algorithm. Let a_{ij} denote the values obtained from FFGE. Denote by a^D_{nn} the final value obtained by divisionless Gauss elimination for the n, n element of A. Then we have $a_{nn} = \det A$ and

$$a^D_{nn} = \left(a_{11}^{n-2} a_{22}^{n-3} \cdots a_{n-2,n-2}\right) \det A \qquad (1)$$

This growth factor is a realistic estimate of the saving in dynamic range achieved by the fraction-free algorithm. Even for the very small dimensional example above, this represents a saving of a factor of 24 in the magnitude of a_{44}.

For a 12×12 matrix with initial entries represented by 8-bit integers, we may conjecture a "typical" magnitude of initial entries around 16 (approximately the square-root of the initial range). Even if we assume that no growth in the diagonal entries the ratio $a^D_{nn}/\det A$ given by (1) is $\prod_{k=1}^{10} 16^k = 16^{55} = 2^{220}$. That is some 220 bits have been saved from the effective wordlength demanded by this dynamic range growth! Similarly, for a 12×12 polynomial matrix where each factor has degree at least 1, this saving amounts to removal of a factor of degree 55 from the final row!

2.2 Fraction-free Diagonalization (Gauss Jordan)

As with conventional real arithmetic, the Gauss elimination algorithm is easily modified to perform elimination *above* the diagonal as well as below. The result of this *Gauss Jordan*, GJ, algorithm (provided the original matrix is nonsingular) is a diagonal matrix. In a symbolic setting, the loss of (numerical) stability is not a factor since the computation is exact. Further the diagonal matrix which results is a multiple of the identity so that any right-hand side is just a scaled solution of the original system — or perhaps a multiple of the inverse of the original matrix. The only complication is the need to retain the "old" value of $a_{i-1,i-1}$ because its value will have been changed before it is used as a divisor.

Algorithm 2 Fraction-Free Gauss Jordan (FFGJ)

Input $n \times n$ matrix A and right-hand side vector $\mathbf{b}$
Compute

```
for i = 1 : n
  if i > 1 then d := a_{i-1,i-1}
  for j = 1 : n, j ≠ i
    b_j := a_ii b_j − a_ji b_i
    if i > 1 then b_j := b_j / d
    for k = 1 : n
      a_jk := a_ii a_jk − a_ji a_ik
```

if $i > 1$ then $a_{jk} := a_{jk}/d$

Output diagonal modified matrix A and right-hand side **b**

Remark It is important to note here that many redundant operations have been included for the sake of a simple algorithm description.

Example 2 *FFGJ applied to the same example as in the previous section.*

The output of the first two stages of FFGJ consisted of the arrays

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 10 \\ 0 & -2 & -3 & -4 & -9 \\ 0 & -3 & -6 & -8 & -17 \\ 0 & -10 & -20 & -30 & -60 \end{bmatrix};$$

$$\begin{bmatrix} -2 & 0 & 0 & 0 & -2 \\ 0 & -2 & -3 & -4 & -9 \\ 0 & 0 & 3 & 4 & 7 \\ 0 & 0 & 10 & 20 & 30 \end{bmatrix}$$

In the second stage of the elimination we see that the not only have we eliminated all off-diagonal elements in the second column but also the first diagonal entry has been modified before its use as a divisor. This is why the additional local (scalar) variable d is introduced. The final two stages produce

$$\begin{bmatrix} 3 & 0 & 0 & 0 & 3 \\ 0 & 3 & 0 & 0 & 3 \\ 0 & 0 & 3 & 4 & 7 \\ 0 & 0 & 0 & -10 & -10 \end{bmatrix};$$

$$\begin{bmatrix} -10 & 0 & 0 & 0 & -10 \\ 0 & -10 & 0 & 0 & -10 \\ 0 & 0 & -10 & 0 & -10 \\ 0 & 0 & 0 & -10 & -10 \end{bmatrix}$$

The solution of the original system is now apparent. Note that the diagonal entries are indeed just $\det A$, as predicted.

