

Rational Function Computing with Poles and Residues

Richard J. Fateman
Computer Science Division, EECS
University of California, Berkeley

March 21, 2006

Abstract

Computer algebra systems (CAS) usually support computation with exact or approximate rational functions stored as ratios of polynomials in “expanded form” with explicit coefficients. We examine the consequences of using a partial-fraction type of form in which all rational functions are expressed as a polynomial plus a sum of terms each of which has a denominator consisting of a monic univariate linear polynomial perhaps to an integer power. We show that some common operations including rational function addition, multiplication, and matrix determinant calculation can be performed many times faster. Polynomial GCD operations, the costliest part of rational additions, are entirely eliminated. The cost of the common case of multiplication is also reduced.

1 Introduction

Computer algebra systems (CAS) usually support computation with exact rational functions stored as ratios of polynomials in expanded form with explicit coefficients. For example,

$$\frac{7x^3 - 70x^2 + 231x - 252}{x^2 - 11x + 30}. \quad (1)$$

Alternative forms are sometimes used for particular algorithms or for attaining special efficiencies. These include factored (or partly factored) form:

$$\frac{7(x-4)(x-3)^2}{(x-6)(x-5)}. \quad (2)$$

Another variation is to rearrange using Horner’s rule:

$$\frac{x(x(7x-70)+231)-252}{(x-11)x+30} \quad (3)$$

or approximations such as this Taylor series expansion at $x = 0$:

$$-\frac{42}{5} + \frac{231x}{50} - \frac{539x^2}{1500} - \frac{2359x^3}{45000} + \dots \quad (4)$$

A less familiar variation of representation is the use of “straight-line programs” (SLP). One possible SLP version might be:

```
f(x) :=
{t1=x-3; t2=t1*t1; t3=x-4; t4=t2*t3; t5=7*t5;
 t6=x-5; t7=x-6; t8=t7*t6; return(t5/t8)}
```

A “black-box” version might be just like this SLP or some equivalent algorithm, but with the detailed computation hidden. The only information available from a black-box version is a response, for each input value, of a corresponding output value. It is surprising how many computations can be performed (fast) on this apparently impoverished representation [5].

The form that is provided by most CAS that is closest to the topic of this paper is the partial-fraction expansion form:

$$-\frac{28}{x-5} + \frac{126}{x-6} + 7x + 7. \quad (5)$$

While CAS can convert to this form, they usually do not support efficient further computation maintaining this form, especially through multiplication operations.

Versions of this last representation, giving explicit poles and residues are frequently encountered in digital filter design. We have yet to see a situation justifying large-scale direct applications here for semi-symbolic methods, but this may be possible. In fact it seems rather implausible that the definition of a curve provided from measurements of some signal would map naturally into a ratio of expanded polynomials; locations of poles may be more directly available. The use of this format for tools of integration and Laplace transforms is straightforward. It can be helpful in elucidating the potential for numerical overflow when evaluating at a point near the zeros of a denominator. It can also be useful for computations consisting of sequences of multiplies and adds; for example a matrix determinant. (Note that there are popular methods for computing determinants that require division. Such methods are a bad idea in this representation. Fortunately some of the best *symbolic* methods do not use division [11].)

Barton Willis, independent of this work, requested assistance¹ in algorithms for fast manipulation of partial-fraction forms—namely preserving the partial fractions, and therefore *not* dependent on polynomials zero-finding. His application (unpublished) involves the solution of differential equations in terms of ${}_2F_1$ hypergeometric functions, generalizing work by Bronstein and Lafaille [3].

¹Maxima mailing list, Jan. 19, 2005

2 Poles, Residues, Zeros

For the rest of this paper we discuss adopting a version of partial fractions, but with the additional requirement that all factors of the denominator be monic linear factors (again, as in the example). The usual partial-fraction expansion program in a computer algebra system may compute “over the rationals” and thus may not allow a complete decomposition of terms with irreducible (over the rationals) denominators of $1/(x^2 + 1)$. Our requirement of linear factors means that we need a partial-fraction expansion over algebraic numbers, or more likely, approximations of algebraic numbers. Thus we would express $1/(x^2 + 1)$ as $0.5*i/(x + 1.0*i) - 0.5*i/(x - 1.0*i)$, or in reality as a compact data structure with 4 complex floating-point numbers, $(0.5i, 1.0i)$, $(-0.5, -1.0i)$: the two roots and the two residues.

In general producing an expansion requires finding approximations to roots of polynomials in the complex plane, an operation available in most CAS. Maple in fact provides conversion of an expression E to such a form in *one* built-in command, `convert(E,parfrac,x,complex)`.

