Heuristic Algorithms

Gaston H. Gonnet
Institute for Scientific Computing
ETH Zurich, 8092 Zuerich, Switzerland
phone: +411 632 7470
e-mail: gonnet@inf.ethz.ch

Chapter 1

A Heuristic polynomial gcd algorithm

This is an example of the general scheme of heuristics. In this case we use a many-to-one mapping from polynomial variables onto the integers. This is an evaluation mapping. Under certain conditions satisfied by the size of the coefficients in the result and the evaluation value, we can invert this mapping.

We will assume the following:

- Computing multivariate polynomial gcds is difficult.
- Computing integer gcds is easy (even if the integers are thousands of digits long).
- Exact polynomial division is easy.

Let $a(x)$ and $b(x)$ be the polynomials. We can assume without loss of generality that these polynomials are over the integers and primitive (the content, that is the integer gcd of the coefficients, has been divided out). Each polynomial can be decomposed in two parts: the gcd and its cofactor.

$$g(x) = \mathrm{gcd}(a(x), b(x))$$

$$a(x) = g(x)c_a(x)$$
$$b(x) = g(x)c_b(x)$$

where $c_a(x)$ and $c_b(x)$ are relatively prime polynomials. We now choose our evaluation point n to be an integer sufficiently large (as will be defined later) and compute $a(n)$ and $b(n)$. Let

$$\mathrm{igcd}(a(n), b(n)) = g(n)r(n)$$

where $r(n)$ is an unknown integer computed as the integer gcd of $c_a(n)$ and $c_b(n)$. For our purposes $r(n)$ is a spurious factor. If it becomes too large in magnitude, it may prevent correct lifting of the answer.

We may have two problems:

- from $g(n)$ we may not reconstruct $g(x)$

- from $r(n)g(n)$ we may not reconstruct $r(n)g(x)$

Let $g^*(x)$ be the polynomial reconstructed from $r(n)g(n)$. This reconstruction is done from the lowest degree coefficient to the highest by symmetric mod n operations. The loop for this reconstruction is as follows:

```
# Algorithm to lift a polynomial from an integer
#  i.e. reverse the evaluation of g(x) at x=n

g := igcd( a(n), b(n) );
gstar := 0;
for i from 0 while g <> 0 do
    c := mods(g,n);
    gstar := gstar + c*x^i;
    g := (g-c)/n
    od;
```

In Maple this lifting operation is supported by the kernel and called `genpoly`. The above code would look like

```
gstar := genpoly( igcd( a(n), b(n) ), n, x );
```

It is obvious from the lifting, that all the coefficients lifted are $|c_i| \leq n/2$. So, if n is chosen to be at least twice as large as the largest coefficient in $g(x)r(n)$, then the lifting will always be successful.

But this has two problems. First we can only estimate the coefficients of $g(x)$ and have no idea of the magnitude of $r(n)$. Secondly, we would like to choose n to be as small as possible, so that the integer gcd computations are efficient. This second argument becomes very important for the multivariate case when we have to recursively evaluate for each variable; then the successive n_i will become doubly exponentially large.

So instead of guaranteeing that we choose a safe (large enough) n, we will choose a minimal one and test to make sure that the results are correct. The verification will be done as follows:

Once that we have obtained $g^*(x)$ we perform two trial divisions: we test whether $g^*(x)$ divides $a(x)$ and whether it divides $b(x)$. If this test fails, then the lifting was incorrect due to one of the above reasons. If this test passes, then $g^*(x)$ is an integer multiple of $g(x)$, i.e. $r(n)g(x)$ and by removing its content we obtain $g(x)$.

```
# algorithm GCDHEU

n := initial_n;
while true do
    gstar := genpoly( igcd( a(n), b(n) ), n, x );
    if divide(a(x),gstar) and
       divide(b(x),gstar) then break
    else n := 2*n+1 fi
    od;
g := gstar / icontent(gstar);
```

Proof that the primitive part of $g^*(x)$ is the correct gcd.

By passing the two division tests, we know that $g^*(x)$ is a factor of the gcd (it divides both polynomials). We will now prove that the degree of $g^*(x)$ is not less than the degree of $g(x)$ and hence it must be equal.

So we must find a large enough n such that $g(n)$ will lift a polynomial with the correct degree (or larger). Let

$$g(x) = g_n(x-\alpha_1)(x-\alpha_2)...$$

and

$$r(n)g(n) = r(n)g_n(n-\alpha_1)(n-\alpha_2)...$$

To guarantee that $g^*(x)$ will have the correct degree, this value should be larger than $\frac{n^{d_g}}{2}$ where d_g is the degree of $g(x)$, or

$$r(n)g_n(1-\alpha_1/n)(1-\alpha_2/n)... > 1/2$$

Assuming the worst case, i.e. $r(n)=1$, $g_n=1$ and letting $\alpha = \max_i |\alpha_i|$ we obtain

$$(1-\alpha/n)^{d_g} > 1/2$$

or

$$n > \frac{\alpha}{1-2^{-1/d_g}} > \frac{d_g \alpha}{\log 2}$$

For such n, the algorithm `GCDHEU` produces the correct gcd of $a(x)$ and $b(x)$. Notice that this is a heuristic because the loop may never terminate, but if it terminates, the result is correct. This type of heuristic is commonly called “Las Vegas”.

