Algebraic Manipulation

A. C. Norman

January 1994

1 Introduction

These notes cover the Lent Term lecture course on Computer Algebra. There are now a number of plausible textbooks that cover the relevant material too, but some of them have quite ridiculous prices while others key themselves very strongly to one particular (perhaps commercial) algebra system, so I find it hard to produce a clear-cut and simple recommendation on that front. I will mention some of the options in the lectures but have tried to collect the topics that need most careful documentation here. In parts of the course where I think that the lectures themselves can give adequate coverage these printed notes may degenerate to section headings or cryptic comments.

There are a number of different algebra systems available these days, and it can be useful to try examples out on one of them. This can both give you an understanding of why the various algebraic algorithms discussed here are important, and how the application of computer algebra feels. It should not matter much which system you use, but it is useful to have one single concrete syntax for examples included here. I have used REDUCE, since that is both a system that I know well and it is also available to all of you both on Phoenix and CUS. To use REDUCE on CUS you will need to set up some Unix environment variables before starting the system itself — the "man reduce" entry explains what has to be done. Phoenix users should try "help reduce" to ensure that they have up to date instructions for loading the system. REDUCE can also run on almost any other style or brand of computer you can think of, and there is a free demonstration version for MSDOS which is not capable of solving huge problems but which may be helpful for the small-scale investigations needed in support of this course. A full REDUCE manual is available on-line, but you almost certainly do not want to print it all out, so to help with experimentation and getting a feel for the system I include a glossary of the operators, keywords and switches that are provided.

People who have their own copies of Maple, Axiom, Macsyma, Mathematica or Derive will find that the same sorts of fundamental issues come up with those system as do with REDUCE, and although the details of how things are expressed will differ most of this course should still be relevant. Just as numerical analysis does not depend (much) on whether the processes are coded in Fortran, C or ADA, there is much useful that can be said about computer algebra that transcends the differences between particular packages.

This course covers Computer Algebra from two rather different perspectives. The first views it as an area in which there are many interesting and unexpected algorithms to be described. It can thus be viewed as a specialised extension of

the earlier "Data Structures and Algorithms" course illustrating the challenges of one particular application domain. In this context the nature of Computer Algebra means that it can involve looking at algorithmic interpretations or uses of what had previously seemed to be inapplicable pure mathematical results. A significant amount of essential material on the algorithms needed in computer algebra is included in Knuth's "Art of Computer Programming" (mainly Volume 2 on semi-numerical algorithms). Despite the age of that work it is still very readily available and its explanation of (for example) the polynomial GCD problem is quite sufficient for a first course.

The second thread in this course is in applications. As algebra systems become more widely available[1] many calculations which would previously have been attacked by pure numerical methods will be treated symbolically. When this succeeds the results will frequently be more general or more reliable then the results of numerical analysis. With this in view this course includes some basic ideas about the proper use of algebra systems, analogous to the hints that will be incorporated in a first course on numerical analysis. Perhaps the one overview remark in this context is that algebra systems are generally expected to produce exactly correct values — and the big problem with numerical analysis is the control or (rounding) error. So how can anything possibly go wrong if you work algebraically? The answer is that clumsy application of an algebra system can use all the memory and all the CPU time you have an still not produce a result, even in cases where alternative approaches could turn out results in very short order.

2 Computing with Polynomials

2.1 Introduction

A common exercise in the use of programming languages that support dynamic storage allocation and records is that of representing and differentiating algebraic formulae. Formulae are represented directly as parse trees. Addition, subtraction, multiplication and so on just create new tree nodes, and substitution, differentiation and some other obviously useful operations are rather easy to code up. Getting a user's input into parse tree form is just a direct application of material from a course on compiler construction. Displaying a formula to typeset quality can be harder, but is fundamentally a problem in text handling or graphics. Parse

[1] For at least the last 20 years algebra systems have been (as reported by their designers) on the verge of massive wide-scale acceptance.

trees can represent all sorts of formulae almost equally well — nodes can stand for logarithms and trigonometric functions, or summations or integrals. But it is very difficult to answer questions about formulae (for instance to detect if they are zero!) or guarantee to simplify them properly.

A very good start towards a proper development of computer algebra is to concentrate on a very limited class of expressions — polynomials. For these (made up out of coefficients [2] and indeterminates using the operations of addition and multiplication) everything can be made well-behaved. And it even turns out that many real-world applications of computer algebra require nothing more!

2.2 Canonical and Normal forms

A *Canonical Form* for a class of expressions is one that formulae in the relevant class can be converted into such that any two equivalent formulae get mapped into exactly the same structure. Note that this may not be exactly the same as a naive interpretation of the word "simplificaton". For instance a canonical form for polynomials can be obtained by multiplying out all brackets, collecting terms and then arranging the remaining terms in some standard order. But for a formula such as $(a + b + c)^{10}$ performing these steps causes a small and well-structured expression to expand into something large and rather ugly. Despite this a good basis for a polynomial algebra system will involve a canonical form.

A weaker idea than that of a canonical form is a *normal form*. Here the only requirement is that when transformed any expression that is equivalent to zero gets mapped onto the object "0". Note that with a normal form it is possible to make reliable comparisons (for equality) between pairs of expressions: to compare u and v just form $u - v$ and reduce it to its normal form — if the result is visibly "0" the two original objects were equal. Given such an equality test theoreticians will assert that a canonical form can be produced. One enumerates all valid expressions (is is not hard to show that there are only a countable number) and maps each expression onto the first item in this enumeration that it is equal to. Such a recipe is not very practical.

[2] usually just integers, but it is also reasonable, and sometimes very useful to support rational numbers.

2.3 Distributed and Recursive representations

If you multiply $(a+b+c)^2$ out you can write the result as

$$\frac{1}{1}a^2 + \frac{2}{1}ab + \frac{2}{1}ac + \frac{1}{1}b^2 + \frac{2}{1}bc + \frac{1}{1}c^2$$

where I have sorted the terms lexographically. Each term has a coefficient (which will be shown here as a rational number) and may mention several variables each raised to some integer power. This representation is known as *distributed*. With distributed representations it is often useful to suppose that there are just a few indeterminates and that their names are known in advance — then a term just has to record the exponents associated with each indeterminate. An exponent of zero allows for variables not visibly present. So for instance the first two terms of the above expression could be represented as records:

```
[1, 1 :  2, 0, 0]       % (1/1) * aˆ2 * bˆ0 * cˆ0
[2, 1 :  1, 1, 0]       % (2/1) * aˆ1 * bˆ1 * cˆ0
```

and complete polynomials are now just lists of terms. The representation is clearly easy to implement and work with.

An alternative scheme would view $(a+b+c)^2$ firstly as a polynomial in just a, which expands out to

$$1a^2 + (2(b+c))a^1 + (b+c)^2a^0.$$

Concentrating just on the top level of this structure it is made up out of terms each of which has a coefficient, a main variable and a degree. The trick is that the coefficients are now not just integers but they can be polynomials in variables that are "less important" that a. The benefit of this recursive representation is that many operations can be coded as if they were just working on univariate polynomials (an easier case), allowing the power of recursion into the coefficients deal with all the subsidiary variables.

If the terms in a polynomial are kept sorted then (in either case) adding polynomials is (almost) the same as merging lists as used in merge-sort.

Integration and differentiation of polynomials are basically easy term-by-term operations.

2.4 Polynomial multiplication: unsorted representations

When one looks at polynomial multiplication it is slightly harder. Multiplication by a single term is very easy. So multiplication by a general polynomial can be

achieved by multiplying by each of the terms in it and adding the sub-results so generated. The bad news is that (unexpectedly, perhaps) this turns out to have a bad worst case cost. If the two input polynomials have n and m terms in them then naive multiplication can have cost proportional to m^2n. To show this is unreasonable note that there are at worst mn terms in the product, so all the terms can be generated in mn stages, then ensuring that these are collected into the correct order can be done in $mn\log(mn)$ steps.

The bottleneck is ensuring that the product polynomial is created with its terms properly sorted. One way to respond to this challenge is to accept that the terms that make up a polynomial will not be kept in any special order. But now combining like terms on addition in a reasonably fast way becomes a non-obvious process. The lectures will explain an unordered merge algorithm (based on the use of hash tables) that can leave addition of unsorted polynomials fast while reducing the cost of multiplication to around mn steps. The main penalty of using it is that polynomials when printed out will be presented with their terms all jumbled up, or maybe extra sorting effort will be required in the print routines.

3 Computing with Truncated Power Series

3.1 Introduction

Many mathematical and physical problems do not have neat solutions in closed form, but do have solutions that can be expanded as series in terms of one of the variables present in the problem. Even when closed form solutions do exist, these series solutions will still usually be easier to find and almost as useful: this section is concerned with the application of algebra systems to this task. There are many techniques available for deriving series solutions to equations, and it should be expected that whichever of these is used the same results will be produced. Examples given here will show, however, that there can be dramatic differences in the amounts of computer time and memory that will be consumed when different approaches are taken to the solution of any given problem. There can also be large differences in the complexity of the mathematical and computational consequences of following different routes towards a solution, so a user has to develop an awareness of the various possibilities for compromise between simple but slow algorithms and elaborate and delicate but fast ones. When it is only necessary to produce the first few terms in the series expansion of some fairly simple function it will usually be appropriate to use the most direct method available, but even quite

modest looking problems can consume quite unreasonable amounts of computer time if attacked in too clumsy a way, and for large problems it will be essential to consider efficiency when planning solution techniques.

Even though proper power series are infinite in extent all useful computation with them will work with just their first few terms. The treatment of power series given here will treat the expansions as formal ones, that is to say it will not concern itself with the question as to whether any given series converges, or for what range of its independent variable it gives how accurate an approximation to some true result. Indeed there are some calculations that can quite properly be performed using series without any concern about convergence: for instance when the values of individual coefficients in the series are more interesting than the behaviour of the series as a whole [3]. Although truncated series are only approximations to the function that they represent, the coefficients in them are expected to be calculated exactly using exact integer and fractional arithmetic: in this way the algebraic approximation of a series has a different character to numerical schemes that use floating point arithmetic and where it is hard to produce formal descriptions of the introduced error.

The first example problem given here is somewhat artificial, but will make it possible to introduce a number of the facilities of the Reduce algebra system. It is to find the coefficient of x^{20} in the series expansion of $(1+x)^{100}$. There are a number of ways of attacking this problem. With access to an algebra system the easiest is probably just to display $(1+x)^{100}$ with brackets multiplied out and pick out the term in x^{20} by eye. The request to Reduce is just:

```
(1 + x)^100;
```

and the displayed result covers several pages. The next refinement would be to get Reduce to select out the required coefficient from the full formula. This can be done using the built-in selector function `coeffn`, which takes three arguments: an expression to inspect, the name of a variable and the degree in that variable of the coefficient required. Thus the desired value can be displayed by:

```
coeffn((1 + x)^100, x, 20);
```

A rather different technique for solving our problem makes recourse to the binomial theorem and an assertion that the desired value is just the binomial coefficient ${}^{100}C_{20}$. To compute this we can define two Reduce procedures, one for factorials and the second for binomial coefficients:

[3] eg. when the series is thought of as the expansion of some generating function.

```
procedure fact n;
     for i := 1:n product i;

procedure binom(n, r);
     fact n / (fact r * fact(n-r));

binom(100, 20);
```

This produces results noticeably more cheaply than the direct selection of the x^{20} term out of the fully expanded formula, but at the cost of requiring much more mathematical knowledge and of being a method that is not obviously adaptable to solving related problems, such as finding the coefficient of x^{20} in $(1 + x + x^2)^{70}$.

