ON PROGRAM EFFICIENCY AND ALGEBRAIC COMPLEXITY

Jan van Leeuwen

RUU-CS-79-1

January 1979

Rijksuniversiteit Utrecht

Vakgroep Informatica

Budapestlaan 6
Postbus 80.012
3508 TA Utrecht
Telefoon 030-531454
The Netherlands

ON PROGRAM EFFICIENCY AND ALGEBRAIC COMPLEXITY

(or: how to compute the Schur transform of a complex polynomial)

Jan van Leeuwen

technical report RUU-CS-79-1

January 1979

Department of Computer Science
University of Utrecht
P.O.Box 80.012
3508 TA Utrecht, the Netherlands

Proposed running head:

PROGRAM EFFICIENCY AND ALGEBRAIC COMPLEXITY

All correspondence to:

Dr. Jan van Leeuwen
Dept. of Computer Science
University of Utrecht
Budapestlaan 6
P.O.Box 80.012
3508 TA Utrecht, the Netherlands

ON PROGRAM EFFICIENCY AND ALGEBRAIC COMPLEXITY

Jan van Leeuwen

Department of Computer Science
University of Utrecht
3508 TA Utrecht, the Netherlands

Abstract. Minimizing the number of arithmetic operations to determine the Schur transform of a complex polynomial provides a concrete example of how the general concern for program efficiency can lead to the kind of questions algebraic complexity theory attempts to answer.

1. Introduction.

Choosing the proper algorithm for a task is an important phase in developing application software. In order to tell one algorithm from another an analist must not only judge which algorithms score highest in terms of clarity, robustness and maintenance, but also which achieve a desirable or perhaps predetermined degree of efficiency. Analysis of algorithms has provided tools to measure and compare algorithms, and through the results of this area we now know for a good number of practical problems why some programs are "better" than others (cf. Aho, Hopcroft and Ullman [1]). Pushing for the most efficient, i.e. "fully optimized", program for a task can be of great commercial value, but there is going to be some question as to how far the "optimization" can be carried through. Within a given frame of mind (and the limitations of your computer's instruction set) there is likely to be some limit to the number of operations that can eventually be saved. Determining lower-bounds on the number of operations all algorithms (of some admissible type) must perform to achieve a given task is the domain of a field normally referred to as "complexity theory", sometimes with an additional adjective indicating what mathematical model for algorithms is adopted for the theory.

Returning to algorithms, it is not likely that the overall design of algorithms is a programmer's responsibility. All he will do is make sure that the steps of an algorithm are properly transcribed into program

statements when the design is handed to him. When the overall design is fixed then, as one may observe, the programmer's concern for program efficiency often leads him to optimize code locally. Where it pays he attempts to write the shortest code for each "step" of the algorithm. And this may well be the commonest way programmers run into complexity theory.

The one topic of evaluating algebraic expressions using shortest straight-line code (and/or fewest registers) leads to a collection of difficult mathematical questions, which "algebraic" complexity theory has been attempting to answer for the past decade or so (see e.g. Borodin and Munro [2]). Looking at expressions as finitely generated elements of some algebra, one should not be surprised to find that the minimum number of ADD, SUB, MULT or DIV instructions to evaluate some polynomial or rational expression has intimate ties to concepts of height, degree and rank in classical mathematics (see Strassen [19, 20, 21] for a more detailed account of this). In its widest sense, algebraic complexity theory is concerned with all applications of algebra to lowerbound problems for algorithms.

We shall attempt to illustrate how a programmer can very easily run into the problems of algebraic complexity by exploring a relatively simple (yet practical!) task, the computation of the Schur transform of a complex polynomial. What the Schur transform is and why you may be interested in it will be explained in section 2. Suffice it for now to say that the Schur transform figures as a frequently called subroutine in some universal polynomial root-finder. It will appear that one can substantially save on the arithmetic in at least one documented program for it. In section 3 we shall explore what algebraic complexity can tell us about the optimality of the resulting program. I hope that the very same exploration will show the programmer that the questions of algebraic complexity (and the meaning of results "with or without preconditioning") are of more than academic interest and that it will inspire others, perhaps, to study the few concrete problems which I did not fully solve. No particular background is needed to read the results below, but in the proofs I shall assume some familiarity with the basic results of algebraic complexity as presented in e.g. Aho, Hopcroft and Ullman [1] (Chapter 12) and Borodin and Munro [2] (Chapters II, III).