The roots of $g(x)$ are not known, but these should be roots of both $a(x)$ and $b(x)$. Then if n satisfies

$$n > \frac{\min(deg(a(x)), deg(b(x))) \min(\max_{a(\alpha_i)=0} |\alpha_i|, \max_{b(\alpha_i)=0} |\alpha_i|)}{\log 2}$$

How large is $r(n)$?

It is easy to see that there are polynomials which for which every evaluation over an integer give values which are divisible by $d!$ This is called the *fixed divisor* of a polynomial. E.g.

$$p(x) = x(x-1)(x-2)(x-3)...(x-d+1)$$

is always divisible by $d!$. In particular, $p(x)$ and $p(x+d)$ are relatively prime polynomials and $\mathrm{igcd}(p(n), p(n+d)) = d!$ for every value of n. So $r(n)$ could be rather large (in these cases it is composed of small prime factors).

It is useful to bound $r(n)$. The fixed divisors can be bounded using the original polynomials $a(x)$ and $b(x)$. As a matter of fact it is very easy to compute the fixed divisors:

```
# Compute the fixed divisor of a(x)
fd := a(0);
for i while fd > i! do fd := igcd( fd, a(i) ) od;
```

or to compute the common fixed divisor of both $a(x)$ and $b(x)$ (a multiple of $r(n)$).

```
# Compute the fixed divisor of g(x)
fd := igcd( a(0), b(0) );
for i while fd > i! do fd := igcd( fd, a(i), b(i) ) od;
```

Other common factors may arise with high probability. For example the polynomial $p(x) = x(x-1)(x-3)(x-4)+5$ does not have a fixed divisor. $p(x)$, $p(x+10)$ are relatively prime, yet for random integers 80% of the time their $r(n)$ is a multiple of 5.

1.1 Further improvements

Once that we have computed the minimum value of n and the first value of $r(n)g(n)$, it is possible to bound the degree of $g(x)$.

From the equation bounding $r(n)g(n)$ we can derive

$$g(n) = g_n(n-\alpha_1)... > g_n(n-\alpha)^{d_g} > \frac{g_n n^{d_g}}{2}$$

or

$$d_g < \log_n \frac{2g(n)}{g_n} \leq \log_n 2r(n)g(n)$$

This bound is rather tight. The first use of this bound is to estimate n itself, in case that the first iteration did not lift a correct gcd.

The second use of the upper bound on d_g is more interesting. So far we have tried to lift $g(x)$. But it is equally possible to lift one of the cofactors, that is $c_a(x)$ or $c_b(x)$. By knowing the degrees of $a(x)$, $b(x)$ and an estimate of the degree of $g(x)$ we can decide to lift the polynomial with the lowest degree (which in practice will the least expensive to lift).

```
# Cofactor lifting

n := any_choice;
an := a(n);
ca := genpoly( an / igcd( an, b(n) ), n, x );
if divide( a(x), ca, 'g' ) and
    divide( b, g ) then # success
else g := FAIL fi;
```

The cofactor lifting has a significant advantage on the previous lifting. The above procedure, if successful, produces the correct result for any value of n. So we can choose n as small as we wish. The proof of correctness is based on the fact that a poor choice of n or $r(n)$ may lift a $c_a(x)$ of lower degree, which means that $g(x)$ will be of higher degree. Hence this will never pass the test.

If $r(n)$ is not 1, then the above procedure will not work, i.e. it will always fail. This can be improved by finding the common fixed divisor of both polynomials as shown above. It is also possible to allow for some extra spurious factors in $r(n)$ and multiply $a(n)$ by small integers.

The combination of lifting the gcd or one of its cofactors is quite complementary. It happens for unbalanced problems that either the gcd is large (large degree and large coefficients) or the cofactor is large. The lifting of non-large polynomials is more efficient (will not need to increase n).

The main paper on this subject is [1]. The original proofs and the actual algorithm implemented in Maple differ from the one presented here.