Even this scheme provides for further variations. The definition of `binom(n,r)` given above is computationally wasteful and can be rewritten as

```
procedure binom1(n, r);
    if (2*r > n) then
         (for i := r+1:n product i) / fact(n-r)
    else (for i := n-r+1:n product i) / fact r;
```

which makes an attempt to avoid multiplying factors into the result if they will subsequently be divided out. The definition `binom1` is slightly faster than `binom`, but it will only make enough difference to be noticed in cases where very heavy use is made of it. Conversely the same values can be defined in terms of a basic identity satisfied by binomial coefficients:

```
procedure binom2(n, r);
    if r=0 or r=n then 1
    else binom2(n - 1, r - 1) + binom2(n - 1, r);
```

As `n` increases the cost of using procedure `binom2(n,r)` grows very rapidly. Calculating `binom2(100,20)` would certainly require that the `binom2` was called with either `r=0` or `r=n` a number of times equal to the value eventually to be returned, i.e. 535983370403809682970. On the fastest Reduce implementations on large mainframes it could be that a million such exits could be taken per second, in which case the entire calculation would complete in about 17 million years. Even `binom2(20,10)` a very substantial calculation.

As a final attack on the original problem it is worth considering another mathematical result that states that the terms in a series expansion are related to the derivatives of the function being expanded. In Reduce the derivative of `y` with

respect to `x` is obtained by writing `df(y,x)`, and the nth derivative can be requested as `df(y,x,n)`. Evaluating an expression at a value `v` for the variable `x` is performed using a construct of the form `sub(x=v,`$<$*expression*$>$`)`. Using these we can obtain the coefficient we require by writing just:

```
sub(x=0, df((1 + x)^100, x, 20)) / fact 20;
```

The method shown earlier using `coeffn` was only applicable if the expression being decomposed was built up using just the operators $+$, $-$ and $*$. This scheme using differentiation is a little more expensive but can be applied to a much larger range of functions.

3.2 Simple iterative methods

When series expansions are produced it is almost invariably the case that intermediate calculations generate terms of much higher degree than will be required in the final result. It makes sense to discard such unwanted terms as early in the course of a computation as possible. In Reduce if terms in x with degree n or more are to be ignored a directive of the form

```
let x^n = 0;
```

should be issued. This indicates that until further notice any terms involving x^n or higher powers of x should be suppressed. If at some later stage in a calculation this action is not required the rule introduced by the "`LET`" statement can be removed by

```
clear x^n;
```

In the code given earlier to find the coefficient of x^{20} in a large formula it would have been sensible to issue

```
let x^21 = 0;
```

before calculating $(1 + x)^{100}$ thereby avoiding a large proportion of the work that want into finding high degree terms in the full expansion. When Reduce is used to compute with power series in some variable, x say, the readability of the results it produces is often enhanced by setting some output control flags by saying:

```
factor x;
on div,revpri;
```

These do not have any effect on Reduce's internal workings or the values of the algebraic results that it produces: they just control the format in which results are displayed. Their effects can be cancelled to return to Reduce's default print style by

```
remfac x;
off div,revpri;
```

With these flags set and a truncation established so that powers of `x` higher than (say) the 10th will not appear it is possible to consider some techniques for exploiting power series arithmetic.

The first of these to be considered is Picard's method for solving ordinary differential equations, which will be illustrated using the equation $t' = 1 + t^2$ (which of course has the exact solution $t = \tan(x)$ provided $t(0) = 0$). Picard's method develops a solution to the equation starting from some initial approximation by repeatedly substituting its approximation into the right hand side of the equation and integrating to get a new approximation:

```
let xˆ11 = 0;
factor x; on div;
tt := 0;
for i := 1 : 6 do
    write tt := int(1 + ttˆ2, x);
```

Running this program[4] will cause Reduce to display a succession of values for the power series, and it will be observed that the initial terms in the series stabilise rapidly, and that in this particular case each integration leads to one more correct term in the series. Note that the above code relied on the Reduce integration operator `int` handing back a result with a zero constant of integration. It is also worth noting how for even rather simple calculations the rational number coefficients that get generated can become quite complicated.

For an example differential equation where the required solution has a nonzero value at $x = 0$ consider $r' = -r^2 g'$ for some known power series g (here we will use $g = 1 + x$). By choosing $r(0) = 1/g(0)$ this has an exact solution $r = 1/g$. Assuming that a suitable let statement is in force, the Reduce code is

```
g := 1 + x;
```

[4]The code uses `tt` rather than just `t` as the variable since REDUCE reserves plain `t` and will get upset if you try to use it here.

```
r := r0 := 1 / sub(x=0, g);
for i := 1:10 do r := r0 + int(-r^2*df(g, x), x);
```

where it should be remarked that `sub(x=`<*value*>`,`<*expression*>`)` makes the specified substitution for `x` in the given expression (used here to evaluate `g` at `x=0`), and `df(g,x)` stands for the derivative of `g` with respect to `x`. With `g` set up as shown this program computes the series $1 - x + x^2 - x^3 + \ldots$ which is perhaps not very exciting. However if its initial line is altered it can compute an expansion of the reciprocal of any power series despite the fact that Reduce does not have a built in capability for series division, and this is worthy of note. As before this Picard iteration exhibits first order convergence: to obtain n correct terms in the results it is necessary to perform the integration step n times.

In cases where the function defined by a differential equation is known, as in the two examples given so far, the terms in a series expansion can be found by differentiation, as in

```
tt := for i := 0:10 sum
          (sub(x=0, df(tan x, x, i)) / fact i);
```

which should generate the same series for `tt` as was produced earlier. In many cases it turns out that the Picard iteration produces results faster even though it at first seems to be doing more work. For instance a test on the $\tan(x)$ expansion finding terms up to degree 20 in x showed Picard's method to be about 50% faster than repeated differentiation. Obviously the cost difference will vary greatly from problem to problem, and in some cases it will involve order of magnitude differences.

The important thing in the above iterations is not the presence of an integration operator, but the fact that if an expansion correct to order n in x is substituted into the right hand side result will be an expansion correct to some higher order. This can be achieved by multiplication by x as well as by integration, as in

```
r := 1;
for i := 1:10 do write r := 1 + x*r;
```

which provides another way to compute $1/(1-x)$, and which can easily be adjusted to expand other quotients in series form. Iterations of this form can often be derived by just separating an equation into a set of leading terms that do not depend on x and a correction factor that does. Even in cases where this does not at first seem possible a little rearrangement can help: consider the problem of expanding $\sqrt{1+x}$ as a power series. To find an iterative formula for deriving the

expansion it is necessary to put in a leading term for the expansion, and so express the square root in the form $1 + xq$. Then the fact that this is the square root of $1 + x$ amounts to the identity

$$(1 + q)^2 = 1 + x$$

and after a very small amount of rearrangement this leads to the iteration

```
q := 1;
for i := 1:10 do q := (1 + x*q^2) / 2;
```

which is certainly easier to program than a formula based on use of the binomial expansions.

A somewhat similar form of rearrangement can be needed if a differential equation has terms in it which are multiplied by powers of x, as in Legendre's equation

$$(1 - x^2)y'' - 2xy' + n(n + 1)y = 0$$

where the need to divide by $(1 - x^2)$ can be avoided by rearranging the equation to give a recurrence rule

```
dy := dy0 + int(x^2*df(dy, x) + 2*x*dy - n*(n+1)*y, x);
y := y0 + int(dy, x);
```

with `y0` and `dy0` providing initial conditions. Observe that in this case it is reasonable to use the derivative of `dy` in the right hand side of the first assignment because it is immediately multiplied by $\texttt{x}^2$, which compensates for the shift in order produced in the differentiation.

The general idea behind Picard's iteration is that evaluating the right hand side of the recurrence formula must produce a result correct to a higher degree in x than the previously best known approximation to the solution y. In many cases each iteration will just increase the order of accuracy of the solution by one, but in some cases (e.g. the one just given) each iteration may increase accuracy by two (or even more) terms.

3.3 Newton's method and second order convergence

Probably the best known iterative technique in numerical analysis is Newton's method. To find a solution to the equation $f(z) = 0$ it starts with some initial approximation z_0 and defines a sequence of further approximations by

$$z_{n+1} = z_n - f(z_n)/f'(z_n).$$

Except when the solution for z that is being found is a repeated root of $f(z) = 0$ this iteration exhibits second order convergence, ie. the number of correct digits in the approximations z_n roughly doubles each time n increases by one. This very general iteration can be applied in the context of power series calculations. The initial approximation can almost always be taken to be just the leading (constant) term of the desired series, and this is usually trivial to find. The general form of the iteration involves a power series division, and although some algebra systems support this directly Reduce does not. However the required quotient can be computed using a special case of the Newton iteration itself. Consider the function $f(z) = y - 1/z$, then $f(z) = 0$ will be solved when $z = 1/y$. $f'(z) = 1/z^2$, and so Newton's iteration simplifies to something in which no division is present. A suitable starting value for z_0 will be obtained by substituting $x = 0$ into the power series y to obtain the leading term, which being numeric can be divided by. Thus the program for computing a power series a for $1/y$ becomes

```
z := 1 / sub(x=0, y);
for i := 1:n do z := z*(2 - y*z);
```

together with a suitable let statement that will avoid terms with too high a degree in x from accumulating. Second order convergence in this context means that each step in the iteration doubles the number of correct terms in the series, and so if for example an expansion correct to terms in x^{16} is required it will be necessary to make the iteration count (`n` in the program fragment) four. This iterative scheme for computing the reciprocal of a power series can be packaged to provide a general series division capability for Reduce by encapsulating it in a procedure:

```
let x^16 = 0;

procedure tpsquotient(a, b, x);
   begin
      scalar z, z1;
      z := 1 / sub(x=0, b);
      repeat << z1 := z;
                z := z*(2 - z*b)
             >> until z = z1;
      return a*z
   end;
```

The procedure tpsquotient will always perform one unnecessary iteration at the end of computing z, the reciprocal of b, since it decides when to stop by

observing when Newton's formula does not lead to a change in the value of z, but in almost all circumstances this small inefficiency will be unimportant. Some of the work done by the above procedure is unnecessary for another reason: during early stages in the iteration it is known that the intermediate results will only be accurate to low order in terms of powers of x, but the global `LET X^16 = 0` reduction does not reflect this. A more refined version of the algorithm would arrange to truncate all intermediate results to keep the smallest possible number of terms in them. With Reduce the heavy use of `LET` and `CLEAR` that this would involve is clumsy, and for second order (where only a very few cycles of the iteration are needed) the extra complication is usually not worthwhile.