As a further example, FFGJ was used with the same matrix A and $B = I$. The final output is the 4×8 array

$$U = \begin{bmatrix} -10 & 0 & 0 & 0 & 10 & -10 & 0 & 0 \\ 0 & -10 & 0 & 0 & -10 & 20 & -10 & 0 \\ 0 & 0 & -10 & 0 & -4 & -10 & 20 & -4 \\ 0 & 0 & 0 & -10 & 3 & 0 & -10 & 3 \end{bmatrix}$$

and, as expected, $A * U(:, 5:8) = -10I$ or $(\det A) * I$. We see that A does *not* have an inverse in the integers but does, of course, in the appropriate field of fractions **Q**.

The greatest disadvantage in using FFGJ rather than the simpler FFGE algorithm for solving a linear system results from the increased arithmetic operation count. Just as with their real arithmetic counterparts, there is approximately a 50% increase in the operation count for FFGJ relative to FFGE. The rational solution computed by the fraction-free Gauss-Jordan algorithm is essentially just that which would be found using Cramer's rule - except that all the various determinants are computed using the same row operations. In a sense therefore FFGJ could be viewed as an "efficient" Cramer's rule for symbolic computation. The saving is, of course, enormous — from $O([n+1]!)$ to $O(n^3)$ operations. (For even moderate n this represents a reduction from "lifetimes" to "seconds".)

2.3 Fraction-free LU Decomposition of a matrix

The first observation to be made about the fraction-free LU decomposition algorithm that is the subject of this section is that it is *not* a true factorization of the original matrix. The primary objective of the conventional LU decomposition is that several linear systems (with the same coefficient matrix) may be solved without the need to recompute the full Gauss elimination. By storing the lower and upper triangular factors subsequent systems are solved at the cost of only a forward and back substitution procedure which is only $O(n^2)$ rather than $O(n^3)$ operations.

The algorithm designated fraction-free LU decomposition, FFLU, below achieves these ends in almost exactly the manner of the real-arithmetic algorithm even though the "factors" do not have the original matrix as their product. In the standard floating-point algorithm, elements below the diagonal are simply overwritten by the multipliers that are used in the corresponding stage of the elimination. In the fraction-free setting, these multipliers and the overwriting are implicit.

Algorithm 3 Fraction-free LU decomposition, FFLU

Input $n \times n$ matrix A

Compute

for $i = 1 : n-1$
  for $j = i+1 : n$
    for $k = i+1 : n$
      $a_{jk} := a_{ii}a_{jk} - a_{ji}a_{ik}$
      if $i > 1$ then $a_{jk} := a_{jk}/a_{i-1,i-1}$

Output (modified) matrix A

The lower-triangular and upper-triangular parts of the output matrix are the LU "factors". Both pieces have the same diagonal.

Example 3 *FFLU decomposition of the matrix A used in the earlier examples yields*

$$\mathsf{LU} = \begin{matrix} 1 & 2 & 3 & 4 \\ 2 & -2 & -3 & -4 \\ 3 & -3 & 3 & 4 \\ 9 & -10 & 10 & -10 \end{matrix}$$

in which we can see (from careful comparison with Example 1) almost the complete history of the FFGE computation.

The key question is how this can be used for solving a linear system since certainly the product of the "factors"

$$\mathsf{L} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & -2 & 0 & 0 \\ 3 & -3 & 3 & 0 \\ 9 & -10 & 10 & -10 \end{bmatrix}$$

$$\mathsf{U} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 0 & -2 & -3 & -4 \\ 0 & 0 & 3 & 4 \\ 0 & 0 & 0 & -10 \end{bmatrix}$$

is very different from A.

It should be noted here that in fact these "factors" are closely related to a true factorization of A. Recently, Corless [4] has extended the theoretical aspects of this algorithm so that a true factorization is achieved. The usefulness of FFLU becomes apparent from its subsequent use with forward and back substitution. Pivoting is again easily achieved when necessary.

2.3.1 Forward and Back Substitution

To take advantage of this decomposition in the solution of linear systems of equations both forward and back substitution algorithms are needed. The forward substitution must of course take account of any row interchanges and of any divisors which have been used during the elimination phase. Again, however, all the necessary information is contained in the modified matrix.