2. Software for the Schur transform.

Let $p(z) = a_0+a_1z+\ldots+a_nz^n$ be a (complex) polynomial of degree n and let its reverse p^* be

$$p^*(z) = \bar{a}_n+\bar{a}_{n-1}z+\ldots+\bar{a}_0z^n.$$

Definition. The Schur transform of p is the $(n-1)^{st}$ degree polynomial Tp given by

$$Tp(z) = \bar{a}_0p(z) - a_np^*(z) = \sum_{k=0}^{n-1}(\bar{a}_0a_k - \bar{a}_{n-k}a_n)z^k$$

It is interesting to see how the Schur transform got into being. Ever since the days of Gauss, mathematicians and many others have considered the questions of locating the roots of polynomials in the complex plane. Near the end of the last century, some published the first results aimed at finding effective criteria to actually bound the number of roots within a given region. For example, the "Routh-Hurwitz problem" asked for the most elegant necessary and sufficient conditions that all roots would lie in the left half-plane. In a 1917-1918 paper, Schur [17] apparently was first to run into a criterion for all roots of a polynomial to lie outside the unit circle. (He immediately noted that, at the expense of a simple transformation, it gave a new solution for the Routh-Hurwitz problem at the same time.) Acknowledging Schur for guidance, Cohn [3] subsequently found a method for counting roots by location with respect to the unit circle which directly extended Schur's original criterion. Here one encounters the first occurrence of what became the "Schur transform".

In its simplest form the Schur-Cohn test goes like this. Let T^j denote the j^{th} iterate of transform T and always regard T^jp as an $(n-j)^{th}$ degree polynomial (even when its first coefficient is 0). Let

$$\delta_j = T^jp(0) \qquad (j = 1,\ldots,n)$$

Theorem 2.1. Polynomial p has no roots in (or on) the unit circle if and only if $\delta_j > 0$ for all j.

(We shall give the generalization to a counting formula in a moment.)

One should realize that mathematicians in those days were greatly interested in all sorts of problems concerning the location of roots of functions (and not just of polynomials). See van Vleck [23] for an excellent account of the work in the early 1900's. So the "Schur-Cohn

problem" was not "solved and forgotten", but it stayed quite alive throughout the twenties (see e.g. Herglotz [7]). Fujiwara [5] showed that the solutions to the Routh-Hurwitz and the Schur-Cohn problem could be derived by one uniform method. In the late forties, Marden [12] (see also Marden [13]) captured Cohn's results in an elegant new way. Let

$$\gamma_j = \delta_1 \cdots \delta_j \qquad (\delta\text{'s as above, } j = 1 \ldots n)$$

Theorem 2.2. If k of the γ_j's are negative and the remaining n-k are positive, then p has k roots within, no roots on and n-k roots outside the unit circle.

(There are ways to get around zero γ_j's, but this will not be of interest to us here.)

Ever since electronic computers became widely available for high-speed general purpose computing (in the fifties), the world of algorithms has not been the same. Problem solutions that had never been feasible could now be programmed and run in seconds. As far as polynomial root-finding is concerned, some tried to find new methods specifically designed for execution by computer and so powerful that, when properly programmed, higher degrees of accuracy could be achieved faster than by any previous algorithm. (The earliest attempt to find such a "universal" method probably has been Moore [14].) In 1961 our story continues, when Lehmer [10] showed how the Schur-Cohn test (which he did not mention by that name, by the way) could be used for a practical algorithm to isolate all roots of a polynomial in circles of any radius desired, no matter how small. Lehmer must have been aware of Marden's [13] thorough treatment of the Schur-Cohn results, but he did the necessary mathematics completely over in more modern function-theoretic terms. Given the validity of the Schur-Cohn test, the idea of the algorithm is easy enough to explain. By means of a proper change of coordinates the Schur-Cohn test can be used on any circle in the complex plane. Now begin with a sufficiently large circle of radius R containing all roots, and recursively probe circles which contain at least one root with covering sets of some constant number of smaller circles (of about half the radius). At each stage the regions that contain no roots are discarded. See also Lehmer [11].

The details of Lehmer's algorithm are included in most standard texts on numerical mathematics today. See e.g. Ralston [15] (where it is referred

to as the "Lehmer-Schur method") and Henrici [6]. A precise study of the practical merits of Lehmer's algorithm was undertaken by Stewart [18], who came up with several modifications to alleviate the algorithm's numeric instability. Compared to more classical root-finders, Lehmer's algorithm reportedly is somewhat slow, mainly because of the computational overhead involved in each step to a (full) set of smaller circles. Since the overhead consists of performing the Schur-Cohn test on an entire set of new circles, it is very important that it is programmed to run fast in each individual case. Savings, however small, will pay off in the end because of the large number of times the test is performed. Hence the requirement that the Schur-transform Tp itself be programmed with the greatest care for efficiency.