Having synthesised a power series division procedure it is now easy to use Newton's method to more elaborate equations. For instance finding $y = \sqrt{a}$ where a is some formula such as $1 + x$ can easily be achieved by starting with

```
y := sqrt(sub(x=0, a));
```

and using the iteration

```
y := (y + tpsquotient(a, y, x)) / 2;
```

and for simple formulae a the results will match those that could have been predicted using binomial expansions. The scheme is, however, equally easily used on problems where the solution can not obviously be obtained otherwise. Consider the equation

$$2y^3 - y = 1 + x$$

and the problem of expanding its solution y as a power series in x. It is first necessary to consider an initial approximation, and this can be found by discarding all instances of x from the original problem to leave

$$2y^3 - y = 1$$

which has a solution $y = 1$. For this particular equation the other potential initial values for y are complex, and it might be that the originator of the problem can indicate that only a real solution is required: otherwise it would also be necessary to consider the other possible starting values for y, viz the complex numbers $(-1+i)/2$ and $(-1-i)/2$.

Newton's formula then dictates the iteration to be used:

$$y_{n+1} = y_n - (2y_n^3 - y_n - 1 - x)/(6y_n^2 - 1)$$

and this translates directly into a program

```
y := 1;
for i := 1:4 do
    y := y - tpsquotient(2*y^3-y-1-x, 6*y^2-1, x);
```

As for the reciprocal program the number of correct terms in the result double each time the iteration is used, and so a very small number of cycles of the loop will usually be adequate. By changing the first line of the above from `Y:=1;` to `Y:=(1+I)/2;` the same program would find an expansion for one of the other solutions to the original equation. In a similar way the program that finds $\sqrt{1+x}$ using Newton's method will naturally find the other solution to $y^2 = 1 + x$, i.e. $-\sqrt{1+x}$, if started with -1 as an initial value for y.

Applying the iteration

$$z_{n+1} = z_n - f(z_n)/f'(z_n)$$

involves repeated evaluation of the derivative $f'(z_n)$ and division by it, both of which may be expensive operations. A simple modification of the Newton Raphson iteration keeps using the initial approximation to f' (which is often just a number), so that each step in the iteration is faster:

$$z_{n+1} = z_n - f(z_n)/f'(z_0).$$

This modified Newton's method generally exhibits first order convergence, but in cases where the function f' is more complicated than f the reduced cost per step can result in it be more efficient overall.

3.4 The use of undetermined coefficients

The discussion so far has concentrated on the order of accuracy of series expansions. In the following section an alternative view will be presented. This considers the leading error term in an approximation, and attempts to eliminate it. The result will generally be a new approximation where the leading is of higher degree, and so this new leading error term will be eliminated next. When expressions are dense (ie. almost all possible terms in the expression are present) there is no practical difference between repeated approximation methods expressed as iterations over the degrees of formulae and those thought of as the successive elimination of leading error terms. For sparse expressions, however, term by term methods can lead to useful saving.

3.4.1 Series reversion

Consider the problem where a function $y(x)$ is defined by a power series

$$y = y_0 + y_1 x + y_2 x^2 + \ldots$$

and it is desired to express x as a function of y, also in series form:

$$x = x_0 + x_1 y + x_2 y^2 + \ldots$$

The coefficients x_i can be derived using a repeated approximation algorithm. It is first necessary to exhibit the initial terms of the expansion. By drawing a graph of y against x it can be seen that the problem is only a sensible one if $y_0 = x_0 = 0$, and in that case $x_1 = 1/y_1$. The dependence of the remaining x_i on the known coefficients y_i can be derived in a step by step manner. Thus to find x_2 a provisional expansion for x is set up with a new indeterminate acting as an undetermined value for x_2. Here the symbol z will be used, and so to order 2 in y,

$$x = (1/y_1)y + zy^2.$$

This can now be substituted into the original identity, keeping only those terms with degree no higher than 2 in y. The two sides of the identity should be in agreement in their constant and linear terms (because the those terms in the expansion for x were supposed to be correct already), and so the y^2 term can be used to give an equation to be solved for z. This establishes that $x_2 = -y_2/y_1^3$. If higher order terms are required the same sequence of steps can be repeated. Re-using the symbol z, to order 3 in y,

$$x = (1/y_1)y - y_2/y_1^3 y^2 + zy^3.$$

and substituting this into the equation for y in terms of x will again lead to an equation which can be solved to find x_3, with similar calculations leading to as many more terms in the expansion for x as are required.

3.4.2 Series techniques with other than simple polynomials around

Imagine a polynomial in the variables a, b, c and u, v and w. Now imagine that u stands for e^{ix}, v for e^{iy} and w for e^{iz}. Basic arithmetic on the polynomials is not altered by their interpretation as complex exponentials. The rules for integration and differentiation need altering, but not in very dramatic ways. The effect is that

for very little extra cost rather broader class of expressions can be handled. But who wants to worry with complex exponentials? Well the magic step is to write

$$\sin x = (e^{ix} - e^{-ix})/(2i)$$

and similarly for $\cos(x)$ — suddenly a system capable of dealing with complex exponentials can deal with formula involving a bunch of polynomial style variables and a collection of trig functions. The restricted sort of series with sines and cosines in are knows as Poisson Series.

As an example of repeated approximation applied to Poisson Series it is almost easy to generate an expansion of the solution to the Kepler equation,

$$e = u + x\sin(e)$$

by starting with a first approximation $e_0 = 0$ and repeatedly substituting into the right hand side of the equation. The calculation $\sin(e)$ can be performed by substituting the Poisson series approximation for e into the power series for the $\sin$ function.

3.5 History, state of the art, future prospects

3.5.1 Delaunay's analytic lunar theory

A major classical use of Poisson Series was in producing an analytic Lunar Theory. One of the early triumphs of computer algebra was the reproduction of a series of massive calculations that had been performed by hand by the Frenchman Ch. Delaunay. Since then the same technology has been applied to artificial satellite theory.

3.5.2 Selection done during multiplication rather than afterwards

In repeated approximation methods many operations are performed in circumstances where it is known that high order terms in the result will not be meaningful. Major savings in both time and space can be achieved if the polynomial multiplication procedures are adjusted so that they respect the cut-off (known as a *selection*) and avoid generating parts of the result that would be of too high an order.

3.5.3 Specialised power series packages

Various special purpose computer algebra systems have been written at various times with the specific aim of making polynomial, power series and Poisson series working as fast as possible. By fixing the names of variables and limiting the magnitude of exponents it becomes possible to use very tightly packed and neat datastructures, which again helps speed and space efficiency. In a Cambridge context the main system to note is CAMAL. As computers have become larger and faster the special purpose algebra systems have somewhat fallen from favour.

3.6 Case studies

3.6.1 Legendre polynomials done lots of different ways

This section tries to illustrate that even for polynomial and series calculations there can be many very varied ways of calculating the same values. These may differ radically in terms of programming convenience or the demands that they place upon an algebra system. You might like to try the following out to see which are easier to get working and which run fastest.

1. Using pure polynomial arithmetic the Legendre Polynomials can be computed using a recurrence formula.

$$\begin{aligned} p_0(x) &= 1 \\ p_1(x) &= x \\ p_n(x) &= ((2n-1)xp_{n-1}(x) - (n-1)p_{n-2}(x))/n \end{aligned}$$

2. The formula $d^n/dx^n(1-x^2)^n/n!$ (Roderigue's formula) computes the same polynomials using differentiation of polynomials.

3. If you write the power series expansion

$$\frac{1}{\sqrt{1-2x^2t+t^2}} = \sum_{i=0}^{\infty} p_i(x)t^i$$

   and work out explicit values for the coefficients $p_i(x)$ then once again you will have found the Legendre polynomials.

4. The differential equation $(1-x^2)y''-2xy'+n(n+1)y = 0$ has a polynomial solution that is the nth Legendre polynomial.

3.6.2 Van der Pol equation

Consider the equation

$$y'' + y = ey'(1 - y^2)$$

subject to constraint $y'(0) = 0$. If e is small this is nearly just a simple harmonic oscillator. For small oscillations (ie. the average value of y is small) if e is positive the amplitude of the oscillations will tend to increase. For large amplitudes on average $1 - y^2$ will be negative and this will damp things down. Somewhere in between there is a stable state — a *limit cycle*. I will sketch (but not give full details of) how Poisson Series can be used to find the limit cycle. The technique is applicable to a wide range of weakly nonlinear periodic and and almost periodic systems, and can be thought of as a very simplified model of what was involved in lunar and satellite theory analysis.

3.7 Exercises

1. Compare performance for lots of ways of computing $\sqrt{1+x}$, $1/(1-x)$, $\tan(x)$ series.

2. Tchebychev polys: check them out in a suitable book and see how many different ways of evaluating them you can invent. For instance $T_n(x) = \cos(n \arccos(x))$.

3. Show how to interpolate a polynomial through (x_1, y_1), (x_2, y_2), ...(x_n, y_n)

4. The Duffing equation is $y'' + y = ey^3$. Compare with Van Der Pol.

4 Rational Functions

4.1 Introduction

In section two all expressions were put in the form of series. This made it possible to generate expansions of the solution to various algebraic and differential equations, but meant that a simple fraction such as $1/(1-x)$ had to be represented by some initial segment of the infinite series $1 + x + x^2 + \ldots$

This transformation can be avoided if quotients are kept as such. The main technical problem that this raises for algebra system implementors is one of keeping the resulting fractions reduced to their lowest terms. Even when high powered

algorithms are used this process can be unexpectedly costly when the formulae being processed are of high degree or involve many different indeterminates, and so the first part of this section discusses ways of reducing the need for the calculation of greatest common divisors (GCDs) in calculations. Of course for sufficiently small problems the techniques described will represent irrelevant complication, but as progressively larger computations are attempted it will often be the case that GCD calculation limits what can be done on a given computer in a reasonable amount of time.

The procedures discussed here are applicable to formulae built up out of integers and indeterminates using addition, subtraction, multiplication and division. The word "polynomial" will be taken to apply to those formulae with no division in them (not even fractional coefficients), and all other cases will be treated as quotients of pairs of polynomials. Such quotients will be called *rational functions*. Thus $(1/2)x + (1/3)$ will be thought of as a rational function with the polynomial $3x + 2$ as its numerator and the constant polynomial 6 as its denominator. This is in fact the way that the formula is represented inside the Reduce algebra system, and the `ON DIV` flag used in 3.2 just changes the style in which results are printed, not the internal organisation of the system. A consequence of this is that in Reduce some built-in functions that require polynomials as arguments (e.g. `COEFF`, `REDUCE`) take a strict view and expect to be given expressions with whole number (and not fractional) coefficients.

The remainder of this section investigates the algorithms used for rational function manipulation and various closely related processes (e.g. factorisation). Some understanding of these can help a user appreciate why large GCD calculations can be so extraordinarily costly, while others that seem at first sight to be as complicated are completed very rapidly. The algorithms also provide an illustration of the way in which techniques derived from abstract "modern" algebra find a direct applicability in the solution of apparently elementary problems. Some of the algorithms will be illustrated by code fragments showing how they could be implemented by a Reduce user: these may form prototypes for user written packages for other mathematical procedures.