Throughout the discussion of these algorithms we shall assume that the triangular matrices are the result of applying FFLU to an original matrix. At the beginning of the forward substitution, the right-hand side is unmodified. Algorithm 4 is written for a single right-hand side vector — its generalization to multiple columns is obvious.

Algorithm 4 Fraction-free Forward substitution, FFFS

Input $n \times n$ matrix A (the output from FFLU) and right-hand side $\mathbf{b}$

Compute

for $i = 1 : n-1$
  for $j = i+1 : n$
    $b_j := a_{ii}b_j - a_{ji}b_i$
    if $i > 1$ then $b_j := b_j / a_{i-1,i-1}$

Output (modified) right-hand side $\mathbf{b}$

Example 4 *FFFS applied to our usual example with* LU *being the result obtained in Example 4 generates the modified right hand side* $(10, -9, 7, -10)'$ *which is just the same as was obtained in Example 1 using FFGE. However, if we now wish to solve the system with the right-hand side* $\mathbf{c} = (-10, 11, -13, 30)'$ *we can use FFFS again without having to recompute FFLU first. This yields* $(-10, 31, 59, 190)'$.

The only additional cost is the arithmetic for this forward substitution which is $O(n^2)$ operations. The savings obtained are precisely as they are for the conventional (floating-point) LU factorization relative to its equivalent Gauss elimination.

Even for these very small systems, the operation counts tell the story quite plainly. In the table below, we show MATLAB's reported operation counts for the different instructions. Since MATLAB is performing double precision real arithmetic and counts *flops* (floating-point operations) these give a genuine idea of the relative arithmetic workloads of the different algorithms for a 4×4 system neglecting any overhead for loop control.

TABLE 1 Comparison of operation counts for FFGE and FFLU with FFFS for solving two 4 × 4 systems

MATLAB command	Operation count	
FFGE(A,b)	108	
FFGE(A,c)	108	
Total for two systems		216
LU=FFLU(A)	81	
FFFS(LU,b)	31	
FFFS(LU,c)	31	
Total for two systems		143

Even at this small dimension and for just two systems, the saving resulting from using the LU decomposition approach is apparent. Of course these total do not include the back substitution but this is the same for either approach since after FFGE or FFLU and FFFS the remaining task is identical.

Algorithm 5, FFBS describes the fraction-free back substitution which is needed to complete the solution process. The objective here is to retain exact computation as long as possible. In our examples we will see that integer arithmetic is preserved up to a final division. In order to retain exact computation, the output from FFBS has two parts consisting of a scaled solution vector and the determinant of the original matrix — which is the scaling factor. That is the true solution is the scaled solution divided by the determinant.

Algorithm 5 Fraction-free Back Substitution, FFBS

Input $n \times n$ matrix A (the output from FFLU) and right-hand side $\mathbf{r}$ (the output from FFFS)

Compute

$\mathbf{s} := \mathbf{r}$
$d := a_{nn}$
for $i = n-1 : -1 : 1$
  $s_i := d * s_i$
  for $j = i+1 : n$
    $s_i := s_i - a_{ij} s_j$
  $s_i := s_i / a_{ii}$

Output scaled solution $\mathbf{s}$ and d (which is $\det A$)

Example 5 *The complete solution of our original example system can be achieved using the computation sequence:*

```
LU=fflu(A); R=fffs(LU,b); [S,d]=ffbs(LU,R);
```

The output of the first two commands we have already seen. The last yields $\mathsf{S} = (-10,-10,-10,-10)'$ and $\mathsf{d} = -10$ from which the integer solution is easily obtained. The second system can be solved using `R=fffs(LU,c); [S,d]=ffbs(LU,R);` which yields the scaled solution vector $\mathsf{S} = (-210, 450, -450, 190)'$ again with $\mathsf{d} = -10$.

For these 4×4 systems the operation count for the back substitution was a mere 22 flops each.

Example 6 *Compare FFGJ with FFLU–FFFS–FFBS for inverting a matrix.*

We applied the two approaches to our matrix A using $B = I$ as a right-hand side.