Before proceeding further, we wish to anticipate some objections:

- The representation we propose does not generalize to more than one variable. Except in special cases the presence of multiple variables leads to computational difficulty since (for example) we would need a good way of representing the “pole” in $f(x, y) := 1/(x - y)$. Considering how many papers are written on algorithms that are only good for *polynomials* in one variable, a much more restricted domain, perhaps we should not worry about this!
- There are possible problems with numerical stability: the traditional objective in computer algebra systems (CAS) is to get exact answers when possible. Here we necessarily commit an approximation of algebraic numbers by (in general) complex floating-point numbers in order to represent polynomial roots. Of course the initial statement of the problem may be approximate already, and we may be able to justify continuation of this representation in further computation. Or we may be able to avoid numerical programs (as in some of our later examples) when the initial input is presented in (exact) partial fraction form. If we can continue to use exact rational arithmetic we maintain the (exact) representation of the answer. What is interesting though, is that the poles are present in the input to the addition, multiplication, and determinant programs, and they determine exactly the poles that may be present in the answer. That is, no poles are created, nor are additional errors introduced by roundoff. It is possible that the order of the poles will be increased or decreased, perhaps to order zero. If the arithmetic is done using floating-point, there is a possible accumulation of errors in the numerators. While these are potential problems, there also is a positive side: In some examples it is possible to use the approximate answers to derive exact results: by bracketing known integer-valued problems so that only one integer is in an approximating

interval. The approximate calculates necessary to do this can be far faster than attempting the exact calculation directly.

As an example of numerical difficulties that might occur during the translation of a conventional form into a residue/pole form, consider the simplest case of a double pole at zero in $f(x) = 1/x^2$. Programs will likely find these roots exactly but let us play devil’s advocate and (mis)represent the roots instead by a pair of equally spaced poles $\pm\epsilon$ away from zero. Then we have a function

$$g(x) := \frac{1}{x^2 - \epsilon^2} = \frac{1/(2\epsilon)}{x - \epsilon} - \frac{1/(2\epsilon)}{x + \epsilon}.$$

While the two forms are mathematically identical, the second (partial-fraction) version of $g(x)$ is computationally less accurate where $|x| \gg 0$ since (as written) it would be computed as the difference of two nearly-equal values.

Mathematically, $g(x)$ is very close to $f(x)$ except near $x = 0$; in that case we can compare the definition of f to the Taylor series for g :

$$f(x) := 1/x^2$$

$$g(x) := -\frac{1}{\epsilon^2} - \frac{x^2}{\epsilon^4} - \frac{x^4}{\epsilon^8} + \dots$$

How close are these? The worst relative error is when $x = 0$ when the approximation gives $1/\epsilon^2$ instead of ∞ . Nearby, say with $\epsilon = 10^{-8}$, the result, $g(10^{-10}) = 1.0001 \times 10^{20}$ can be compared to the exact answer, $f(10^{-10}) = 1.0000 \times 10^{20}$.

We have not attempted to completely characterize the efficiency or the nature of the errors produced by polynomial root-finding programs [6, 12], or the errors committed if we *force* clusters of zeros into multiple zeros. These errors are, for the most part, committed *prior* to using our programs during the conversion process; they are propagated in changes in the values of residues during multiplication, but the poles remain in the same locations. Nevertheless, we note that the cost of such conversion computations as well as the nature of errors in a modern CAS setting may be highly variable: it may be plausible to base an algorithm on finding approximations to extremely high accuracy by means of multiple-precision (software) floating-point arithmetic. If the original coefficients in the problem have some possible error, then these input errors can be made to dominate the calculation²

Knowing all of a function’s poles and zeros along with their multiplicities, and at least one residue, fully determines a rational function. Alternatively, all poles, (with multiplicities), and their residues plus a “direct term” (polynomial part) will determine a rational function. For example, the Matlab system [7] provides a program for changing between two rational function representations. Oddly enough, its syntax uses the same name for a function and its inverse:

²Some of the historical concern for avoiding approximations in CAS seems to have faded, judging by the number of recent publications on approximate polynomial GCD, including some more “applied” researchers (consider [13, 9, 8]). In reality we have not seen any version of an approximate polynomial GCD algorithm become a central part of a computer algebra system, although Maple now has a SNAP [15] library with some approximate GCD algorithms.

```
[r,p,k] = residue(b,a)
[b,a] = residue(r,p,k)
```

The operation `residue(b,a)` takes a pair of polynomials $\{\mathbf{b}, \mathbf{a}\}$ and returns a triple, say $[\mathbf{r}, \mathbf{p}, \mathbf{k}]$, which is equivalent to b/a in “pole-residue” representation. This triple is a list of residues, poles, and a (polynomial) direct term, computed as part of a partial-fraction expansion for b/a . The polynomials b , a and k (in some unstated parameter, say s) are represented as lists. For example, $[3,4,5]$ is $3s^2 + 4s + 5$. These Matlab conversions are done with complex floating-point numbers and are naturally subject to numerical error if the polynomial roots are not representable exactly, or if errors are committed in the solving process.

3 Arithmetic with poles and residues

3.1 Addition

Adding two rational functions in residue form is easily done in linear time: $A = r_1/(s - p_1) + r_2/(s - p_2)$ can be added to $B = r_3/(s - p_3) + r_4/(s - p_4)$ if the poles $\{p_i\}$ are distinct, just by merging lists of poles and residues. The “definite” parts are added as polynomials. The data structure holding the poles and the collection of residues at that position must allow for rapid search and insertion. During the merge, if two terms have the same pole, the residues must be added, and if their sum is zero, the term must be removed. Our programs actually use hash tables.