4.2 Reducing the need for GCD calculations

In many cases when a fraction p/q occurs in the course of an algebraic calculation, p and q will have no common factors. If they do have a common factor, say $g = \gcd(p, q)$, then the quotient should be represented as $(p/g)/(q/g)$, where both divisions by g will be exact. For uniformity it will be normal to ensure that

the leading coefficient in the denominator q is positive, if necessary by multiplying top and bottom of the fraction by -1.

Throughout a calculation the repeated checking of fractions for factors that need cancelling represents an overhead. This is particularly so if it happens that no significant gcds are ever found. Unfortunately simple gcd algorithms exhibit their worst behaviour (ie. they consume most time and store) precisely when they are eventually going to report that their inputs are coprime, and so with early algebra systems the size of problem that could be solved was strongly limited by the form of the rational function arithmetic involved. With the larger memories and better algorithms now available many more algebraic problems can be handled without gcd calculation giving trouble, but it is still the case that for large calculations it can still be necessary to take steps to reduce the costs associated with reducing fractions to their lowest terms. Thus most problems should initially be presented to the algebra system as if rational function manipulation were not a problem: if resource limits prevent the required results from being obtained it may be worth considering some of the transformations discussed below.

The first principal to be applied when performing large scale algebra is to avoid computing a result that contains more information than is actually required. Apparently equivalent ways of writing even quite simple fragments of code can lead to very different computational behaviours. For instance given four polynomials p, q, r and s it may be necessary to decide if $p/q = r/s$. The obvious test to apply would be

```
if (p/q) = (r/s) then ...
```

but this involves reducing the fractions p/q and r/s to their lowest terms, which will turn out to be unnecessary. A second attempt, which although mathematically equivalent to the above will be computationally different is

```
if (p/q) - (r/s) = 0 then ...
```

where not only are p/q and r/s reduced to their lowest terms, but if nonzero their difference also has to be so reduced. Since the only information that is required is whether the difference is zero, it will probably be better to concentrate on its numerator, avoiding all gcd calculations and writing

```
if p*s = r*q then ...
```

Another example in the same vein involves a test to see if the polynomial q divides exactly into p. The test

```
r := p/q;
if den r = 1 then ...
```

forms the rational function (p/q) and tests if its denominator is one. If so it is clear that q divided exactly into p, and in this case the quotient r will often be required for further processing. If q is not a factor of p the above code will reduce the fraction p/q to its lowest terms, and if the reduced fraction is not needed computing it will be a waste of time: the test

```
if remainder(p, q) = 0 then ...
```

does not suffer from this, but does have the disadvantage that it does not automatically provide a value for the quotient p/q in the exact division case, even though the process of producing the remainder will have had to go through the steps needed to obtain it. Later on there will be an explanation of how users can obtain access to the function that Reduce uses internally for test division[5], thus providing a way of obtaining further improved performance at the cost of having to understand more about the fine details of the algebra system being used.

In general if calculations are performed on rational functions and fractions are not reduced to lowest terms promptly the size of expressions grows explosively. In special cases, however, it can be predicted in advance that this will not be so: for instance it may be known that some series of calculations will necessarily lead to an answer expressed in its lowest terms even without any gcd calculations being performed to ensure this.

In such cases it may be advantageous to instruct the algebra system not to attempt to reduce fractions to lowest terms. In Reduce this is done with a directive

```
off gcd;
```

the effect of which can be cancelled by `ON GCD`. Since calculations which benefit from this treatment are fairly rare it seems possible that the best use of `OFF GCD` mode is to allow the user to see how dramatically expressions can grow if not kept in reduced form, and hence how important a good gcd algorithm is in an algebra system!

Having adjusted a program so that it does not compute more than the minimal interesting part of an answer it can be becomes useful to seek ways of simplifying the problem formulation. Any transformation that reduces the degrees of polynomials found in the problem or which reduces the number of variables present will

[5] See "testdivide" defined shortly.

be useful, and to a lesser extent transformations which remove large numeric values can help. It is often possible to make changes of variables so that factors that appear in the denominators of intermediate expressions become simply powers of some indeterminate — this renders greatest common divisor extraction trivial. Thus if in some calculation there are two variables u and v, and the denominator of many fractions will involve powers of $u + v$ it will be useful to introduce a new symbol (w say) to represent $u + v$ and use a substitution such as

```
... sub(v = w - u, <formula>) ...
```

to express original expressions in terms of it. A simple substitution at the end of the calculation will then make it possible to present results in terms of the original variables. It can make sense to introduce a new indeterminate to represent the reciprocal of some complicated denominator even when it is not then possible to eliminate any of the original variables. For instance in calculations involving a matrix M and its inverse, the introduction of a symbol that stands for $1/\text{determinant}(M)$ can often simplify things substantially.

In some cases it is possible to reformulate calculations so that rational function arithmetic is not needed at all. If, for instance, there is a way of predicting in advance some common denominator for all the expressions that will arise throughout a calculation then everything can be reduced to polynomial arithmetic, which is generally well behaved. Later in this section (in the discussion of the *subresultant PRS* method for computing gcds) there will be an example of a case where the mathematical structure of a calculation makes it possible to predict factors that are guaranteed to divide exactly into some of the intermediate results, and this dividing out of predicted exact factors saves significant amounts of time.

The costs of computing greatest common divisors can grow rapidly with the size of the expressions involved, and this means that it is generally better to perform several small gcd calculations rather than allowing common factors to build up and finally do a single large reduction. An application of this principle can be found inside the implementation of Reduce where two rational functions are to be added. The natural way of forming the sum of (a/b) and (c/d) would be to compute the numerator $p = ad + cb$ and the denominator $q = bd$ of the result and then form them into a fraction (p/q) cancelling any common factors. The scheme that is in fact used tries to find common factors as early as possible so as to reduce the size of intermediate results and hence the cost of the final gcd calculation. First the greatest common divisor, g, of b and d is computed. Now the numerator of the final result can be evaluated as $p = a(d/g) + c(b/g)$ and the denominator as $q = b(d/g)$ where the divisions indicated are known to be ones that will be exact.

If it happens that g is large (for instance b and d might be the same) this will result in significantly smaller values for p and q. In forming the final quotient (p/q) it is still necessary to look for further common factors. If b and d are coprime the new method is no help, but in many realistic cases it speeds calculations up by a useful amount. Analogous optimizations are possible when forming products and quotients of rational functions, and potentially within any user-written code that ever generates the numerator of a rational function separately from its denominator.

A scheme that has been proposed (but not widely adopted) that subsumes several of the above ideas is that of representing expressions as *partially factored forms*. Rather than keeping all brackets multiplied out and the numerators and denominators of expressions separate, a partially factored representation holds values as products of powers of items. Negative powers are used to indicated factors that belong in the denominator of the expression. During multiplication and division common factors are merged, and in some cases gcd calculations will reveal further factorisations of some of the terms. For instance if the product of $x^2 - 1$ and $x + 1$ were to be required the process of merging lead to the result $(x - 1)(x + 1)^2$. During addition and subtraction and factors common to the expressions being combined will remain separated out, but other terms will have to be expanded out thus destroying knowledge of their factor structure.

The final suggestion here for reducing the demands on a gcd procedure involves breaking away from the closed form representation of quotients and dropping back to the use of power series. Section 4.4.2 illustrates how it is possible to recover an exact rational result from its power series representation subject only to the need to have a bound on the degrees of the numerator and denominator of the expected rational function. For univariate calculations where it is known a priori that the result will be fairly simple but where intermediate stages in the working would lead to rational functions with very high degree denominators this radical transformation can perform wonders.

To end this section here is the REDUCE code for test division mentioned earlier. As defined here it returns the quotient of its arguments if the division involved is exact. Otherwise it returns 0. It expects to be given polynomial arguments, but does not check. The explanation of the incantation used to provide this interface to a system-level function within Reduce is outside the scope of this section.

```
symbolic procedure testdivide(p, q);
   mk!*sq (quotf(numr simp!* p, numr simp!* q) ./ 1);

flag('(testdivide), 'opfn);
```

4.3 Polynomial Remainder Sequences

4.3.1 Integer GCD

Computing the greatest common divisor of a pair of positive integers can be done simply and efficiently using Euclid's algorithm:

```
PROCEDURE numeric_gcd(a, b);
   IF b = 0 THEN a
   ELSE numeric_gcd(b, remainder(a, b));
```

4.3.2 Polynomial Remainder and pseudo-remainder

The same idea can be applied to find polynomials once we understand what is meant by the "remainder" operation in that case. Dividing one univariate polynomial by another is a pretty straightforward process and what gets left at the end is clearly the remainder. But observe the case when $x^2 + 1$ is divided by $2x - 1$: the quotient comes out as $\frac{1}{2}x + \frac{1}{4}$ and the remainder as $\frac{3}{4}$. The fractional coefficients here are not a help at all, and turn out not to contribute usefully to GCD calculations, so it is normal to get rid of them. A few moment's though will show that the only denominators that can be introduced are powers of the leading coefficient of the polynomial that is being divided by. If, before the remaindering step, the other polynomial is multiplied by a suitable power of same then the unwanted fractions will not appear. This variation on computing a remainder is known as the *pseudo-remainder* of the two polynomials.

If the recursive representation is used for polynomials then a scheme the computes univariate pseudo-remainders can be applied directly (to the top level of the recursive datastructure) to obtain multi-variate remainders. But note now that the result obtained will depend on the ordering of variables in the structure. You may like to try out a very simple case — evaluate the pseudo-remainder when $a^2 + b^2$ is divided by $2a + b$ first when each polynomial is treated as being in terms of a main variable a then in terms of b.

4.3.3 a PRS to compute a GCD

If a polynomial $a(x)$ is written $a_n x^n + \ldots + a_1 x + a_0$ then we defined the *content* of a to be $\gcd(a_n, \ldots, a_0)$. For instance the contents of $2yx^2 + 4(y^3 - 1)x - 8y + 4$ (when viewed as a polynomial in x) is just 2. The content of a polynomial does not have to be just number if all the coefficients have some common factor that

involves subordinate variables, as in $(y-1)x+(y-1)$ which clearly has contents $y-1$. If a polynomial has a content of 1 it is referred to as being *primitive* and a dividing a polynomial by its content gives its *primitive part*.

Now suppose we have two primitive polynomials and want their GCD. The GCD will also be primitive. Is this obvious to you? Use the Euclidean GCD algorithm starting with the two inputs, but doing pseudo-remainders rather than true remainders all the way. At the end you will have a polynomial that is *almost* the desired GCD. But the pseudo-remaindering operation can have left unwanted extra coefficient-domain factors multiplied through, so the true answer wanted is just the primitive part of what comes out of the Euclidean method. You might suspect that the need for this final calculation of a primitive part was just because we used pseudo- rather than real remainders, but in fact use of real remainders would have left in equally nasty stray multipliers, but ones that could be fractional and hence even more painful to handle. If you work through a small example (I suggest x^2-1 and x^2+2x+1) you will see what happens quite clearly — it is not very difficult. The series of results computed during the gcd process is a *polynomial remainder sequence* or PRS.