The command `ffgj(A,B)` gives the expected result

$$\mathsf{U} = \begin{bmatrix} -10 & 0 & 0 & 0 & 10 & -10 & 0 & 0 \\ 0 & -10 & 0 & 0 & -10 & 20 & -10 & 0 \\ 0 & 0 & -10 & 0 & -4 & -10 & 20 & -4 \\ 0 & 0 & 0 & -10 & 3 & 0 & -10 & 3 \end{bmatrix}$$

and uses a total of 341 flops.

The output from `LU=fflu(A); R=fffs(LU,B); [S,d]=ffbs(LU,R)` is

$$\mathsf{S} = \begin{bmatrix} 10 & -10 & 0 & 0 \\ -10 & 20 & -10 & 0 \\ -4 & -10 & 20 & -4 \\ 3 & 0 & -10 & 3 \end{bmatrix}, \qquad \mathsf{d} = -10$$

This sequence of commands used a total of 251 flops representing a saving of some 26% relative to the Gauss-Jordan approach.

3 Fraction-Free Algorithms and the Dixon Resultant

In this section we show how to combine the fraction–free algorithms with the Dixon resultant elimination to triangularize a system of polynomial equations. Often the solution set of the resulting "triangular system" *contains* all the solutions of the original system. First recall the basic Dixon method ([5], [7], [9], [8], [3]) and its generalization by Kapur, Saxena, and Yang [9]. For more details see [13].

3.1 The Classical Dixon Resultant

We start with Cayley's formulation [2] of Bezout's method for solving a system of two polynomial equations. Let $f(x)$ and $g(x)$ be polynomials in x, let d=max(deg(f), deg(g)), and let s be an auxiliary variable. The quantity

$$\delta(x,s) = \frac{1}{x-s} \begin{vmatrix} f(x) & g(x) \\ f(s) & g(s) \end{vmatrix}$$

is a symmetric polynomial in x and s of degree $d-1$ which we call the **Dixon polynomial** of f and g. Cayley (and Bezout in different notation) observed that: Every common zero of f and g is a zero of $\delta(x,s)$ for all values of s. Hence, at a common zero, each coefficient of s^i in $\delta(x,s)$ is $\equiv 0$. In matrix notation,

$$M \begin{bmatrix} x^0 \\ \vdots \\ x^{d-1} \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$$

where, the rows of the $d \times d$ matrix M consist of the coefficients of the s^i's. This yields a homogeneous system in new variables $v_1, \ldots, v_d$ corresponding to $x^0, \ldots, x^{d-1}$ and equations corresponding to the coefficients of s^i.

$$M \begin{bmatrix} v_1 \\ \vdots \\ v_d \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$$

The system has non–trivial solutions if and only if its determinant D is zero. D is called the **Dixon resultant** of f and g and M is the **Dixon matrix.** We conclude that: *The vanishing of D is a necessary condition for the existence of a common zero of f and g.*

Dixon generalized Cayley's approach to Bezout's method to systems of three polynomial equations in two unknowns.

$$\begin{aligned} f(x,y) &= 0 \\ g(x,y) &= 0 \\ h(x,y) &= 0 \end{aligned} \tag{2}$$

δ is now defined by

$$\delta(x,y,s,t) = \frac{1}{(x-s)(y-t)} \begin{vmatrix} f(x,y) & g(x,y) & h(x,y) \\ f(s,y) & g(s,y) & h(s,y) \\ f(s,t) & g(s,t) & h(s,t) \end{vmatrix} \quad (3)$$

for auxiliary variables s and t. One gets a homogeneous linear system as before, by setting the coefficients of the power products $s^i t^j$ equal to zero. The corresponding determinant of the coefficient matrix is the Dixon resultant D and is free of the variables x and y.

To illustrate, let us consider the following example discussed in [6].