To have any hope of a canonical external appearance, the lists of poles should be uniquely ordered. The poles may be sorted by a lexical ordering, perhaps real then imaginary parts. The sorting need not be done except on output, and so this is not actually part of the time for addition.

The addition operation can be performed in parallel in constant time if you have enough processors.

3.2 Evaluation

We deal with evaluation first, because this operation is subsumed in the multiplication of expressions, at least as we propose in the next section, where multiplication at a pole point p requires the evaluation of the (other) expression at that point p . There are a variety of “fast” evaluation methods which could be attempted, but they usually are advantageously only for large expressions. Consider the cost then of an evaluation of a pole/residue expression at a generic (i.e. non-pole) point. The polynomial part, say of degree d can be done by Horner’s rule, in $O(d)$ operations, assuming (falsely in general) that coefficient arithmetic is constant cost. For each of q isolated (non-multiple) poles, the cost is one subtraction to compute the denominator, and one division, for cost $O(q)$. If there is a multiple pole of order m then the cost is $O(m)$ if we used Horner’s rule in the obvious way, computing a polynomial at a numerical point $1/(x - p)$.

Thus the evaluation cost is essentially linear in the number of terms, either polynomial or residue/pole, given a plausible dense case expression. Another issue is numerical accuracy. If the point location is very nearly at a root, the denominator of that term will be nearly zero and so any non-zero numerator value will be magnified. If the root is inaccurately computed, this magnification will happen at another location (perhaps close to the root). This does not seem inherently more hazardous than evaluating a high-degree polynomial divisor more-or-less accurately by floating-point methods; it is more hazardous if the denominator can be evaluated exactly. That possibility exists if arbitrary-precision arithmetic is used, but represents a substantially more costly operation.

3.3 Multiplication

Multiplying two $[r, p, k]$ forms is somewhat messy. We are unaware of any prior explicit description of these algorithms in the literature:

Each of the inputs R , S has pole-terms and a polynomial (or “definite” term). By pole-term at p we mean the sum of all the residues of different order, $\sum a_i/(x-p)^i$.

If we use a classical approach to multiplications, we form the cross product of each term in R with each term in S , and add them all together.

Multiplication of the purely polynomial terms can be done by any standard algorithm, with corresponding complexity. The asymptotically fastest known methods are based on FFT techniques; in reality most data is unlikely to be so large and dense with small coefficients, and so FFT techniques are probably not going to be well-suited to typical polynomial problems. A reasonable growth estimate of the cost for multiplying n terms by $m > n$ terms using classical methods is $O(d^2 nm \log n)$ where d is related to a bound on the size of the coefficients. Multiplication of *matching* poles in R and in S is done by matching them by order and multiplying the residues. This is mapped fairly directly into polynomial multiplication (essentially of Laurent series), with corresponding complexity.

Two other cases remain: Multiplication of a polynomial by a pole-term, and multiplication of pole-terms at different poles.

Consider the first of these, pole-term times polynomial.

$$\sum_{i=1}^m \frac{a_i}{(x-p)^i} \times \sum_{i=0}^n b_i x^i$$

Expand the second term (the polynomial) in a power series in $(x-p)$. The coefficients $\{c_i\}$ can be computed by “synthetic division”. Also, let us re-index the $\{a_i\}$ to correspond to negative numbers:

$$\sum_{i=-m}^1 a_i (x-p)^i \times \sum_{i=0}^n c_i (x-p)^i.$$

If we “virtually” multiply by $(x-p)^m$ then the computation above is equivalent to a polynomial multiplication. The m lower order terms become the

pole result. Thus the complexity is still dominated by the cost of a related polynomial multiplication.

The second case is multiplication of pole-terms that are not at the same pole. Our programs assume that any two poles with different floating-point representations are distinct. Any interest one might have in claiming that two (or more) poles that are “very close” should be perturbed to be at the same location are outside the scope of our arithmetic operations.

Some algebra (assisted by a CAS) led us to the following (so far as we know, new) formulas for generating products. The first formula computes an expansion for $w_n := 1/((x - q)^n(x - p))$:

$$w_n := \frac{1}{(p - q)^n(x - p)} - \sum_{i=1}^n \frac{1}{(p - q)^i(x - q)^{n+1-i}}$$

Note that the right-hand side explicitly displays the denominators of powers of $(x - p)$ and $(x - q)$.

We can incorporate the formula into a more general result, for an expansion for $k_{m,n} := 1/((x - q)^n(x - p)^m)$:

$k_{m,n} :=$ **if** $m = 1$ **then** w_n **else**

$$\frac{1}{(p - q)^n(x - p)^m} - \sum_{i=1}^n \frac{k_{m-1,n-i+1}}{(p - q)^i}.$$

Note again that the right-hand side explicitly displays the denominators of powers of $(x - p)$ and $(x - q)$.