Now I can take the GCD of pairs of primitive polynomials: what about the general case. Well that can now be achieved by splitting each of my real input polynomials into contents and primitive parts. Taking the gcds of the contents is a calculation in the coefficient domain so I suppose (by recursion) that it is easy. Taking the gcd of the primitive parts is what I have just covered. I finally multiply the two sub-results together. Easy!

4.3.4 Euclidean, Primitive, Reduced, Subresultant, Trial division

This section is not written out in detail here because Knuth covers it pretty well — but the main messages are:

1. If you compute a PRS of even medium-degree polynomials the coefficients in the intermediate results can become huge, and working with them dominates the computing time of the entire process. A simple GCD procedure is not useful for serious applications.

2. The coefficient growth can be cured somewhat by taking primitive parts at each step in the generation of the PRS. But then the cost of all the coefficient GCDs involved in working out the contents that are to be divided by becomes an embarrassment.

3. Some clever mathematics (not covered in this course) makes it possible to predict factors that can be divided out. These factors are in fact just certain powers of the leading coefficients of polynomials that arose earlier in the PRS. The effect is that at least almost always the bad growth in coefficients can be cured at quite modest cost.

4. Since the values divided out by the *Reduced* and *Sub-resultant* methods are just powers of previous leading coefficients, an easier scheme to program and understand just keeps a list of these previous coefficients and uses test division to remove unwanted factors of them. The test division turns out to be much cheaper than the gcds that were at first needed to keep everything primitive.

4.3.5 Eliminating main variable between two polys: Resultant

This is maybe an aside: if you have two equations each in two unknowns you may want to eliminate one of the variables between them. If you look at the steps that you take you will find that you are computing a PRS, and that the univariate equation you end up with is just the final polynomial in that sequence. Moral: PRSs arise in GCD calculations but have other uses too.

4.3.6 PRS as Gaussian elimination on a certain matrix

Imagine two polynomials a and b with degrees n and m and coefficients a_i and b_i. Consider the matrix

$$\begin{vmatrix}
a_n & a_{n-1} & a_{n-2} & \ldots & & & \\
0 & a_n & a_{n-1} & \ldots & & & \\
0 & 0 & a_n & \ldots & & & \\
\vdots & & & & & & \\
 & & & \ldots & a_1 & a_0 & 0 \\
 & & & \ldots & a_2 & a_1 & a_0 \\
b_m & b_{m-1} & b_{m-1} & \ldots & & & \\
0 & b_m & b_{m-2} & \ldots & & & \\
\vdots & & & & & & \\
 & & & \ldots & b_1 & b_0 & 0 \\
 & & & \ldots & b_2 & b_1 & b_0
\end{vmatrix}$$

If you imagine doing Gaussian Elimination on this matrix you will rapidly discover that the steps you take bear an uncanny resemblance to those that need to be used when computing a PRS. The determinant of the above matrix is known as the *resultant* of the two polynomials a and b with respect to the variable x. The importance of resultants (here) is that they show a perhaps unexpected link between PRS calculation and simple linear algebra (ie. solving simultaneous linear equations and computing determinants). This link is at the root of the mathematics that led to the Reduced and Sub-resultant GCD algorithms. This course does not provide me with sufficient time to explore it further!

4.4 Evaluation/Interpolation methods

The PRS methods for computing GCDs work, but a serious problem with them is that their worst case is when the GCD that is to be found is just 1. In that case the remainder sequence will be of maximum length and despite all attempts to keep them small intermediate coefficients still grow pretty alarmingly. But if you take two polynomials at random their GCD is probably 1. It would be nice to have a method which behaved best in this especially common case.

4.4.1 Modular Probabilistic Checking

Consider first the univariate case. The only big cost in computing a PRS is because the coefficients tend to get big. Try *approximate* coefficient arithmetic to prevent this. In particular, choose a prime p and perform all arithmetic modulo p. If p is a single precision integer then this avoids all the painful overhead of allowing for multiple-precision intermediate results. But is the result computed by a modular PRS any use at all? Yes! If the modular calculation indicated that two primitive polynomials are coprime then they really are. The only possible problem is when the modular calculation says there is a real non-trivial GCD, since then it could be that the polynomials are really coprime and the apparent GCD is an artefact of the approximate arithmetic, and anyway the modular GCD does not instantly allow us to recover the real one.

For multivariate polynomials it can be useful to reduce to the univariate case by substituting random (well, arbitrary, or pseudo-random or something) integer values for all bar one indeterminant. If this is done mod p it can not lead to over-large numbers. Then a modular PRS is computed for the resulting univariate polynomials. This provides a cheap and in fact rather reliable test that identified most cases where the full GCD will be 1.

4.4.2 The Hensel Construction

Given the above cheap check it would be very nice if the true GCD could be reconstructed from a modular image in non-trivial cases. For multi-variate polynomials the way of doing this is known as the Hensel Construction, and we will see it in action again when we come to factorisation. The presentation here is at first going to gloss over a few problem areas to try to give as clear an overview as possible. So imagine a bi-variate case where polynomials in x and y are being used. We are computing the gcd of $a(x, y)$ and $b(x, y)$. First view a and b as power series in y, eg.

$$a(x, y) = a_0(x) + a_1(x)y + a_2(x)y^2 + \ldots$$

Of course this will only be a finite series, stopping at the highest degree term in y present in a. Now suppose that the required gcd of a and b will be $g(x, y)$, and write $h = a/g$. This division of course goes exactly since g, being the gcd of a and b, must divide neatly into a.

The key step is then to observe the (obvious) identity $a = gh$ but to write it out in power series form. Looking at the various powers of y in this gives a series of identities (I have written just a_i instead of $a_i(x)$ etc. here to keep things concise, but everything mentioned will be dependent on x).

$$\begin{aligned} a_0 &= g_0h_0 \\ a_1 &= g_0h_1 + h_0g_1 \\ -(g_1h_1) + a_2 &= g_0h_2 + h_0g_2 \\ -(g_1h_2 + g_2g_1) + a_3 &= g_0h_3 + h_0g_3 \\ &\ldots \end{aligned}$$

I have collected terms on the two sides of the equals sign in a slightly curious way, but otherwise the equations are pretty straightforward. Next consider $\gcd(a_0, b_0)$. If we are *lucky*[6] this will be exactly g_0, the leading term in the power series representation of the required gcd. Then we can compute h_0 as a_0/g_0. The first of our equations has just been satisfied. Now suppose we can solve the next equation to find g_1 and h_1, then all the subsequent equations would be of the same form, ie.

$$g_0h_i + h_0g_i = \textit{<something>}$$

and thus it will be possible to compute as many of the coefficients g_i as are required. The largest one that can possibly be nonzero is the one corresponding to

[6]The term "lucky" is treated as a technical term here!

the highest power of y present in the gcd g, and this is bounded by the degree in y of a and b.

To show how to solve the odd equations we have come up with I need to impose two technical constraints. The first is that g_0 and h_0 must be co-prime. For now let me just observe that this can only possible fail to be so if the original polynomials a and b had repeated factors, and that it is possible to reduce the general problem one where no repeated factors are present. The second condition is that all the polynomials that we are working with have their coefficients reduced modulo a suitable[7] prime p. The reason for this is that when working mod p it is possible to divide any coefficient by another, while when working with integer coefficients some such divisions may not go exactly.

Now before showing how to solve the given equation for *polynomials* let's look at the analogous one for integers, and for given u and v try to find α and β such that $\alpha u + \beta v = 1$. This can be achieved by looking at a remainder sequence that computes the gcd of a and b. Start by writing $a = 1a+0b$ and $b = 0a+1b$, and note that if $q = \lfloor a/b \rfloor$ then remainder$(a, b) = a - qb$, or otherwise $1a + (-q)b$. Following through each value in the remainder sequence can be expressed as a combination of the two previous ones, and hence as a linear combination of the original values a and b. Eventually if $\gcd(a, b) = 1$ this will lead to the final 1 in the remainder sequence being expressible in the form $\alpha a + \beta b$ as required. By multiplying these values α and β by any value c that value c can be expressed as a linear combination of a and b. The same basic process can be used on coefficients (with modular coefficients), and hence completes the method for extending a univariate gcd to a bivariate one. It can clearly be used to cope with polynomials in any number of variables.

4.4.3 Hensel and integer coefficients

The Hensel construction as sketched above makes it possible to reconstruct a multivariate gcd, but only with coefficients reduced mod p. To get back to integer coefficients a little more is needed. In fact this is remarkably easy and similar to the ideas we have already seen. Observe that if we write integers in radix p an integer n can be written out as a combination of digits[8]

$$n = n_0 + n_1 p + n_2 p^2 + \ldots$$

[7] The rules about what counts as "suitable" are not especially complicated but are not discussed in depth here.

[8] Negative numbers might need a little more effort, but that is not a real difficulty and will be glossed over here.

where each digits is restricted to the range 0 to $p-1$. This looks sufficiently like a power series that the previous presentation of the Hensel construction can be applied to it. The methods even work cheerfully if instead of just numbers we work with polynomials written out as

$$a(x,y) = a_0(x,y) + a_1(x,y)p + a_2(x,y)p^2 + \ldots$$

where now the correct normalisation constraint is that all the coefficients in each $a_i(x,y)$ must be bounded by p. Use of the Hensel construction in this way makes it easy to extend a gcd that has been computed correctly modulo p to one that is correct modulo p^2, p^3, and so on until a value has been produced correct modulo some power of p that is larger than any possible integer coefficient in the true answer. At this stage the computed result must in fact be the true gcd.

4.4.4 Unluckiness

Sometimes the modular-number based gcd processes will fail because the prime used interacts badly with the particular polynomials used. For instance if one worked modulo 3 and while trying to compute $\gcd(x+1, x+4)$ something would be bound to go wrong because modulo 3 the two inputs alias together. A complete program has to cope with this by being prepared to accept that the Hensel extension process may fail to converge. In that case trying other primes and and other small adjustments[9] make it possible to proceed. Fortunately it can be shown in this case that almost all primes will be "lucky".

4.5 Factorisation and algebraic number arithmetic

Factorising polynomials follows on from gcd calculation as the next major algorithm-cluster. One first restricts attention to primitive polynomials. Then removes the frivolous complication of repeated factors.

4.5.1 Square-free decomposition

If a polynomial $u(x)$ has a repeated factor then $\gcd(u, \mathrm{d}u/\mathrm{d}x)$ will be non-trivial. By following up on this observation it is possible to do a number of gcd operations that will express u unambiguously in the form

$$u = K u_1 u_2^2 u_3^3 \ldots$$

[9] which will be mentioned briefly in the lectures.

where each u_i is now free of repeated factors.

4.5.2 Kroneker's algorithm

A classical way of finding factors of a polynomial can be illustrated by looking for possible quadratic factors of $u(x)$. Find all integer factors of each of $u(0)$, $u(1)$ and $u(2)$. For each possible combination of one factor from $u(0)$, one from $u(1)$ and one from $u(2)$ it is possible to interpolate a quadratic. If this quadratic has integer coefficients and divides exactly into u then we have found a factor as desired. If there is any quadratic factor we will come across it is this search.