Example 7 *Eliminate y and z from the system (intersection of two planes and a sphere):*

$$\begin{aligned} x - az + b &= 0 \\ y - cz + d &= 0 \\ x^2 + y^2 + z^2 - R^2 &= 0 \end{aligned}$$

Solution: The Dixon polynomial is given by δ where $(y-s)(z-t)\delta(y,z,s,t) =$

$$\begin{vmatrix} x-az+b & y-cz+d & x^2+y^2+z^2-R^2 \\ x-az+b & s-cz+d & x^2+s^2+z^2-R^2 \\ x-at+b & s-ct+d & x^2+s^2+t^2-R^2 \end{vmatrix}$$

so

$$\begin{aligned} \delta(y,z,s,t) &= xt - azt + bt - ays - ady + cxy - aR^2 \\ &\quad + ax^2 - ads + bcy + cxs + bcs + xz + bz \end{aligned}$$

Therefore, the Dixon matrix is

$$\begin{array}{c} \\ s^0t^0 \\ s^0t^1 \\ s^1t^0 \end{array} \begin{array}{c} \begin{array}{ccc} y^0z^0 & y^0z^1 & y^1z^0 \end{array} \\ \begin{bmatrix} -aR^2 + ax^2 & b + x & cx + bc - ad \\ b + x & -a & 0 \\ cx + bc - ad & 0 & -a \end{bmatrix} \end{array}$$

Its determinant is the Dixon resultant,

$$\begin{aligned} a\big(&-a^2R^2 + a^2x^2 + b^2 + 2bx + x^2 + c^2x^2 + 2bc^2x \\ &- 2acdx + b^2c^2 - 2abcd + a^2d^2\big) \end{aligned}$$

which is free of y and z.

Note In practice we compute the Dixon polynomial δ as the following determinant:

$$\delta(x,y,s,t) = \begin{vmatrix} f(x,y) & g(x,y) & h(x,y) \\ f_1(x,y,s) & g_1(x,y,s) & h_1(x,y,s) \\ f_2(y,s,t) & g_2(y,s,t) & h_2(y,s,t) \end{vmatrix} \quad (4)$$

where,

$$\begin{aligned} f_1(x,y,s) &= \frac{f(s,y) - f(x,y)}{x-s} \\ f_2(y,s,t) &= \frac{f(s,t) - f(s,y)}{y-t} \end{aligned}$$

and g_1, g_2, h_1, h_2 are defined similarly. This works very well when the polynomials are not of high degree and highly sparse. For example, $x^{100} - 1$ is not a good candidate since it introduces an 100 term quotient polynomial when divided by $x - 1$.

Dixon proved that *for three generic 2degree polynomials, the vanishing of the Dixon resultant D is a necessary condition for the existence of a common zero. Furthermore, D is not identically zero.*

Dixon's method and proofs easily generalize to a system of $n+1$ generic $n$degree polynomials in n unknowns. The method applies to polynomials with symbolic coefficients, which allows for simultaneous elimination of a block of unknowns by only one calculation. Often, even if the polynomials are not generic $n$degree the method may yield a necessary condition for the existence of a common zero. These features, along with the relatively small size of the resulting determinants (compared with other resultant methods) makes the method *very* attractive.

One of the problems in using this method with non–generic polynomials is that the Dixon resultant may be identically zero. This is the case, for example, for the polynomial set

$$\begin{aligned} f &= xy + xz + x - z^2 - z + y^2 + y \\ g &= x^2 + xz - x + xy + yz - y \\ h &= x^2 + xy + 2\,x - xz - yz - 2\,z \end{aligned} \quad (5)$$

3.2 The Kapur–Saxena–Yang Approach

Kapur, Saxena and Yang generalized Dixon's theorem for non–generic polynomials as follows: Suppose we have a system of $n+1$ polynomial equations in n variables such that the coefficients of the polynomials are themselves polynomials in a finite set of parameters. Let M be the Dixon matrix obtained as before. Let M' be an echelon form matrix obtained from M by using elementary row operations except for scaling of rows. (Such a reduction is always possible.) Let D be the product of all pivots of M'.

Theorem 1 (Kapur–Saxena–Yang) *If the column that corresponds to the monomial $1 = x_1^0 x_2^0 \ldots$ of the Dixon matrix is not a linear combination of the remaining ones[1] then $D = 0$ is a necessary condition for the existence of common zeros.*

This theorem yields a *simple* algorithm for obtaining the necessary condition $D = 0$. We refer to D in the theorem as the Kapur–Saxena–Yang (KSY) Dixon resultant.

Let us now return to System (5) whose Dixon resultant is identically zero. The KSY Dixon resultant is

[1] In our notation this is the first column of the Dixon matrix.

$$D = 8z(z-1)(2z-1)(2z^2+3z-2) \qquad (6)$$

which is non identically zero. In fact, solving $D = 0$ for z yields all the values of z that lift to the solutions of the original system!