Rasmussen [16] apparently contains the first explicit program for Lehmer's (unmodified) algorithm documented in the open litterature. We shall favor it, rather than Stewart's later program for a modified Lehmer algorithm (microfiche appendix to [18]), for our discussion below. Sure enough Rasmussen's program contains a subroutine for computing Tp, but it will appear not to compute it in the most efficient way! This is the point where the tools from (algebraic) complexity theory will become important. To explain Rasmussen's routine for Tp we need some additional notation. Recall that we must compute the coefficients of

$$Tp(z) = (\bar{a}_0 a_0 - \bar{a}_n a_n) + \ldots + (\bar{a}_0 a_k - \bar{a}_{n-k} a_n) z^k + \ldots + (\bar{a}_0 a_{n-1} - \bar{a}_1 a_n) z^{n-1}$$

An immediate complication is that we must do complex arithmetic. Let us represent the coefficients a_k as

$$a_k = X[k] + i\,Y[k],$$

with X[k] and Y[k] locations of the arrays X[0:n] and Y[0:n] containing the real and imaginary parts of a_k, respectively. In section 3 we shall just write X_k and Y_k to refer to the constituent parts of the coefficients a_k. The coefficients of Tp will be symbolically referred to as $A_0, \ldots, A_{n-1}$ when needed. With some further renaming of variables, Rasmussen's program segment for Tp goes like this. (We omit scaling of the X and Y arrays and a test to determine whether a complete transform is required or not.)

```
comment  First calculate A0  end of comment
p1 := X[0]; p2 := Y[0]; q1 := X[n]; q2 := Y[n];
X[0] := p1 * p1 + p2 * p2 - q1 * q1 - q2 * q2;
Y[0] := 0;
comment  Note that the values of a0 and an are saved in p1, p2
and q1, q2 respectively, so there is no harm in overwriting their
array representation. Next we compute A1 to An-1 in pairs
Ai, An-i  end of comment
imax := n÷2;
for i := 1 step 1 until imax do
    begin
        r1 := X[i]; r2 := Y[i]; s1 := X[n-i]; s2 := Y[n-i];
        X[i] := p1 * r1 + p2 * r2 - q1 * s1 - q2 * s2;
        Y[i] := p1 * r2 - p2 * r1 + q1 * s2 - q1 * s1;
        if n≠ 2 * i then
        begin
        X[n-i] := p1 * s1 + p2 * s2 - q1 * r1 - q2 * r2;
        Y[n-i] := p1 * s2 - p2 * s1 + q1 * r2 - q2 * r1;
        end
    end;
comment  The coefficients of Tp are now stored in X[0:n-1] and
Y[0:n-1] like they were for p  end of comment
```

The program certainly shows some clever programming. By using p1, p2 etcetera the number of array references needed for the formulae is kept to a minimum. Most interesting of all is the fact that the real and imaginary parts of Tp's coefficients can be stored in the same arrays X and Y immediately after being computed and no extra, auxiliary arrays are needed. Yet the program has some weak points as well. For instance, why would you want to include the test "if n≠ 2 * i..." in the for-loop when you know in advance it's going to yield true for all values of i except, perhaps, for i = imax. A moment's reflection will show that the program remains valid when the test is dropped entirely, the only penalty being that at the end (when n is even) the value of $A_{n/2}$ may get calculated twice. If the effort wasted on this compares favorably to performing n/2 redundant tests, then you might just do it. Otherwise a program like the following will avoid redundant testing.

```
imax := n÷2; loopmax := imax - 1;
for i := 1 to loopmax do
    begin
        compute A_i and A_n-i as before
    end;
r1 := X[imax]; r2 := Y[imax];
if n even then
    begin
        X[imax] := p1 * r1 + p2 * r2 - q1 * r_1 - q2 * r2;
        Y[imax] := p1 * r2 - p2 * r1 + q1 * r2 - q2 * r1
    end
else
    begin
        s1 := X[n-imax]; s2 := Y[n-imax];
        compute A_imax and A_n-imax as before
    end;
```

In either case we note that the calculations within the for-loop dominate the run-time of the code. As it stands there are 16 multiplications and 12 additions/subtractions performed each time 'round the loop (some n/2 times!). Even when n is small there is reason for wanting to do better because, as explained, the routine is used many times in the course of Lehmer's algorithm and savings of any kind will add up in the long run.

3. Optimal computation of the Schur transform.

We shall now gradually move towards the domain of algebraic complexity theory. Let us call $\bar{a}_0 a_k - \bar{a}_{n-k} a_n$ the k^{th} Schur coefficient. The task to determine all Schur coefficients in terms of real parameters amounts to an evaluation of the following formulae in the "smallest number of steps":

$$\begin{array}{l} X_0^2 + Y_0^2 - (X_n^2 + Y_n^2) \\ \vdots \\ X_0 X_k + Y_0 Y_k - X_n X_{n-k} - Y_n Y_{n-k} \\ X_0 Y_k - Y_0 X_k + X_n Y_{n-k} - Y_n X_{n-k} \\ \vdots \end{array} \qquad (3.1)$$

for k from 1 to n-1. (Verify it from Rasmussen's program!) In a way we're asking for the fastest assembly language program for it, a question which can certainly not be answered universally for all machines. The model for programming adopted in algebraic complexity theory makes use of straight-line coding in purely algebraic terms.

Definition. A straight-line program for a set of formulae $F_0, F_1, \ldots$ is a (finite) sequence of steps $\pi_1, \pi_2, \ldots, \pi_m$ such that

(a) each π_i is of the form $\tau_1 \left\{ \begin{matrix} * \\ + \\ - \end{matrix} \right\} \tau_2$ for $\tau_{1,2}$ which either are a scalar, an input-variable or some π_j for $j < i$,

(b) each formula F_i is the computed result of some π_j.