This method can clearly be used to see factors of any desired degree, but in becomes grotesquely expensive for non-trivial cases and is not obviously adaptable to the needs of multivariate cases.

4.5.3 Berlekamp/Hensel

A better way to factorise polynomials will start by reducing an arbitrary input to a univariate polynomial modulo a prime p. If this can be factorised then the Hensel construction will make it possible to reconstruct the multivariate factors over the integers.

Berlekamp's algorithm factorises a univariate polynomial modulo some given prime. The inputs are:

A polynomial u(x), which should be primitive and square-free.

A prime p.

We will imagine that the (irreducible) factors of $u(x)$ are $w_1(x)$, $w_2(x)$,..., $w_j(x)$. All arithmetic mentioned here will be done modulo the selected prime p, and so all coefficients will be in the range 0 to $p-1$.

The factorisation process works by considering the class of polynomials $v(x)$ that have the properties:

degree$(v) <$ degree(u)

u divides exactly into $(v-0)(v-1)(v-2)...(v-[p-1])$.

It will turn out that these polynomials v can be found quite easily, and that from them it is possible to derive the factors w_i that will be our final results. I will demonstrate the first of these facts first.

Finding the v polynomials: Because we are working modulo a prime p the formula

$$(v-0)(v-1)...(v-[p-1])$$

can be simplified to $v(x)^p - v(x)$. A further result relating to polynomials with modular coefficients asserts that this in turn is equal to $v(x^p) - v(x)$. The justification of these two steps is not included in this note, but the results are standard mathematical ones. The condition that $u(x)$ divides $v(x^p) - v(x)$ can be expressed as

$$v(x^p) - v(x) = 0 \pmod{u(x)}.$$

A polynomial v can be characterised by a vector giving its coefficients v_0, v_1, ..., v_{n-1} (we know that the degree of v is to be less than that of u, and so can establish the number of coefficients to be considered). The equation we are solving now turns into

$$v_0(x^0 - x^0 \bmod u) + v_1(x^p - x^1 \bmod u) +$$
$$v_2(x^{2p} - x^2 \bmod u) + v_{n-1}(x^{(n-1)p} - x^{n-1} \bmod u) = 0$$

where all the polynomials involved now have degrees less than that of $u(x)$. Each of the polynomials $x^{kp} - x^k$ can be written out as a row of coefficients, and the rows (one for each value of k) stacked to form a matrix, generally known as Q. The equation that was to be solved now takes the form of a matrix multiplication:

$$(v_0, v_1, \ldots)Q = 0.$$

Solving such a set of equations is a standard piece of linear algebra. The complete set of solutions for v form a vector space and a set of independent vectors spanning this space is known as a null space basis for the matrix Q. To save space I will not fill in the full details here, but finding a null space basis proceeds by going through the steps of Gaussian elimination as if the matrix involved contained coefficients for a set of linear equations. For equations it is unusual and generally an error for the matrix to be found to be singular: for a null-space finding algorithm this is just what is expected and the null vectors amount to nothing more than a record of how and when singularity was found.

If a basic set of j independent null vectors are found then all solutions to our original equation can be expressed as linear combinations of these. Since we are working $\pmod{p}$ these linear combinations can only have numbers in the range 0 to $p-1$ as coefficients, and so a grand total of p^j solutions for $v(x)$ exist.

Deriving factors: If $u(x)$ divides exactly into $(v-0)(v-1)...(v-[p-1])$ then necessarily all the factors $w_i(x)$ also divide into it. But except for the dull case that v is a constant, $(v-0)$, $(v-1)$ and so on are all pairwise coprime, and so this means that each w_i must divide exactly into some particular one of the $(v-s)$. This leads to the observation that given some particular factor w_1 (say) and one of our v polynomials, there is a constant s such that:

w_1 divides exactly into $u(x)$ [of course it does!],

w_1 divides exactly into $(v(x)-s)$.

Since w was a proper factor of $u(x)$ its degree is greater than zero, and since it divides into $v(x)-s$ its degree is at most the same as that of v, which is in turn strictly less than that of u. So $u(x)$ and $v(x)-s$ have a w as a common factor. We compute g as their greatest common divisor. The polynomial g will be either the factor w_1 itself or some multiple of it, but since the degree of g is less than that of u the identity

$$u(x) = g(x)(u(x)/g(x))$$

has exhibited a non-trivial split of $u(x)$. There is no way of telling what value s will work in the above, but provided the prime p is fairly small it is quite cheap enough to try all the possible values 0, 1, ..., $[p-1]$ until one is found to lead to a factorisation of $u(x)$.

In fact the number of independent null vectors that are found is exactly equal to the number of irreducible factor that $u(x)$ will have, and by performing gcd operations between $u(x)$ and terms of the form $(v-s)$ letting v range over the null space basis and s over the numbers from 0 to $[p-1]$ it is always possible to complete the factorisation of $u(x)$.

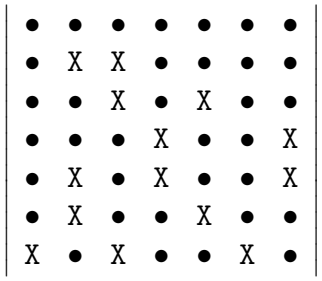
Example: Factorise $u(x) = x^7 + x^5 + x^3 + x^2 + 1$ modulo the prime 2.

Start by writing down

$$\begin{aligned}
x^0 - x^0 &= 0 \\
x^2 - x^1 &= x^2 + x \pmod{u(x)}, \pmod{2} \\
x^4 - x^2 &= x^4 + x^2 \\
x^6 - x^3 &= x^6 + x^3 \\
x^8 - x^4 &= x^6 + x^3 + x \\
x^{10} - x^5 &= x^4 + x \\
x^{12} - x^6 &= x^5 + x^2 + 1.
\end{aligned}$$

The right hand sides are just the remainders left when the left hand sides are divided by $u(x)$, and so in the first 4 cases where the degree of the left hand side is less than that of $u(x)$ all that has happened is that mod 2 arithmetic has allowed us to write $+$ instead of $-$.

This makes the Q matrix:



and I can exhibit two independent null vectors:

```
      [1, 0, 0, 0, 0, 0, 0]
and   [0, 1, 1, 0, 0, 1, 0].
```

You can check that if you multiply either of these row vectors by the matrix (mod 2) you get zero. Once again I am not going to go into the fine details of the (simple) calculation that leads to this null space basis - I will just go on from the fact that I have v-polynomials $v_1 = 1$ and $v_2 = x^5 + x^2 + x$ to work from. The fact that there are two independent null vectors indicates that my polynomial will have 2 factors. I now compute

$$\gcd(u, v_1 - 0), \gcd(u, v_1 - 1), \gcd(u, v_2 - 0), \gcd(u, v_2 - 1)$$

and expect one of these to be non-trivial. In this case it turns out that $\gcd(u, v2 - 0) = x^4 + x + 1$, and so this is one of the factors of u. The other is easily found (by dividing the known one into $u(x)$) to be $x^3 + x + 1$, and since we knew that $u(x)$ had just two factors we are finished.

Observations: For large primes p this algorithm becomes costly because of the need to try all possible values of s. It is therefore usual to use it with primes in the range (say) up to 100. The case $p = 2$ can be handled particularly neatly using bit-vectors to represent all of the polynomials and exclusive-OR and shift instructions for some of the operations on them. For large primes (e.g. about the size of a machine word) there are slightly more elaborate variants on the algorithm - see Knuth vol. 2 for some pointers. A coded version of the complete algorithm is no more than a couple of pages long.

4.5.4 Algebraic number arithmetic

The natural technical follow-on to gcd and factorisation algorithms involve *algebraic numbers*. These are values that arise as the roots of polynomial equations. Thus values such as $\sqrt{2}$ and $\sqrt[3]{5}$ are covered, as will be the imaginary i and the roots of any higher degree equation such as $\alpha^5 + \alpha + 1 = 0$.

Three main approaches are taken, and these correspond to different interpretations of what a symbol such as $\sqrt{2}$ really stands for:

1. **Root isolation:** Given a polynomial $u(x)$ set up a sequence of polynomials s_i with $s_0 = u, s_1 = u'$ and $s_{i+2} = -\text{remainder}(s_i, s_{i+1})$. Note that apart from the negative signs this is just a PRS. Now look at the curious function $S(z)$ which counts the number of sign changes in the sequence $s_0(z)$, $s_1(z)$, ... For any rational number z with $u(z) \neq 0$ it is clearly possible (if perhaps expensive) to compute the integer $S(z)$ quite unambiguously. Now, as I hope I will be able to make slightly plausible by drawing pictures on the blackboard, $|S(z_1) - S(z_2)|$ gives the number of real roots $u(x)$ has between z_1 and z_2! Given this tool (A Sturm Sequence, but note it is just another sort of PRS)) the roots of u can be counted and isolated to utterly arbitrary accuracy.

2. **Minimal Polynomial Arithmetic:** Represent any algebraic number by a symbol and a polynomial equation that it satisfies. Thus for $\sqrt{2}$ write $(\omega : \omega^2 - 2 = 0)$ and for $\sqrt[3]{5}$ use $(\eta : \eta^3 - 5 = 0)$. Note that this representation simultaneously allows for both the $\sqrt{2}$ that is about 1.414 and the one that is -1.414, ie. what is being represented is really $\pm\sqrt{2}$.

   To combine values that are represented this way is actually rather straightforward. Consider the addition of the the values mentioned above. Write out a set of equations:

$$\begin{aligned} \omega^2 - 2 &= 0 \\ \eta^3 - 5 &= 0 \\ \nu - (\omega + \eta) &= 0 \end{aligned}$$

   where the first two equations are the definitions of ω and η and the final one expresses the arithmetic operation being performed. Now just eliminate ω and η from these equations to leave one in just ν and a representation for the algebraic number sum has been derived as required. Hah! Eliminating

variables between polynomial equations was what resultants were about, and again we are really using a method based on the PRS.

3. **Reduction rewrites:** Set up rewrite rules like $(\sqrt{2})^2 \rightarrow 2$ and $i^2 \rightarrow -1$. Apply these throughout your calculations whenever possible. Hope or expect that this will lead to consistent and well-simplified results. A major problem that can arise with the use of rewrites is that sometimes some complicated rewrites can interfere with each other so that the order in which reductions are performed influences the final result produced. For polynomial rewrites it is possible to extend any initial set of rules into a so-called *Groebner base* which has the good property that whatever strategy is used for selecting rewrites to apply the same end result will always be obtained. Algorithms for the construction of Groebner bases have been a central part of research into Computer Algebra over the last about 15 years, and you should inspect the specialist literature if you want to know more about them or about more of their applications.

4.6 Exercises

4.6.1

Find an example that illustrates that when the sum $(a/b) + (c/d)$ is evaluated as $(a(d/g) + c(b/g))/(b(d/g))$ [where $g = \gcd(b, d)$] the final quotient can still need a common factor removed from the numerator and denominator.