3.3 Triangularization

Let us now see how to combine the fraction–free algorithms with the Dixon resultant elimination to triangularize a system of n polynomial equations in m variables, with $m \geq n$. Such a system we view as one in $n-1$ variables with parametric coefficients. For simplicity, let us consider System (2). Instead of computing the Dixon polynomial δ in one step, we apply a fraction–free elimination to the matrix A with determinant δ from (4).

$$A = \begin{bmatrix} f(x,y) & g(x,y) & h(x,y) \\ f_1(x,y,s) & g_1(x,y,s) & h_1(x,y,s) \\ f_2(y,s,t) & g_2(y,s,t) & h_2(y,s,t) \end{bmatrix}$$

Fraction–free elimination of A yields a matrix of the form

$$E(A) = \begin{bmatrix} f(x,y) & * & * \\ 0 & \det(A_1) & * \\ 0 & 0 & \det(A) \end{bmatrix}$$

where, $\det(A_1)$ is the principal 2×2 minor of A. Explicitly,

$$\det(A_1) = \frac{f(x,y)g(s,y) - g(x,y)f(s,y)}{x-s}$$

Note that if $\det(A_1)$ is considered as a polynomial in its own right its Dixon resultant will be free of x. Also $\det(A)$ has Dixon resultant free of x and y. So, we see that the diagonal elements of $E(A)$ yield Dixon resultants that produce a system of the form f, g^x, h^{xy}, where g^x is free of x and h^{xy} is free of x and y. If the assumption of the theorem holds for each Dixon matrix then any solution of the original system is also a solution of f, g^x, h^{xy}.

3.3.1 Example

To illustrate let us apply Dixon triangularization to System (5). The Dixon polynomials after the fraction–free Gauss elimination are:

$$-y - z - 1$$

$$\begin{aligned} &-ys + s + z^2 - y - st - s^2 + t - ty - y^2 \\ &\quad - tz - z - xt - xs - x - xy \end{aligned}$$

$$\begin{aligned} &-2z + 2z^2 s - ys^2 + 2s^2 z - 3xs^2 + 2yzs^2 \\ &\quad +2yzxs + 2zxs^2 - 2ty - ty^2 + 2tz^2 \\ &\quad +2y^2 z + 2txyz + 2tzxs + 2tyzs + 2y^2 zs \\ &\quad -2yz^2 s + 2z^3 y + 2z^3 x + 2z^2 + 2ty^2 z \\ &\quad +2tzx - txy + 4tyz + 2tsz - tys \\ &\quad -3txs - 2xz^2 s - 3y^2 s + yz^2 + zx \\ &\quad +2xyz + yz + 3zxs - 4ys - 2xs \\ &\quad +5yzs - 3xys + z^2 x - 2tz \end{aligned}$$

and the corresponding Dixon resultants yield the following triangularization:

$$\begin{aligned} &xy + y^2 - z^2 + xz - z + x + y \\ &-\left(y^2 + 2\,y - z - yz - 2\,z^2 + 1\right) z\,(y + z + 1) \\ &8\,(-1 + 2\,z)\,z\,(z-1)\,\left(3\,z - 2 + 2\,z^2\right) \end{aligned}$$

The solutions of the triangular system over $\mathbf{Q}$ are:

$$\begin{aligned} &(-r, r, 0), (r, -3/2, 1/2), (r, -2, 1), (r, -1, 0), \\ &(0, 1, 1), (\frac{1}{2}, 0, \frac{1}{2}), (3, -5, -2) \end{aligned}$$

for any rational r. These certainly include the solutions of the original system:

$$(0,0,0), (1/2, 0, 1/2), (1/2, -3/2, 1/2), (0, -2, 1), (3, -5, -$$

The part of this manuscript that refers to the Dixon triangularization is in its preliminary stage and it needs more theoretical work. However, there seem to be some advantages to our approach. Experiments indicate that this method is fast compared with Gröbner bases solutions. It is also fast compared with other resultant methods. Unlike the one step elimination of other resultant methods we eliminate one variable at a time until triangularization.