(Compare e.g. Aho, Hopcroft and Ullman [1], sect. 12.2.) It is tacitly understood that the scalars are taken from $\mathcal{R}$ and that we shall not want to use non-scalar divisions for our computational task ahead. (Note that Stewart's program [18] uses both complex arithmetic and non-scalar divisions, by the way.) The effect of choosing different fields of scalars on algebraic complexity has recently been studied by Winograd [25, 26].

With all vagueness of the computational model resolved and guaranteed machine-independence, we can now ask for the shortest straight-line program to compute all Schur-coefficients. Minimizing a total operation count immediately appears to be a hard question. Algebraic complexity theory offers powerful techniques for establishing a smallest number of multiplications or a smallest number of additions/subtractions, but is somewhat at a loss for minimizing the two combined. In a naive computation of the Schur-coefficients multiplications dominate the operation count (it's 16 versus 12, remember) and it may be a good idea to try and save on them rather than on the additions/subtractions. There may be another good reason for doing so, when the same algorithm is to be efficient in multi-precision arithmetic as well. Multiplications tends to become rapidly more expensive than additions and subtractions if higher-order precision is required and it is clear that the algorithms using fewest multiplications will win. In our case the number of additions/subtractions (fortunately) will stay low as well. Thus having defined our aim, there is only one catch to our model that might give us headaches (it won't, but it could in general). If one has a parametrized set of formulae F(n) like we do here, then the "optimal" straight-line code for F(n+1) may very well have no similarity at all to the optimal code for

F(n)! It does not matter if one just needs a program for one (fixed) value of n, but it's a different ball-game if a general routine is desired: there is no guarantee (apparently) that the different straight-line texts can be coded together into one parametrized, finite subroutine in a higher level programming language. Consequently, straight-line complexity may not be quite as realistic a measure for "real program" complexity as we initially thought. Yet straight-line programs will do here to obtain achievable (read: programmable) lower-bounds.

Let us first consider the more special task of computing the Schur transform of a real polynomial (i.e. $Y \equiv 0$). The coefficients (3.1) reduce to a much simpler form

$$\begin{array}{l} X_0^2 - X_n^2 \\ \vdots \\ X_0X_k - X_nX_{n-k} \\ \vdots \end{array} \qquad (3.2)$$

Nevertheless, straightforward calculation of (3.2) would take 2n multiplications and n additions/subtractions. The following scheme shows that we can do with only n multiplications, together with a mere 2n additions/subtractions

$$\begin{array}{l} X_0^2 - X_n^2 = (X_0 - X_n)(X_0 + X_n) \\ \vdots \\ X_0X_k - X_nX_{n-k} = \frac{X_0 - X_n}{2} \cdot (X_k + X_{n-k}) + \frac{X_0 + X_n}{2} \cdot (X_k - X_{n-k}) \\ \vdots \\ X_0X_{n-k} - X_nX_k = \frac{X_0 - X_n}{2} \cdot (X_k + X_{n-k}) - \frac{X_0 + X_n}{2} \cdot (X_k - X_{n-k}) \\ \vdots \end{array} \qquad (3.3)$$

It is hereby understood that for even n the term for $k = \frac{n}{2}$ is just written as

$$X_0X_{\frac{n}{2}} - X_nX_{\frac{n}{2}} = (X_0 - X_n) \cdot X_{\frac{n}{2}}$$

There is nothing profound to this scheme and my guess is that you suspected a simple trick like this right when you saw (3.2). Anyone could see it! So why don't we try to be smarter and save even more multiplications... But can we? Elementary algebraic complexity provides a criterion that shows that we can't. Hence we found a first example of a problem in

formula-coding that needs algebraic complexity to solve it. We give the solution in two parts.

Theorem 3.1. Evaluating $X_0X_k - X_nX_{n-k}$ for k from 1 to n-1 requires n-1 multiplications (hence n-1 multiplications are optimal).

Proof.

We have seen that n-1 multiplications suffice. Reformulate the problem as the task to compute

$$\underbrace{\begin{bmatrix} X_1 & -X_{n-1} \\ X_2 & -X_{n-2} \\ \vdots & \vdots \\ X_{n-1} & -X_1 \end{bmatrix}}_{M} \begin{bmatrix} X_0 \\ X_n \end{bmatrix}$$

, where one should note the separation of variables. In this form the problem precisely fits Winograd's row-rank criterion (Winograd [24], also theorem 12.1 in Aho, Hopcroft and Ullman [1]), which learns that the minimum number of multiplications required is bounded below by the row-rank of M. This rank is seen to be n-1.