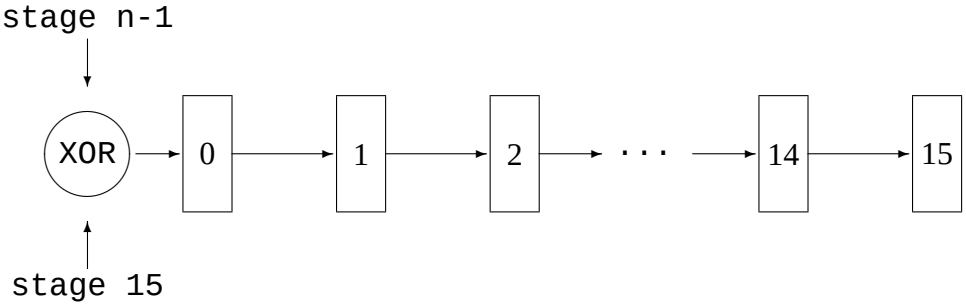
4.6.2

Show how to form the product and quotient of (a/b) and (c/d) using sequences of small gcd calculations rather than a single large one. Note that it is possible to exploit the fact that the incoming expressions should already be in their lowest terms and so gcd(a,b)=gcd(c,d)=1 can be assumed.

4.6.3

Use Berlekamp's algorithm to find a polynomial of the form $x^{16} + x^n + 1$ which is irreducible mod 2. Clearly n will be in the range 1 to 15, and an argument of symmetry shows that it is only necessary to consider n from 1 to 7. A value of n that gives an irreducible polynomial corresponds to the position of a tap in a feedback shift-register (as shown below) that will cycle with largest possible

period: such shift registers can be used to generate checksums or pseudo-random bitstreams.



Each stage in the shift-register is a D type flip-flop (all driven by a common clock signal), with the shift register's input being derived by forming the exclusive OR of the output from the final and some intermediate stage. Possible lengths of cycles in the bit patterns generated by this arrangement are related to the factor structure (mod 2) of a polynomial that has 1 coefficients at the places where taps are taken from the shift register. Textbooks on error correcting codes (e.g. Peterson, 19xx) provide further explanation and background: it was for the analysis of this sort of circuit that Berlekamp's algorithm was first introduced.

5 Transcendental Functions

5.1 Introduction

Consistent algebraic calculation with formula involving transcendental functions is difficult. In particular, for many of the classes of algebraic formulae that seem natural it is known to be impossible to produce a systematic way of determining if an expression is zero or not. If E is an expressions which might possibly be zero, but there again might not be, the simplification of formulae such as $AE+B$ can not proceed. Other questions become hard to answer: the expressions $Eexp(x^2) + x$ has an integral in closed (elementary) form if and only of $E = 0$, and so the inability to decide if an expression vanishes means that it is impossible to exhibit a reliable and complete integration algorithm.

A frustrating aspect of the main undecidability result for algebraic simplification is that key technical aspects of the proof suggest strongly that current understanding in the area is incomplete, and that the pessimism expressed above may not be entirely justified. To explain this, a sketch of a proof (due to Richardson)

that identifying real-valued functions that are in fact zero will be given. Many details of the proof will be omitted or glossed over, but the line of argument given can be made watertight. The purpose of showing this proof is so that a certain key oddity in it can be exposed and not with the intent that the proof as a whole be followed in detail.

Richardson's result shows that there is no algorithm that will reliably determine if a real function of a single real variable is in fact identically zero. The class of functions it allows are built up from the variable x, the constant π and the integers, using the four normal arithmetic operations, together with uses of the normal elementary functions: $\sin$, $\cos$, $\exp$, $\log$ and sqrt. It is supposed that any expression that can be shown to have a constant value can be tested to see if it is zero. If this were not true then deciding if expressions were zero would certainly be undecidable! Richardson restricts his analysis to real values. Since the $\log$ and sqrt functions are (in this context) undefined for negative arguments, a function from the class considered may be partial. The result proved will show that it is impossible to tell if the function is zero at such places as it has a value, thereby allowing for formulae such as $((x/x) - 1)$ which are certainly zero everywhere except at $x = 0$, but are undefined there.

The proof proceeds by assuming that there was an algorithm to decide if functions from the given class were zero, and shows how this would imply an algorithmic solution to a different problem that has already been shown to be algorithmically undecidable. The known undecidable problem is that of solving Diophantine equations: that is finding sets of integer values that result in given multivariate polynomials (with integer coefficients) vanishing. Of course given any particular Diophantine equation it may be possible to find the solutions: the thing which can not be done is to produce a single algorithmic method that guarantees to solve any such equation in a finite amount of time. (In this context solving also includes the case of deciding that an equation does not have any solutions in terms of whole numbers).

Richardson shows how, for any given Diophantine equation, it is possible to derive an analytic function of one variable such that that function being identically zero corresponds means that the Diophantine equation has no solutions. Thus if it were possible to test the function for zero it would be possible to tell that the Diophantine equation could not be solved, and that is known to be impossible.

The first step in the derivation of this result looks like the first cheat! Consider a polynomial equation P(x,y,z)=0 where we are only interested interested in solutions where x, y and z take integer values. Then for any such solution it is

certainly true that

$$P(x,y,z)^2 + \sin(\pi x)^2 + \sin(\pi y)^2 + \sin(\pi z)^2 = 0$$

and indeed a similar statement can be made however many variables were present in the original formula. Furthermore if we restrict our attention to real values the new function can only possibly be zero at places that are integer solutions to $P = 0$ even if x, y and z are now thought of as variables that can range over all possible real values.

The next step shows that from any multivariate function (for instance the one just written down) it is possible to derive a function of one variable which takes the value zero at places related to the zeros of the original function. The essence of this is the construction of packing functions p_1, p_2,... (details are not given here) and the replacement of

$$f(x,y,z)$$

by

$$f(t) = f(p_1(t), p_2(t), p_3(t))$$

where as t varies the functions p_1, p_2, ... arrange that all possible integer values of x, y and z are visited eventually. Despite the fact that this may seem peculiar, this sort of packing transformation is a standard tool in the analysis of what is computable and what is not.

The univariate function $f(t)$ is now multiplied by a large positive value so that any nonzero minima of $f(t)$ are scaled to have value greater than 1. Inventing a scale value (which will in fact be a function of t derived from the expression that represented $f(t)$) is not a particularly obvious step, but can be done. If the scaled univariate function is $F(t)$, then we now have $F(t) > 1$ except in the immediate vicinity of a place where $F(t) = 0$, and these places correspond to integer solutions of the Diophantine equation we started from.

Finally, Richardson shows how to switch from consideration of whether a function takes the value zero anywhere to a test on whether it takes a non-zero value anywhere. He observes that since we are working in real numbers, he can express a version of the absolute value function as, for instance $\text{abs}(x) = (\sqrt{x})^2$. If the use of a square root is objected to he can achieve the same effect using exponentials and logarithms. He then defines $h(x) = (\text{abs}(x-1) - (x-1))/2$ and a quick sketch of the graph of $h(x)$ shows that if its argument is less than 1 it is nonzero, while if $x > 1$ then $h(x)$ vanishes. This is enough to show that $h(F(t))$ can only have nonzero values if $F(t)$ takes values less than 1 somewhere,

and hence if $f(t)$ has roots. So any procedure that can find out if $h(F(t))$ is ever nonzero can be used to solve arbitrary Diophantine equations!

In the above proof the most objectionable step is the synthesis of the absolute value function as $(\sqrt{x})^2$, which amounts to the synthesis of a non-analytic function out of several analytic ones, but it has been found amazingly difficult to find any general way of prohibiting that while still allowing for calculations with an interestingly rich collection of functions. One obvious possibility would be to work with complex numbers rather than reals. Doing so reveals that the oddity relating to the absolute value function is just the tip of a huge iceberg of oddities and problems that arise when complex-valued functions can have multiple branches. For instance when working with algebraic formulae over complex numbers it is not obvious that $\sqrt{x} = \sqrt{x}$ or $\log(x) = 2i\pi + \log(x)$ until somebody has decided if the characters we use to stand for a function should represent one branch or all possible ones, or what!

5.2 Reducing the number of distinct functions

A good pragmatic approach to calculating with higher functions is to try to avoid it, or at least reduce the number of potentially interacting functions that are present in the formulae to be worked with.

5.3 Algebraic independence

If we have a formula that involves x, $\log(x)$ and $\sin(1/x)$ it might make sense to try replacing the two nasty functions by new symbols, y and z say. That will reduce the formula to just a rational function which we already know how to cope with. The only extra effort needed will be to remember that y and z need special treatment when differentiated, for example $\mathrm{d}y/\mathrm{dx} = 1/x$. The case above will in fact survive this re-interpretation, but if the original formula included various closely related functions (eg. $\exp(x)$ and $\exp(2x+1)$ or $\log(x)$ and $\log(5x^2)$, or even just $\sqrt{x^2}$) the rational function version of the formula would fail to capture important aspects of the original.

5.4 Structure Theorems and their uses

A *Structure Theorem* is a procedure for testing if a collection of exponentials and logarithms (or whatever) are sufficiently independent that renaming them into rational function variables is safe. The lectures will sketch the process applied in

either the exponential or logarithmic case. If the functions present in an original formula do not pass the test of a structure theorem the technology provides no guidance about what can or should be done — the idea is just to certify various good cases where reliable calculations can be performed and disallow the sorts of things that allowed Richardson's pessimistic result to be proved.

5.4.1 Risch's Integration algorithm

The main technical triumph built on top of the idea of structure theorems is algorithmic indefinite integration. The idea behind this is that in well behaved cases it is possible to predict the form that an integral will have (if one exists). A general expression (with a load of undetermined coefficients in it) of that form is then written down and identified with the original integral. After differentiating both sides the use of a structure theorem makes it valid to consider everything as rational function calculation. In particular it is permissible to compare coefficients, and doing so leads to a bunch of linear equations that can be solved to find values for the unknown coefficients that had been introduced. If the linear equations can be solved an integral has been found, otherwise the lack of a solution amounts to a proof that no integral in closed form exists. There are special (and extreme) complications that arise when integrands start off with square or higher roots, or when higher transcendental functions (error functions etc) are to be handled, and not all combinations can be coped with, but the vast majority of the integrals required in engineering applications can now be done by computer much faster and more reliably than either hand work or inspection of books of tables allows.

6 REDUCE keywords and switches

Note that Reduce[10] is an evolving system — new releases come out every couple of years and these always add a number of new operators and capabilities. In some cases awkward old syntax is replaced by a neater way of expressing things. The list will therefor not always contain a complete list of the functions that are available, but it does include the ones most important for any examples you might want to try for this course.

ABS Computes the absolute value of an expression

[10] And of course all the other major algebra systems.