In practice, a computation would consist of arranging (negative) powers of $(p - q)$ and adding these residues into the correct slots indexed by p , q , n , m . Powers of $(x - p)$ or $(x - q)$ are never actually “computed” but merely indexed into slots. The summations in the formulas are not actually adding anything, but again denoting an indexed vector of the different pole-term components.

These formulas translate directly into a program for manipulating the data structures. There are $O(mn)$ coefficient multiplies to produce $n + m$ terms.

After programming the above formulas we found their performance to be disappointing, and so we re-thought the operation and programmed a much faster (perhaps new) version of multiplying pole-terms by pole-terms by essentially computing a truncated Laurent expansion (up to power -1) of such product around a , that is, the coefficients of $1/(x - a)^n$ by evaluating the other factor, as well as sufficiently many of its derivatives, at the constant a . The expansion has to be computed about each of the two pole locations in the product. In particular, the product

$$f(x) \times g(x) = \left(\sum_{i=1}^{k_1} \frac{r_i}{(x - a)^i} \right) \times \left(\sum_{i=1}^{k_2} \frac{s_i}{(x - b)^i} \right)$$

can be computed by considering the two sets of pole terms, expansion about a and expansion about b . Let

$$g^{(n)}(a) := \frac{1}{n!} \left. \frac{d^n g(x)}{dx^n} \right|_{x=a},$$

and $g^{(0)}(a) = g(a)$. Then the terms at a are

$$\sum_{i=0}^{k_1-1} \frac{\sum_{0 \leq j \leq i} r_{n+j-1} g^{(j)}(a)}{(x-a)^{n-i}},$$

and symmetrically around b . These terms account for all the residues at the two locations. There are various “asymptotically fast” arguments that could be made for some of the sub-algorithms including evaluation of a function and its derivatives. However, using the obvious methods and characterizing each of the inputs as a worst case of t distinct poles gives us an $O(t^2)$ multiplication (assuming constant coefficient arithmetic costs). This would be comparable to the multiplication of two ordinary rational functions whose numerator and denominator were of length t , except that the advantage of this program, compared to the usual CAS approach, is that polynomial GCDs are entirely avoided. The precise time advantage for this program therefore depends on the cost of the chosen polynomial GCD method used by the comparison methods, but polynomial GCDs are generally considered a bottleneck³.

Further details on reducing the work in evaluating this form are given in the program listings, available from the author. We would encourage additional experimental evaluation in system contexts and tuning for other environments.

By tradition one usually asks if an algorithm can be done in parallel: in this case, if one has multiple processors, one processor could be assigned to each distinct pole of one input to the multiplier, and one or more processors could be assigned to the polynomial part. There would be no synchronization necessary for the pole \times pole products, since the residue at a pole cannot depend on residues at other poles. The polynomial \times pole product does require coordination, and the feasibility of parallel multiplication of polynomials has been well-explored in the literature.

Since some pole-product processes may require far more computation than others, it may be plausible to look for finer-grained parallelism. See also consideration of special hardware for the evaluation of a rational function at a point [10].

Note that the size of the answer may be exponential in the input: e.g. $(x^n + 1)/(x + 1)$ is of size $O(\log n)$ but the expansion is $O(n)$ terms long.

3.4 Division

We have no particularly attractive method for division R/S except to point out that computing the inverse $1/S$ reduces the problem to multiplication, once

³Exact univariate polynomial GCDs are no longer a terrible burden given the possibility of modular methods, but avoiding their computation entirely is a major benefit.

again.

We suggest converting S to a traditional form, swapping the numerator and denominator, (they are relatively prime) and finding the zeros and multiplicities of the new denominator. From this we can convert to residue/pole representation. Rootfinding is discussed further, below.

Some algorithms including matrix inversion use mostly multiplication and addition, but with occasional division; this process may be acceptably efficient in such cases.

3.5 Differentiation

This is a simple linear-time algorithm in which the polynomial part is differentiated, and for each pole term $a(x - p)^{-n}$ we shift the exponent by one to produce a term $-an(x - p)^{-n-1}$.

3.6 Integration

For the rational part of the integral, this is a simple linear-time algorithm in which the polynomial part is integrated, and for each pole term $a(x - p)^{-n}$ for $n > 1$ we shift the exponent $(a/(n - 1))/(x - p)^{-n-1}$. For the cases $a/(x - p)$ we must produce another form not in the residue/pole representation, namely a collection of $a \log(x - p)$. We suggest that these terms *not* be collected into a single logarithm via $a \log(x - p) + b \log(x - q) = \log((x - p)^a (x - q)^b)$, since a and b are in general not small integers. If they are, for example, complex floats, we do expect to have a polynomial argument to the log. Normally we would expect to see somewhat more comfortable forms for integration (avoiding full factorization unless needed, maybe using arctangents). Nonetheless, this is an effective procedure for integration, even if the representation's version of "simplest" does not correspond to the conventions typically used for small textbook problems intended to be done by hand. Some results would likely need to be massaged for human consumption.