■

The same proof breaks down if we want to show that when $X_0^2 - X_n^2$ is to be computed also at least n multiplications are required. The separation of variables in the matrix-times-vector formulation, essential for Winograd's criterion, no longer holds (viz. X_0 and X_n occur in both the matrix ánd the vector). We shall have to dig a bit deeper.

Theorem 3.2. Evaluating $X_0^2 - X_n^2$ and $X_0X_k - X_nX_{n-k}$ for k from 1 to n-1 requires n multiplications (hence n multiplications are optimal).

Proof.

Suppose that one could evaluate the formulae using less than n multiplications. By 3.1 at least n-1 are required. Hence there must be a straight-line π and n-1 products $p_1,\ldots,p_{n-1}$ evaluated during the course of π, such that the n formulae desired can be synthesized from p_1 to p_{n-1} using no further products (hence using only multiplication by scalars and additions and subtractions). It follows that there must be an n-by-(n-1) matrix N of scalars such that

$$\begin{bmatrix} X_0^2 - X_n^2 \\ X_0X_1 - X_nX_{n-1} \\ \vdots \\ X_0X_{n-1} - X_nX_1 \end{bmatrix} = \begin{bmatrix} & & \\ & N & \\ & & \end{bmatrix} \begin{bmatrix} p_1 \\ \vdots \\ p_{n-1} \end{bmatrix} + \begin{bmatrix} \\ \mathcal{L} \\ \\ \end{bmatrix}$$

where $\mathcal{L}$ is vector with coordinates purely linear in X_0 to X_n. The last n-1 rows of N must be linearly independent. If they weren't, there would be scalars α_1 to α_{n-1}, not all zero, such that (look at what it means on the left-hand side when you linearly combine rows on the right)

$$\alpha_1(X_0X_1 - X_nX_{n-1}) + \ldots + \alpha_{n-1}(X_0X_{n-1} - X_nX_{n-1}) =$$
$$= X_0(\alpha_1X_1 + \ldots + \alpha_{n-1}X_{n-1}) - X_n(\alpha_{n-1}X_1 + \ldots + \alpha_1X_{n-1})$$

would be linear in all X-variables, an impossibility. On the other hand, the row-rank of no scalar matrix with n-1 columns can be larger than n-1. It follows that the first row of N must be linearly dependent of the remaining rows. This means that coefficients β_1 to β_{n-1} exist such that (again after translating row operations)

$$X_0^2 - X_n^2 = \beta_1(X_0X_1 - X_nX_{n-1}) + \ldots + \beta_{n-1}(X_0X_{n-1} - X_nX_1) + \mathcal{L}' =$$
$$= X_0(\beta_1X_1 + \ldots + \beta_{n-1}X_{n-1}) + X_n(\beta_{n-1}X_1 + \ldots + \beta_1X_{n-1}) + \mathcal{L}'$$

with some purely linear form $\mathcal{L}'$. This is impossible.

■

We now have proof that the simple trick in (3.3) cannot be outsmarted, in the sense that there is no way more multiplications can be saved. It is noted that the line of argument in theorem 3.2 can be used to show that any set of n forms which are linearly independent modulo linear expressions require n non-scalar multiplications, a useful result observed already in 1971 by Fiduccia [4] (see also Borodin & Munro [2], lemma 2.4.1). It is easy but instructive to implement the scheme of (3.3), which one can do again without using auxiliary array space.

```
p := X[0] - X[n]; q := X[0] + X[n];
X[0] := p * q;
p1 := p/2; q := q/2; imax := n ÷ 2;
for i from 1 to imax - 1 do
   begin
        r := X[i] + X[n-i]; s := X[i] - X[n-i];
        one := p1 * r; two := q * s;
        X[i] := one + two;
        X[n-i] := one - two
   end;
if n even then X imax  := p * X[imax]
          else
              begin
                   compute X[imax] and X[n-imax] as above
              end
```

The routine overwrites the coefficients of a polynomial as given in X with the values of the Schur coefficients like in Rasmussen's algorithm.

There is a new twist to this otherwise straightforward program: the subexpressions X[0] - X[n] and X[0] + X[n], needed in each Schur-coefficient, have been moved out of the loop and are "precomputed" instead. This is reasonable, because there is obviously no sense in doing the subtraction and addition time and again around the loop. It immediately leads to the idea that one could perhaps precompute some other (polynomial) expressions in X_0 and X_n to lighten the computational burden in the loop-body. Any programmer knows one should move invariant subexpressions out from a loop! What does it mean for our computational model. In algebraic complexity theory it has long been noted that there is a close tie with computing modulo a subdomain of some algebra, in our case modulo the ideal I generated by the second order terms in X_0 and X_n within the (commutative) polynomial ring $\mathfrak{R}[X_0,\ldots,X_i,\ldots,X_n]$. Thus the theory gets ramified depending on whether "preconditioning" is taken into account or not (cf. Borodin & Munro [2]). Could precomputing polynomial expressions in X_0 and X_n help to reduce the number of multiplications within the loop once more? There is a simple proof that it can't. The idea is to embed straight-line algorithms into $\mathfrak{R}[X_0,\ldots,X_i,\ldots,X_n]$ modulo I and to argue that Winograd's row-rank criterion still holds in the factor ring. (It does, as follows from

the argument in van Leeuwen & van Emde Boas [22]). Hence we can rest assured that the number of multiplications used is fully optimal. The issue of preconditioning will have a major bearing on the general problem.