ACOS	Arc-cosine
ACOSH	Arc-cosh
ADD	Alternative for +
ALGEBRAIC	Used when switching between algebraic and symbolic modes.
ALLBRANCH	Switch. Used with SOLVE. (on)
ALLFAC	Switch. If on, expressions are displayed with common factors first. (on)
AND	Logical operator
ANTISYMMETRIC	Declares operators to be antisymmetric in their arguments.
ARGLENGTH	Number of arguments of top level operator in expression.
ARRAY	For declaring arrays.
ASIN	Arc-sine
ASINH	Arc-sinh
ATAN	Arc-tan
ATANH	Arc-tanh
BEGIN END	Compound statement.
BYE	Finishes Reduce job, clears it from memory.
CARDNO!*	Fortran output option.
CLEAR	For removing assignments and substitutions.
COEFF	partitions polynomial expression into coefficients
COMMENT	Text between COMMENT and ; or $ is ignored.
COMP	Switch. Used to invoke Lisp compiler. (off)
CONS	Alternative for dot operator (more usual to use .).
CONT	Used to continue file input which has been PAUSEd.
CONVERT	Switch. If on, integral real coefficients are replaced by integers (on)
COS	Cosine
COSH	Hyperbolic cosine
COT	Cotangent
CREF	Switch. If on, does a cross-reference analysis. (off)
DEFINE	To define synonyms for Reduce keywords and indentifiers.
DEFN	Used in symbolic mode.
DEG	Leading degree of polynomial in given variable.
DEMO	Switch. If on, press return to execute next command from file input. (off)
DEN	Denominator of a rational expression.
DEPEND	Sets up dependencies between variables/kernels.
DET	Determinant of matrix.

DF	Partial differentiation of expression.
DIFFERENCE	Take difference between two arguments (more usual to use -).
DILOG	dilogarithm.
DISPLAY	For displaying previous inputs.
DIV	Switch. If on, displays have rational fractions, negative powers. (off)
DO	Used in FOR loops and with WHILE
E	The base of natural logarithms (2.71828...).
ECHO	Switch. If on, file input is echoed in display. (on)
EDITDEF	Allows interactive editing of user defined procedure.
ED	Allows interactive editing of any previous command.
END	Terminates a BEGIN END block or a file.
EPS	High energy physics: antisymmetric tensor of order 4.
EQ	Used in symbolic mode
EQUAL	Alternative to =
ERF	Error function.
EXP	Exponential function.
EXP	Switch. If on, expressions are expanded during evaluation. (on)
EXPINT	exponential integral.
EXPR	Used in symbolic mode.
EXPT	Alternative for ** or ^ (raising to a power).
EZGCD	Switch. If on, with GCD on, uses EZ-GCD algorithm to compute gcds. (off)
FACTOR	Switch. If on, expressions are displayed in factored form. (off)
FACTOR	Declares expressions as factors for displays.
FACTORIZE	Factorizes polynomial expression.
FAILHARD	Switch. If on, impossible integration returns error. (off)
FIXP	Returns true if expression is integer, else false.
FLOAT	Switch. If on, allows use of floating point numbers. (off)
FOR	Start of program loop.
FORALL LET	Declares new substitution rule(s).
FOREACH	Used in symbolic mode.
FORT	Switch. If on, display is in a Fortran notation. (off)
FORTWIDTH!*	Fortran output option.
FREEOF	True if first argument does not contain second argument.
G	High energy physics: a Dirac gamma matrix expression.
GCD	Switch. If on, greatest common divisors are cancelled. (off)
GCD	Returns the greatest common divisor of two polynomials.

GEQ	Alternative for >=
GO (TO)	For use with a labelled statement within BEGIN END.
GREATERP	Alternative for >
HIPOW!*	Set to highest non-zero power when COEFF is used.
I	Square root (-1).
IF THEN ELSE	Conditional statement.
IN	Takes input from external Reduce file(s).
INDEX	High energy physics
INFIX	Declares new infix operators.
INPUT	Used to reference previous inputs in new computations.
INT	Switch. Controls whether file input is batch or not. (default depends on implementation)
INT	Integration
INTEGER	Declares local integer variables in BEGIN END block.
KORDER	Declares internal ordering for variables.
LAMBDA	Used in symbolic mode.
LCM	Switch. If on, least common multiple of denominators is used. (on)
LCOF	Leading coefficient of polynomial.
LEQ	Alternative for <=
LESSP	Alternative for <
LET	Declares substitutions.
LINEAR	Declares operators to be linear in their arguments.
LINELENGTH	For setting output linelength - see Manual.
LISP	Synonym for SYMBOLIC.
LIST	Switch. If on expressions are displayed one term to a line. (off)
LOAD	Used in symbolic mode.
LOG	Natural logarithm
LOWPOW!*	Set to lowest non-zero power when COEFF is used.
LTERM	Leading term of expression.
MACRO	Used in symbolic mode.
MAINVAR	Main variable of polynomial.
MASS	High energy physics: assign masses to vectors.
MAT	Used to assign values to matrices.
MATCH	Declares substitutions (less flexible than LET)
MATRIX	Declares matrix variables.
MAX	Returns maximum of any number of numerical expressions.
MCD	Switch. If on, makes common denominators when adding

	expressions. (on)
MEMBER	Used in symbolic mode.
MEMQ	Used in symbolic mode.
MIN	Returns minimum of any number of numerical expressions.
MINUS	Alternative for -
MODULAR	Switch. If on, does arithmetic modulo SETMOD. (off)
MSG	Switch. If off, warning messages are not displayed. (on)
MSHELL	High energy physics: puts variables “on the mass shell”.
MULT	Alternative for *
NAT	Switch. If off, display is in form that could be used for input. (on)
NEQ	Not equal to.
NERO	Switch. If on, zero assignments are not displayed. (off)
NIL	Synonym for zero.
NODEPEND	Removes dependencies created by DEPEND.
NOLNR	Switch. Integration: may be useful if no closed form solution. (off)
NONCOM	Declares operators to be non-commutative for multiplication.
NOSPUR	High energy physics: (traces and Dirac matrix calculations).
NOT	Logical operator.
NUM	Numerator of a rational expression.
NUMBERP	True if argument is a number, else false.
NUMVAL	Switch. If on, expressions are evaluated numerically. (off)
OFF	Turns off the named mode switches.
ON	Turns on the named mode switches.
OPERATOR	Declares new prefix operators.
OR	Logical operator.
ORDER	Declares an ordering for variables in displays.
ORDP	True if first argument is ordered ahead of second argument.
OUT	Directs output to named file or to terminal (OUT T;).
OUTPUT	Switch. If off, there is no printing at the top level. (on)
OVERVIEW	Switch. Factorization: connected with TRFAC. (off)
PART	Extracting parts of expressions
PAUSE	In file input, offers option of continuing from terminal.
PERIOD	Switch. Fortran output option (re inclusion of decimal points). (on)
PFACTORIZE	Factorizes univariate polynomial, modulo given prime.
PGWD	Switch. Used in symbolic mode. (off)

PI	Circular constant.
PLAP	Switch. Used in symbolic mode. (off)
PLUS	Alternative for +
PRECEDENCE	Sets precedence of new operators declared by INFIX.
PRECISION	Sets precision for real arithmetic, used with ON ROUNDED.
PRET	Used in symbolic mode. (off)
PRI	Switch. If off, all output declarations and switches are ignored. (on)
PROCEDURE	Names statement(s).
PRODUCT	Used with FOR to find products.
PUT	To define synonyms for ALGEBRAIC or SYMBOLIC.
PWRDS	Switch. Used in symbolic mode. (on)
QUIT	Exit from Reduce.
QUOTIENT	Take ratio of two arguments (more usual to use /).
RAISE	Switch. If on case of letters is ignored in keywords, expressions. (on)
RAT	Switch. Used with FACTOR for displaying expressions. (off)
RATIONAL	Switch. If on, polynomials use rational numbers. (off)
REAL	Declares local real variables in BEGIN END block.
RECIP	Take reciprocal of argument (more usual to use 1/argument).
REDERR	Print error message.
REDUCT	Reductum of expression with respect to variable.
REMAINDER	Remainder when first polynomial is divided by second.
REMFAC	Clears the effect of FACTOR.
REMIND	High energy physics: removes the effect of INDEX.
REPEAT UNTIL	Provides repetition
RESUBS	Switch. If off, no resubstitutions are made after the first. (on)
RESULTANT	Resultant of two polynomials with respect to given variable.
RETRY	Tries to do the command in which the last error occurred.
RETURN	For transfer out of BEGIN END.
ROUNDED	Switch. If on, gives real number evaluation (see PRECISION). (off)
SAVEAS	Alternative for x:=ws$
SAVESTRUCTR	Switch. If on, causes STRUCTR to store results. (off)
SCALAR	Declares local variables in BEGIN END block.
SETQ	Alternative to :=
SETMOD	Sets modular base, used with mode switch MODULAR.

SHARE	Used in symbolic mode.
SHOWTIME	Displays the elapsed time since last SHOWTIME.
SHUT	Closes output file(s).
SIN	Sine
SINH	Hyperbolic sine
SMACRO	Used in symbolic mode.
SOLVE	Solves one or more simultaneous equations.
SOLVEINTERVAL	Switch. If on, inexact roots are represented by intervals. (off)
SOLVESINGULAR	Switch. If on, solutions may include arbitrary constants. (on)
SOLVEWRITE	Switch. If on, solutions are displayed. (on)
SPUR	High energy physics: traces in Dirac matrix calculations.
SQRT	square root
STEP UNTIL	Used in FOR loops
STRUCTR	Displays the structure of an expression.
SUB	Replace variable by expression in an expression.
SUCH THAT	Used in FORALL LET
SUM	Used with FOR to find sums
SYMBOLIC	Used when switching between algebraic and symbolic modes.
SYMMETRIC	Declares operators to be symmetric in their arguments.
T	Cannot be formal parameter or local variable in procedure.
TAN	Tangent
TANH	Hyperbolic tangent
TERMS	Number of top level terms in numerator of argument.
TIME	Switch. If on, cpu time used by each command is displayed. (off)
TIMES	Alternative for *
TIMINGS	Factorization: connected with TRFAC. (off)
TP	Transposes a matrix.
TRACE	Trace of a matrix.
TRFAC	Switch. Factorization: if on, traces operation of the algorithm. (off)
TRINT	Switch. Integration: if on, traces operation of the algorithm. (off)
UNTIL	Used with FOR and WHILE
VARNAME	Fortran output option, for naming expressions.
VECDIM	High energy physics: setting dimensions

VECTOR	High energy physics: declaring vectors.
WEIGHT	Asymptotic commnad for assigning weights.
WHILE DO	Provides repetition
WRITE	Displays expressions, strings.
WS	Used to reference previous outputs for new computations.
WTLEVEL	Asymptotic command to reset weight level (default 2).
!*MODE	Displays the current mode (algebraic or symbolic).
;	Terminator for a command, result is displayed.
$	Terminator for a command, result is not displayed.
:=	Assignment symbol
.	Dot operator, and decimal point.
<< >>	Indicates a group statement.
( )	Simple brackets.
:	May replace STEP 1 UNTIL in FOR loops, also use with labels.
,	Used as separator in lists.
" "	Delimits text in WRITE statement.
%	Text between % and end of line is ignored.
!	Is an escape character for special symbols in an identifier.
'	Used in symbolic mode.
{ }	Used to write lists.

The following arithmetic operators have their usual meaning on algebraic expressions (** and ^ are synonyms):

```
+   -   *   /   **   ^
```

The following relational operators have their usual meaning for comparing numbers:

```
=   >=   >   <=   <
```