3.7 Finding zeros

Knowing only approximations to the zeros means that when we combine rational functions originating in different calculations, we may have differing approximations for values which, if computed exactly, would be the same. If it is necessary to "cancel" such terms, we may need to refer to more elaborate mechanisms to stage a reasonable algorithm for nearby matching [13]. It is possible, if one begins with exact coefficients, to find polynomial zeros to any desired accuracy, but only by using software "bigfloat" packages. An appropriate choice of error bounds will depend on the application, but one plan might be to run the same program with increasingly higher accuracy until some level of confidence is attained. In some cases it may be possible to attempt reconstruction of exact rationals from (very close) floating-point numbers to confirm the exactness of calculations.

The cost for rootfinding for a polynomial of degree n to precision b bits has a lower bound about $O((n \log^2 n)(\log n + b))$ [14], though actual programs on realistical values of n may not exhibit such a low growth rate. McNamee’s bibliography on root finding [12], which has the important property of being updated periodically includes far more references than can be reasonably cited here. We mention that there are several implementations for polynomial root approximation which deliver results to any requested accuracy. In conjunction with bounds on root separation, such programs can be used (given exact input coefficients) to determine definitively whether apparently “close” roots are actually identical or not. Rootfinding *per se* appears to be a problem solved many times over, even in the arbitrary-precision float domain. What is not resolved is how much additional accuracy might be advisable for obtaining an adequate result *after a sequence of divisions or conversions*. It should be clear that any algorithm requiring such a sequence would likely not be appropriately coded in this representation because of the likely need for elaborate analysis. In passing, we mention that high-precision and high-accuracy root-finding programs also obviate the need to consider exponent overflow or underflow, since the representations used extend both precision and range. The recent interest in using floating-point approximations in (for example) polynomial GCD calculations may provide insight in particular applications.

4 Arithmetic with roots and poles

As an alternative to poles and residues, we will briefly consider the case where the numerator and denominator are each factored. This form, or variants of partially-factored forms, have been used to some advantage in computer systems at least since ALTRAN [4].

4.1 Multiplication

Multiplying or dividing two such rational function representations can be accomplished rather simply. In multiplication, the roots of the numerator are merged, as are the roots of the denominator. The division operation is similarly linear (in the number of terms in the input) in cost.

4.2 Addition

Unfortunately, in this representation *addition is far more expensive*, since knowing the roots of two polynomials gives essentially no information on the roots of their sum. Thus addition can be accomplished by adding rational functions in conventional form and re-factoring numerically; the cost associated with addition is higher than in the conventional form.

4.3 Differentiation, Integration

There are no great advantages in this form for these operations, except for integration, the availability of the roots of the denominator make partial-fraction expansion one step closer.

4.4 Conversion

Transferring between representations is not always quick, but can be accomplished in one direction by using the distributive law, “expanding” results; in the other, by fairly well-known algorithms once you have chosen a numeric root-finding program. The exactness that is a characteristic of the above examples of representations is a simplification, since real or complex values may not be representable as rationals, and even if the roots were exact, it is not necessarily the case that the numerical algorithms would find them. See the previous comments (on division) regarding rootfinding.

The advantages for the poles/zeros representation accrue in several areas:

Multiplication is essentially optimal: proportional to the size of the output.

Integration: rational function integration becomes trivial since partial-fraction expansion is a natural precursor to integration. There may, however, sometimes be shortcuts to rational integration requiring less work than converting to this form before integration, namely finding only those zeros of the rational function necessary to express the integral.

Evaluation, plotting, and qualitative analysis of a function can sometimes be done more rapidly or to greater accuracy. (making this statement quantitative requires some further discussion).

The ALTRAN system inspired other systems (eventually) to adopt some version of (partially) factored representation, which can work with any number of variables (over any unique factorization domain). The result of conversion to a ratio of polynomials can be done more compactly sometimes by keeping the denominator factored. The factors are clearly available, along with their multiplicities, in the residue/pole representation.

The issue that this representation requires refactoring of polynomials after each addition makes roots/poles of limited interest. We return to residue/poles representation.

5 Extensions, Limitations

Of particular interest (while contributing to significant additional pain!) are possible extensions to representations for functions of several variables (perhaps defined implicitly). We see no good way of doing this. Just to point out the problem, consider the rational function in two variables:

$$\frac{y^2 + 2y + x + 3}{y^3 + x^2(y^2 + 2y + 3) + 3y^2 + x(2y + 3) + 2y + x^3 + 1}$$

The denominator is irreducible over the rationals, and we have no convenient way of finding “linear over the complex” roots of such expressions in general. On the other hand, in some specific cases where a rational function can be expressed (say) with a denominator that is homogeneous in some set of variables, we can consider computing a collection of linear factors as denominators. This kind of activity is shown in several papers at the SNAP [15] workshops, e.g. [1]; see also earlier work on the Durand-Kerner method [2].

For purposes of exposition we do not pursue this possibility but merely contrast it to the univariate situation in which we can always find such roots, (in principle, numerically). The above expression, evaluated at $y = 0$ reduces to

$$\frac{1}{(x+1)^2} + \frac{2}{(x+1)^3}.$$

We know of no way to “lift” this solution to encompass the y dependency of the original rational function.