Computing the Schur-transform of an arbitrary (complex) polynomial p is a decidedly more intricate affair, unless one takes complex arithmetic for granted. We won't, but let's briefly examine what would happen if we did. If we write

$$\begin{aligned}\bar{a}_0 a_k - \bar{a}_{n-k} a_n &= (\bar{a}_0 - \bar{a}_{n-k})(a_k + a_n) + a_k \bar{a}_{n-k} - \bar{a}_0 a_n \\ \bar{a}_0 a_{n-k} - \bar{a}_k a_n &= (\bar{a}_0 - \bar{a}_k)(a_{n-k} + a_n) + \bar{a}_k a_{n-k} - \bar{a}_0 a_n\end{aligned}$$

then it follows that we need to perform only about $\frac{3}{2}n$, rather than the naive 2n, complex multiplications (assuming that once you have a value, you can get its complex conjugate free of charge!). This can be shown to be optimal in order of magnitude, yet writing each complex multiplication in terms of its real parameters is nót optimal in real arithmetic. Consider the horrendous formulae (3.2) again, in particular consider the pairs for k and n-k

$$\begin{aligned}&X_0X_k + Y_0Y_k - X_nX_{n-k} - Y_nY_{n-k} \\ &X_0Y_k - Y_0X_k + X_nY_{n-k} - Y_nY_{n-k} \\ &X_0X_{n-k} + Y_0Y_{n-k} - X_nX_k - Y_nY_k \\ &X_0Y_{n-k} - Y_0X_{n-k} - X_nY_k - Y_nX_k\end{aligned} \qquad (3.4)$$

and remember that we are interested in computing these forms modulo preconditioning in X_0, Y_0, X_n and Y_n. Surely we can beat the 16 multiplications of the naive algorithm, but what should we strive for in lowering this number. One idea is to see first what lowerbound can be derived by means of an available criterion from algebraic complexity theory. Write (3.4) as

$$\underbrace{\begin{bmatrix} X_k & Y_k & -X_{n-k} & -Y_{n-k} \\ Y_k & -X_k & Y_{n-k} & -X_{n-k} \\ X_{n-k} & Y_{n-k} & -X_k & -Y_k \\ Y_{n-k} & -X_{n-k} & Y_k & -X_k \end{bmatrix}}_{M} \begin{bmatrix} X_0 \\ Y_0 \\ X_n \\ Y_n \end{bmatrix}$$

A criterion of van Leeuwen & van Emde Boas [22] (see also Kruskal [9] or, more recently, Ja'Ja' [8]), slightly modified to work over the presently

relevant ideal here, shows that r + d multiplications are required even with polynomial preconditioning in X_0, Y_0, X_n and Y_n, where r and d are numbers such that the row-rank of M remains $\geq r$ no matter hów the members of some fixed d-element subset of the variables in M are replaced by a linear combination of the remaining variables. Inspection shows that we can take r = 4 and d = 2 (e.g. take X_{n-k}, Y_{n-k} for rank-preserving elimination), hence a lowerbound of 6 follows. It is not clear that this has any meaning, since general lowerbound criteria like the one we used give no assurance at áll that the lowerbound they produce can be achieved by any straight-line program. By experience the general criteria in algebraic complexity theory are nearly always off by a few multiplications. Thus we can only credit good fortune in the present case that there is a way to compute (3.4) in just 6 multiplications (ánd preconditioning, of course). Here it is. Define the following parameters

$$p = (X[0] - X[n])/2$$
$$q = (X[0] + X[n])/2$$
$$r = (Y[0] - Y[n])/2$$
$$s = (Y[0] + Y[n])/2$$
$$u1 = X[k] + X[n-k]$$
$$u2 = X[k] - X[n-k]$$
$$v1 = Y[k] + Y[n-k]$$
$$v2 = Y[k] - Y[n-k]$$

and compute the following expressions, in the way indicated using just exactly 6 counted multiplications,

$$I = p.u1 + r.v1 = (p + v1)(r + u1) - u1.v1 - p.r$$
$$II = q.u2 + s.v2 = (q + v2)(s + u2) - u2.v2 - q.s$$
$$III = q.v1 - s.u1 = (q - u1)(s + v1) + u1.v1 - q.s$$
$$IV = p.v2 - r.u2 = (p - u2)(r + v2) + u2.v2 - p.r$$