There also some disadvantages of the Dixon triangularization. The method may fail due to singularities. In this case reshuffling of polynomials may help. We get extra solutions that are not solutions of the original system. This is due to the extraneous factors of the Dixon resultant. The method is not very efficient for high degree polynomial systems with more than 5 or 6 equations.

4 Future directions/ Further work

There are several avenues which should be explored in further work on this subject. One potentially important

development is that by taking advantage of the full (triangular) array of minors which are produced during the fraction-free triangularization of the δ-matrix it may be possible to remove some of the extra factors. These minors would yield information on whether the corresponding subsystems have solutions. These subsystem solutions would in turn provide conditions on the overall solution which could be used to eliminate the spurious solutions generated by the Dixon resultant of the full system.

A different aspect which should be investigated is the use, and advantages, of parallelism in the computation. Most elimination-based linear algebra routines can be substantially improved by the use of single-instruction, multiple datastream, SIMD, parallel (or other vector-based) processors. Since these operations are at the heart of all the techniques discussed here, it seems natural to ask what benefit such architectures can provide in the context of systems of polynomial equations. Since the dimensions of the matrices produced by resultant techniques can grow rapidly with the number and degree of equations, the potential benefit of parallel symbolic computation could be considerable.

Another aspect which should be undertaken is an extension of these methods to teh actual solution of the polynomial systems and then a comparison between this approach and others such as the recent work of Manocha [12].

5 Acknowledgments

We wish to thank the Naval Academy Research Council and the Office of Naval Research for financial support under grant N0001496WR20018.

References

[1] E.H.Bareiss, *Sylvester's Identity and Multistep Integer-Preserving Gaussian Elimination*, Math Comp. 22 (1968) 565-578

[2] A.Cayley, *On the Theory of Elimination*, Cambridge and Dublin Math J. III (1865) 210-270

[3] E.Chionh, *Base points, resultants, and the implicit representation of rational surfaces*, PhD Thesis, Dept Comp Sci, Univ of Waterloo, 1990

[4] R.M.Corless & D.J.Jeffrey, *The Turing factorization of a rectangular matrix*, this BULLETIN, pp. 20–28.

[5] A.L.Dixon, *The elimination of three quantics in two independent variables*, Proc LMS, 6 (1908) 468-478

[6] R.F.Gleeson & R.M.Williams, *A primer on polynomial resultants*, NADC Tech Rep (1991) NADC-91112-50

[7] D.Kapur & Y.N.Lakshman, *Elimination Methods: an Introduction*, Symbolic and Numerical Computation for Artifical Intelligence (B.Donald et al, Eds.) Academic Press, New York, 1992

[8] D.Kapur & T.Saxena, *Comparison of various multivariate resultant formulations*, preprint

[9] D.Kapur, T.Saxena & L.Yang, *Algebraic and Geometric Reasoning using Dixon Resultants*, Proc ISSAC94, ACM, Oxford, 1994

[10] B.J.Kirsch & P.R.Turner, *Modified Gaussian Elimination for Adaptive Beamforming using Complex RNS Arithmetic*, NAWC-AD Tech Rep (1994) NAWCADWAR 94112-50

[11] B.J.Kirsch & P.R.Turner, *Adaptive Beamforming using RNS Arithmetic*, Proc ARITH 11, IEEE Computer Society, Washington DC, 1993, pp 36-43

[12] D.Manocha, *Solving Systems of Polynomial Equations*, IEEE Comp Graph & Applics, March 1994

[13] G.Nakos & R.M.Williams *A Primer on the Dixon Resultant*, preliminary report

[14] P.R.Turner, *A Simplified Fraction-Free Integer Gauss Elimination Algorithm*, NAWC-AD Tech Rep (1996) NAWCADPAX 96-196-TR

[15] P.R.Turner, *Gauss Elimination: Workhorse of Linear Algebra*, NAWC-AD Tech Rep (1996) NAWCADPAX 96-194-TR

[16] P.R.Turner, *Fraction-Free RNS Algorithms for Solving Linear Systems*, Proc ARITH13, IEEE Computer Society, Washington DC, 1997, pp 218-224