Instead of finding zeros, we could try factoring the denominators (in any number of variables, and perhaps over some algebraic extension field). This can provide a representation of rational functions as a sum in partial-fraction form, or as a ratio in which the denominator (and perhaps also the numerator) is factored. This is partially factored form [4]. Since we cannot depend on multivariate denominators to be substantially factorable (and we don’t really view the construction of splitting fields as advantageous) we might well see no positive effect. If the denominator doesn’t factor we end up with the usual representation: the ratio of two relatively prime polynomials.

We would also like rapid and accurate mappings between representations, especially including program forms in numerical languages.

Evaluating the usefulness of the interaction between the precision of root-finding programs and the appropriateness of this representation awaits some applications; this paper is intended to raise awareness of the representation and its consequences, not provide a complete catalog of applications.

6 Programs

Starting in November, 2003, we wrote a suite of programs for computing with univariate rational functions represented via poles and residues over “Common Lisp numbers” which include (generically), exact integer, exact rational, single- and double- precision floating-point numbers, and complex numbers (pairs of any of the previous number types). We can extend the operations to arbitrary-precision floats using one of several available packages in Lisp or other languages. When such floats are introduced, a user is immediately faced with a puzzle, namely how much precision should be specified.

We expect that once a calculation is initiated in which polynomial zero-finding programs are used, most users will be compelled to attempt subsequent calculations in complex float format; perhaps initially double-precision if that can be shown to be adequate. If overflow or loss of precision seems to present

problems, an automated but pessimistic estimate of loss of precision could be computed via interval arithmetic, and the precision increased.

Also included in the suite of programs is a (Lisp) version of the Jenkins-Traub (ACM 419) zerofinding algorithm and a determinant expansion-by-minors program.

The operations mentioned above, as well as a few testing programs, including output routines are written in portable ANSI Common Lisp, and are available from the author.

6.1 Benchmarks

It is not our intention in this section to exhaustively compare programs on a wide graded range of problems. It is our purpose to point out (essentially to an audience of designers of computer algebra systems or perhaps programmers intent on doing rational function arithmetic) that these methods have worked in at least one environment, ours. Others can obtain our code or reprogram our algorithms and see if favorable results can be obtained. We believe that some of the results are *so extremely advantageous in some computations* that an advantage is highly likely to be visible. This requires that multiplication and (especially) addition are heavily used.

We obtained these figures on a 2.59GHz Pentium 4 computer, with programs written in Common Lisp (Allegro 6.2). Consider the time to multiply

$$\left(\sum_{1 \leq i \leq 40} \frac{i}{(x-10)^i} \right) \times \left(\sum_{1 \leq i \leq 40} \frac{i}{(x-20)^i} \right).$$

Viewing each of the inputs as a pole/residue form, we obtain the exact product after about 0.54 seconds of computation. This is performing all arithmetic exactly as rational numbers. We believe that this number can be reduced by carefully recoding or using a more efficient arbitrary-precision division. Profiling shows the division, in our environment, consumes about 85% of the time. If we instead feed in machine double-precision floats, the computation takes less than 0.047 seconds⁴.

Compare this to the canonical “ratio of polynomials” form in a similar computational environment. When done exactly in Maxima, the multiplication takes about 0.93 seconds, already slightly exceeding our current time. Providing the answer in partial fraction form takes an additional 7.5 seconds.

Adding the same two expressions in our poles/residues form provides a far more startling comparison. The addition takes a time that is far too small for

⁴A simple change of the arithmetic in one place – doing evaluation at a pole – from generic (which in Common Lisp allows for any combination of exact rationals, double-precision complex, and many alternatives in between), to double-precision floating point operations provided an addition speedup of about a factor of 8. Further speedups can be expected by changing other operations and converting the arrays to packed floats instead of “pointers to floats” as is the default in Lisp.

our timer resolution; by executing it in a loop we find the time to be about 0.00001 seconds. Adding the same two expressions in Maxima takes about 0.45 seconds, about 45,000 times slower.

As indicated earlier, we include in our suite of programs a version of RPOLY, ACM algorithm 419, (1972) by Jenkins and Traub. This polynomial zero finder, (code adapted from the Maxima computer algebra system) is an old standard, but still used widely. For finding double-precision roots of a well-behaved degree 40 polynomial, the program typically takes about 15 ms. On the other hand, if the zeros have some unfortunate configuration such as many multiple zeros, the time may be much higher to allow the program to refine the zeros; it may also fail to find zeros if the refinement fails. A number of recent alternatives can be considered.

We have experimented with assuming that the denominator of p/q is given exactly, and then removing all multiple zeros of q by dividing by the gcd of q and q' , its derivative⁵. This exact calculation then allows the zerofinder to assume that all zeros are distinct. Any intimation that there are multiple zeros means the precision should be increased. This can be done with any of several library sources for arbitrary-precision zero-finders. Further discussion of the benchmarking of zero-finders is beyond the scope of this paper.