The Schur-coefficients from (3.4) can indeed be synthesized from these expressions without much further ado. One easily verifies that

$$X_0X_k + Y_0Y_k - X_nX_{n-k} - Y_nY_{n-k} = I + II$$
$$X_0Y_k - Y_0X_k + X_nY_{n-k} - Y_nX_{n-k} = III + IV$$
$$X_0X_{n-k} + Y_0Y_{n-k} - X_nX_k - Y_nY_k = I - II$$
$$X_0Y_{n-k} - Y_0X_{n-k} + X_nY_k - X_nY_k = III - IV$$

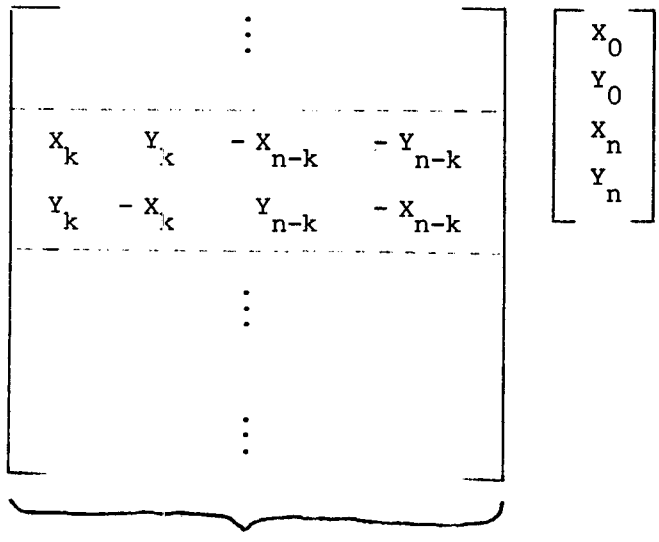
We leave it for the reader to verify that for n even the "lone-standing" Schur-coefficient with $k = \frac{n}{2}$ can be computed in just 3 counted multiplications.

Putting it all together we get a routine for computing the Schur-coefficients for k from 1 to n-1 around the loop using a total of only 3n-3 real multiplications. We have yet to demonstrate that this bound for the entire set cannot be improved further.

Theorem 3.3. Evaluating $X_0X_k + Y_0Y_k - X_nX_{n-k} - X_nY_{n-k}$ and $X_0Y_k - Y_0X_k + X_nY_{n-k} - Y_nX_{n-k}$ for k from 1 to n-1 requires 3n-3 multiplications, even when we do not charge for the computation of any polynomial expression in X_0, Y_0, X_n and Y_n (hence 3n-3 multiplications are optimal).

Proof.

Write the task in matrix-times-vector form again as



We shall apply the "r + d criterion" from van Leeuwen & van Emde Boas [22] again. Let n be odd. It is seen that M keeps row rank (n - 1) * 2 no matter how we replace the n - 1 members of $\{X_{\lceil \frac{n}{2} \rceil}, Y_{\lceil \frac{n}{2} \rceil}, \ldots, X_{n-1}, Y_{n-1}\}$ by linear combinations in the other variables. For n even the same applies with the set $\{Y_{\frac{n}{2}}, X_{\frac{n}{2}+1}, \ldots, X_{n-1}, Y_{n-1}\}$. The r + d criterion learns that the number of multiplications required to compute the set of formulae, even with preconditioning as said allowed, is bounded below by the sum of rank-bound and set-size, i.e. by (n - 1) * 2 + (n - 1) = 3n - 3.

■

Thus we know that in the following program implementing the tricky scheme above, there is no way we can do with fewer multiplications in the loop

```
p1 := X[0]-X[n]; q1 := X[0]+X[n]; r1 := Y[0]-Y[n]; s1 :=  Y[0]+Y[n];
X[0] :=  p1 * q1 + r1 * s1;
Y[0] := 0
p := p1/2; q := q1/2; r := r1/2; s := s1/2; imax := n÷2;
pre1 := p * r; pre2 := q * s;
for i from 1 to imax - 1 do
   begin
        u1 := X[k]+X[n-k]; u2 := X[k]-X[n-k];
        v1 := Y[k]+Y[n-k]; v2 := Y[k]-Y[n-k];
        one := u1 * v1; two := u2 * v2;
        terma := (p+v1) * (r+u1) - one - pre1;
        termb := (q+v2) * (s+u2) - two - pre2;
        termc := (q-u1) * (s+v1) + one - pre2;
        termd := (p-u2) * (r+v2) + two - pre1;
        X[i] := terma + termc;
        Y[i] := termb + termd;
        X[n-i] := terma - termc;
        Y[n-i] := termb - termd
   end;
if n even then
   begin
        one := X[n/2] * Y[n/2];
        X[imax] := (p1 + Y[n/2]) * (r1 + X[n/2]) - one - 4 * pre1;
        Y[imax] := (q1 - X[n/2]) * (s1 + Y[n/2]) + one - 4 * pre2
   end
else
   begin
        compute A(imax-1)  and A(imax+1)  as in main loop
   end;
```

We've come a long way from Rasmussen's original routine to a program that fully optimizes the number of multiplications to a minimum. One more observation is of interest for the final program. It so happens that when it is applied to a real polynomial, the program reduces to the form which is optimal in number of multiplications also: just see what is left after eliminating the predictably zero terms!