We have set up some benchmarks for determinants. We claim that, compared with traditional rational functions in fully canonical form, the pole/residue representation can be computed far faster.

First we discuss the appearance of the answer in this pole/residue form. The result naturally displays the collection of complex residues over poles of different orders. This may or may not be as compact as one that is a ratio of polynomials. In the typical cooked-up benchmarks for determinants, the best form for the answer is only obtained by factorization. Experimentation with “Cauchy matrix” forms is simple to set up and the results show a neat form. Take expressions from a class of matrices that look like this:

$$\begin{pmatrix} \frac{1}{2} & \frac{1}{x} & \frac{1}{x-1} \\ \frac{1}{x} & \frac{1}{2x-2} & \frac{1}{2x-3} \\ \frac{1}{x-1} & \frac{1}{2x-3} & \frac{1}{2x-4} \end{pmatrix}.$$

Determinants in this class can be computed at some expense as

$$\frac{x^3 - 8x^2 + 21x - 18}{32x^7 - 192x^6 + 456x^5 - 536x^4 + 312x^3 - 72x^2}$$

or at higher expense (requiring factoring) in this form:

$$\frac{(x-3)^2(x-2)}{8(x-1)^3x^2(2x-3)^2}$$

and at *very low expense* in the pole/residue form, which looks like this:

⁵It is possible to construct a quick probabilistic modular test for “free of multiple roots.”

$$\frac{0.79167}{x} + \frac{0.25}{x^2} - \frac{2.125}{x-1} - \frac{0.5}{(x-1)^3} + \frac{1.33333}{x-1.5} - \frac{0.125}{(x-1.5)^2}$$

Converting this expression to conventional form, but leaving the denominator factored (and in general chopping some small coefficients) results in this smaller display:

$$\frac{0.125x^3 - x^2 + 2.625x - 2.25}{(x-1)^3 x^2 (2x-3)^2}$$

Turning from the “display format” issue to serious timing matters, we consider a different example. Define matrices with entries of the form $g_{i,j} = \frac{1}{(x+j+i-1)^j}$. We computed the determinant of the 8×8 matrix using minor expansion [11] on a 2.53GHz Pentium 4 computer, using an implementation in Macsyma running on Allegro Common Lisp.

A typical run of the built-in program takes 262 seconds (18 in Lisp garbage collection).

Using exact poles/residues and the minor expansion method, this example uses $61+10=71$ sec, of which 92 percent is bignum arithmetic.

Much of the time is taken by bignum arithmetic, but prominent in the list of heavily-used functions are polynomial add, subtract, quotient and remainder.

Using Gaussian elimination in Macsyma takes much less time: about 11 seconds, of which 82 percent is bignum arithmetic. Using poles/residues with Gaussian elimination on this example uses $34.4 + 6.4$ (GC) = 41 seconds, and so the older method is better

What if we use in double-precision floating point? Then the results are computed using minor expansion answer takes $1.44+0.12=1.61$ seconds.

Trying some other computer algebra systems provides a mixture of timing results because of the variations possible in the form of the determinant. Thus Mathematica 4.1 can “compute” the determinant in 7.8 seconds, but in a huge form that is not simplified. Combining the terms over a common denominator seems to take much longer. I stopped it at 360 seconds. Computing the partial fraction expansion form for the determinant did not seem at all feasible.

There is another issue by which we judge the pole/residue representation superior.

Given the two representations of a determinant as (a) expanded denominator over (say) factored denominator, or (b) sum of fractions as produced by pole/residue computation, which is more accurate to evaluate numerically?

As the size of a matrix increases, the distinction grows further, but we need only consider the 4 by 4 determinant.

Evaluate form (a) and (b) at the same double-float point -4.55. The value for (a) is 6.2d-16. The value for (b) is 321.31, accurate to 7 decimal digits. If one plots the inaccurate form (a) between -5 and -4, the space appears to be filled with huge random up-and-down jags. The form (b) simply shows a kind of steep parabolic curve with a minimum of 321 at $x = -4.55$.

If the two forms are rearranged so that all the coefficients are stored as double-floats, our favored form still gets the right answer. Form (a) gives about -3.8d6, still quite wrong.

Either form, evaluated exactly in rational numbers produces the same answer (equivalent to 321.31007...).

As a side note, one could, from the very earliest design of Macsyma, create and add forms in residue/pole representation as part of its “general representation” for expressions. For modest-length expressions it is also essentially free. However, in a *sequence* of adds and *multiplies* requiring canonical forms, Macsyma will not fare well, since an efficient program for computing and maintaining a residue/pole form for a *product*, such as we have programmed, is not built-in. (The form *can* be produced by calling a partial-fraction expansion program after every multiply, or less often if we can determine when it is required, but this is expensive.) On the other side of the balance, Macsyma will systematically compute with more than one variable.

As stated earlier, profiling reveals that for our *exact* determinant benchmarks some 75 to 95 percent of the time is taken by bignum operations, which we believe can be improved (except for very low precision) through using GMP instead of native Lisp arithmetic. Some Lisps (not the one we used for these tests) use the GMP library by default. This would also substantially improve the speed of the other algorithms too.