4. Summary

We have argued that in writing commendable software a programmer stands a good chance of running into questions, which can only be answered properly by means of algebraic complexity theory. The questions invariably relate to evaluating sets of algebraic expressions using a fewest number of machine operations of some sort. We have demonstrated some of the typical problems that arise in attempting to optimize a subroutine of the Lehmer-Schur algorithm for polynomial root-finding in the complex plane. The paper is not a survey of algebraic complexity theory, but rather an appraisal of the practical side of the area for the non-expert.

5. References

[1] Aho, A.V., J.E. Hopcroft and J.D. Ullman, The design and analysis of computer algorithms, Addison-Wesley Publ. Compl, Reading, Mass., 1974.

[2] Borodin, A. and I. Munro, The computational complexity of algebraic and numeric problems, Elsevier Comp. Sci. Library, Theory of Computation Series, vol. 1, American Elsevier Publ. Comp., New York, NY, 1975.

[3] Cohn, A., Ueber die Anzahl der Wurzeln einer algebraischen Gleichung in einem Kreise, Math. Zeitschrift 14 (1922) 110-148.

[4] Fiduccia, C.M., Fast matrix multiplication, Proc. 3rd Annual ACM Symp. on Theory of Computing, Shaker Heights, 1971, pp. 45-49.

[5] Fujiwara, M., Ueber die algebraische Gleichungen deren Wurzeln in einem Kreise oder in einer Halbebene liegen, Math. Zeitschrift 24 (1926) 160-169.

[6] Henrici, P., Applied computational complex analysis, vol. 1: power series etc., Wiley Interscience, Wiley, New York, NY, 1974.

[7] Herglotz, G., Ueber die Wurzelanzahl algebraischer Gleichungen innerhalb und auf dem Einheitskreis, Math. Zeitschrift 19 (1924) 26-34.

[8] Ja'Ja', J., On the complexity of bilinear forms with commutativity, Techn. Rep. 78-17, Dept. of Computer Science, the Pennsylvania State University, University Park, Pennsylvania, 1978.

[9] Kruskal, J.B., Three-way arrays: rank and uniqueness of trilinear decompositions with applications to arithmetic complexity and statistics, Lin. Algebra and its Applic. 18 (1977) 95-138.

[10] Lehmer, D.H., A machine method for solving polynomial equations, J.ACM 8 (1961) 151-162.

[11] Lehmer, D.H., Search procedures for polynomial equation solving, in: B. Dejon and P. Henrici (eds.), Constructive aspects of the fundamental theorem of algebra, Wiley Interscience, Wiley, London, 1969, pp. 193-208.

[12] Marden, M., The number of zeros of a polynomial in a circle, Proc. National Acad. of Sciences of USA, vol. 34, 1948, pp. 15-17.

[13] Marden, M., The geometry of the zeros of a polynomial in a complex variable, Math. Surveys III, Amer. Math. Society, New York, NY, 1949.

[14] Moore, E.F., A new general method for finding roots of polynomial equations, Math. Tables and other Aids to Computation 3 (1949) 486-488.

[15] Ralston, A., A first course in numerical analysis, McGraw-Hill, New York, NY, 1965.

[16] Rasmussen, O.R., Solution of polynomial equations by the method of D.H. Lehmer, BIT 4 (1964) 250-260.

[17] Schur, I., Ueber Potenzreihen die im Innern des Einheitskreises beschränkt sind (Fortsetzung), Crelle's J. f. d. reine u. angew. Mathematik 148 (1918) 122-145.

[18] Stewart III, G.W., On Lehmer's method for finding the zero of a polynomial, Math. of Computation 23 (1969) 829-235.

[19] Strassen, V., Berechnung und Programm (I), Acta Informatica 1 (1972) 320-335, (II) Acta Informatica 2 (1973) 64-79.

[20] Strassen, V., Berechnungen in partiellen Algebren endlichen Typs, Computing 11 (1973) 181-196.

[21] Strassen, V., Vermeidung von Divisionen, J. f. d. reine u. angew. Mathematik 264 (1973) 184-202.

[22] van Leeuwen, J. and P. van Emde Boas, Some elementary proofs of lowerbounds in complexity theory, Lin. Algebra and its applic. 19 (1978) 63-80.

[23] Van Vleck, E.B., On the location of the roots of polynomials and entire functions, Bull. AMS 35 (1929) 643-683.

[24] Winograd, S., On the number of multiplications necessary to compute certain functions, Comm. Pure and Applied Math. 23 (1970) 165-179.

[25] Winograd, S., The effect of the field of constants on the number of multiplications, Conf. record 16th Annual IEEE Symp. on Foundations of Computer Science, Berkeley, 1975, pp. 1-2.

[26] Winograd, S., Some trilinear forms whose multiplicative complexity depends on the field of constants, Math. Syst. Th. 10 (1977) 169-180.