What does this mean for systems such as Mathematica or Maple? We believe that writing the residue/pole code in those systems’ implementation language would provide substantial speed benefits, for places where this representation is applicable.

A simple alternative for pole/residue computation that might be effective in some contexts, (at least for experimentation in computer algebra systems) is to “rename” denominators. This is an old, and probably insufficient-used tactic for reducing the complexity of tasks in computer algebra. As a simple example, by renaming $a = 1/(x - 5)$ and $b = 1/(x - 6)$, the earlier equation (5) becomes

$$-28a + 126b + 7x + 7.$$

It will become important at some point to re-impose the side-relations produced from renaming, and this can be done perhaps at lower cost than by re-substitution and GCD computations. For example, knowing that $Z = a \cdot (x - 5) - 1$ is zero, we can divide an expression to get quotient Q and remainder R : $E = Q \cdot Z + R$. Since $Z = 0$, E may be replaced by R . Any time two expressions are multiplied, as in $a \cdot b$, we would have to consider whether to re-express the result as linear in a and b . In this case, $ab = b - a$, and the right-hand side would be preferable. When to apply the re-substitutions is critical to keep intermediate expression swell down.

7 Conclusions

We have proposed, implemented, and tested a representation for univariate rational functions that can, in some circumstances, provide substantial speedup in computations. We describe two (perhaps new) approaches to multiplication of these forms. In tests of individual operations, speedups of orders of magnitude are available, most particularly in rational function addition. In a more mixed environment, in computing determinants, we see a factor of 3 or more over the best previous method in Macsyma / Lisp.

The differences we note depend heavily on the speed of the underlying arithmetic as well as the cost-saving of avoiding polynomial GCD calculations.

The benchmarks are intended to be suggestive of improvements, and do not represent the best implementations. Undoubtedly more optimized programs, and especially faster arithmetic could make our specific implementation faster.

We would expect to find similar speedups if this representation were internally coded in other computer algebra systems. We expect that these ideas are of more than theoretical interest and that practical applications can be accommodated to take advantage of this representation in spite of the limitation to univariate functions.

8 Acknowledgments

This research was supported in part by NSF grant CCR-9901933 administered through the Electronics Research Laboratory, University of California, Berkeley. The Maxima computer algebra system was invaluable in helping to debug our programs, since it provided exact partial-fraction forms for our examples.

All the programs used for testing can be obtained from the author.

References

- [1] A-M. Bellido, “Numerical factorization of multivariate polynomials: symbolic computation of a numerical iteration function” in SNAP Proceedings (see below)
- [2] Durand E., *Solutions Numeriques des Equations Algebriques*, tome 1, Masson, Paris 1960.
- [3] M. Bronstein and S. Lafaille, “Solutions of Linear Ordinary Differential Equations in Terms of Special Functions,” *Proc. ISSAC 2002*, ACM Press, New York, 2002. 23—28.
- [4] W. S. Brown, “On computing with factored rational expressions,” SIGSAM Bulletin 8 ACM Press (Aug. 1974).
- [5] A. Diaz and E. Kaltofen. “FoxBox: A System for Manipulating Symbolic Objects in Black Box Representation,” *Proc. ISSAC 1998*, ACM Press, New York, 1998. 30—37.

- [6] P. Kirrinnis. “Fast Numerical Improvement of Factors of Polynomials and Partial Fractions,” *Proc. ISSAC 1998*, ACM Press, New York, 1998. 260—267.
- [7] Matlab. <http://www.mathworks.com/access/helpdesk/help/techdoc/ref/residue.shtml>
- [8] P. Chin, R. M. Corless, G. F. Corliss, “Optimization Strategies for the Floating-Point GCD”, *Proc. ISSAC 1998* ACM Press, New York, 1998. 228—235.
- [9] I. Z. Emiris, “Symbolic-numeric algebra for polynomials,” Research Report INRIA, 1997.
- [10] O. Mencer, “Rational Arithmetic Units in Computer Systems,” Ph.D dissertation, EECS, Stanford University , stanford.edu/tr/mencer.thesis.ps.gz
- [11] W.M. Gentleman and S.C. Johnson, “Analysis of Algorithms, A Case Study: Determinants of Matrices with Polynomial Entries,” *ACM Trans. on Math. Softw.* Volume 2, Issue 3 (September 1976) 232—241.
- [12] J. McNamee, “A bibliography on roots of polynomials,” (last updated to 2004) <http://www1.elsevier.com/homepage/sac/cam/mcnamee/>
- [13] V. Pan. “Numerical computation of a polynomial GCD and extensions,” Research Report 2969, INRIA, 1996.
- [14] V. Pan. “Univariate Polynomials: Nearly Optimal Algorithms for Factorization and Rootfinding”, *Proc. ISSAC 2001*, ACM Press, New York, 2001, 253—267.
- [15] SNAP Proceedings, Symbolic-Numeric Algebra for Polynomials, <http://www-sop.inria.fr/galaad/conf/1996/snap.html>.