Chapter 2

Heuristic equivalence testing and Signature Functions

The problem of equivalence testing is determining whether two expressions (different in appearance) are indeed mathematically the same. This is equivalent to testing whether an expression is identically zero. For example, is

$$x^{10} - 1 + (1 - x^5)(x^5 + 1) = 0?$$

As with other heuristic algorithms, we solve this problem over a simpler domain. The answer to our problem is just `true` or `false`, then it is not necessary (or possible) to do any lifting. In other words, there is too little information in this binary result to be able to lift it from the simplified domain to the original domain. The absence of lifting makes this type of heuristic very appealing. As we will see later, every time that we obtain the value `false` it is correct, and when we obtain the value `true`, it could be incorrect (and we should `FAIL`). Instead of failing, and hence failing with all the results that are identical, we will compute the probability of failing for random parameters and produce a probabilistic result. Since this probability can be tuned, we can make it small enough for any practical purposes.

From now on we will concentrate exclusively on testing whether an expression is zero. To test whether $a = b$, we test $a - b = 0$.

2.1 Modular mappings

For this heuristic we will also work with an evaluation mapping, this time over the integers mod n. This makes the computation time proportional to the expression size. Unknowns are assigned arbitrary random values in Z_n. As a result, every expression is mapped onto an integer. If the expression is identically 0, any evaluation for its variables and for any n, it will evaluate to 0. So a non-zero result is proof that the expression is not identically 0. It is important that we insist on computing in time proportional to the expression size, as it is represented, and not with other measures, like degree of the polynomial, expanded size, etc. These other measures could be exponential in the size of the expression. For example consider the polynomial

$$(((...(x+1)+2)^2+3)^2+4)^2...+k)^2$$

This is a dense polynomial with degree 2^k but the expression is only of size $O(k)$.

The mapping of a mathematical expression to an integer mod n will be called *signature*, and denoted by $S_n(e)$. A signature is a many-to-one function. A signature function $S_n(x)$ must distribute over arithmetic operations, that is:

$$S_n(a \pm b) = S_n(a) \pm S_n(b) \pmod n$$
$$S_n(ab) = S_n(a)S_n(b) \pmod n$$
$$S_n(a/b) = S_n(a)/S_n(b) \pmod n$$

This heuristic has an obvious parameter, which is n which can be set to any arbitrary value. We are going to use this feature quite extensively for two purposes. First we will select n so that all the computations are possible (or are simpler). Secondly, we will select different values of n, so that we obtain many independent results. By running this heuristic for different values of n we can exponentially reduce the probability of making a mistake.

For example, suppose that we want to test whether $x^{10} - 1 + (1 - x^5)(x^5 + 1)$ is identically 0. We will use $n = 7$ and we will map our only unknown to a random value, e.g. $S(x) = 2$. The expression becomes $2^{10} - 1 + (1 - 2^5)(2^5 + 1) \pmod 7$. Notice that the exponentiations are done mod 7 and using binary powering and hence do not require time or size proportional to the exponent, but to the logarithm of the exponent. Since $2^5 = 4 \pmod 7$ and $2^{10} = 2 \pmod 7$ the above becomes $2 - 1 + (1 - 4)(1 + 4) \pmod 7 = 0$. The heuristic answer is `true`.

Obviously this procedure can make mistakes. $x - 2$ will also map to 0 under the same mapping. For polynomial expressions, it is quite easy to derive a bound on the number of mistakes the heuristic will make. If $p(k) = 0 \pmod n$, then $(x - k)$ is a factor of $p(x) \pmod n$. The factored polynomial can have at most d different linear factors, where $d = \deg_x p(x)$. There are at most d values out of n where we can give the wrong answer. Hence for a polynomial we can bound the probability of a mistake by d/n.

Note: Computing the degree of an expression may be as difficult as zero equivalence. It is enough to have an upper bound on the degree. An easy and tight bound on the degree is 2^q, where q is the number of multiplications (or divisions) used to compute the polynomial. The bound is tight since the polynomial $x^{2^q} - 1$ can be computed with q multiplications and has 2^q distinct factors for any prime of the form $n = k2^q + 1$. E.g.

$$x^4 - 1 = (x+4)(x+1)(x+13)(x+16) \pmod{17}$$

In simple terms, for every multiplication/division that we have in the expression we have to multiply n by 2 (add one bit to its length). This bounds the length of n to the size of the expression being tested.

Remember that this heuristic makes mistakes only if $S_n(e) = 0$. If $S_n(e) \neq 0$, then $e \neq 0$.

Our approach for computing (better) heuristics will be to select n from some properties or features of the expression. In this way we can make n large enough to satisfy our probability constraints and satisfy some other constraints that will be seen later.

2.1.1 Rational expressions

If a division by 0 occurs, i.e. $S_n(p/q) = S_n(p)/S_n(q) \bmod n$ and $S_n(q) = 0$, then we try to prove (recursively) that q is identically zero. If it is true, then the original expression does not make mathematical sense. If we have proof that the denominator is not zero, this means that we have found an n for which $S_n(q) \neq 0$, and hence can continue computing the signature of the original expression with this n. If the division fails because $S_n(q)|n$, when n is not prime, then we can try with different values of n. In some cases, when there are other restrictions on n, it may not be possible to compute the division. For example, if n is restricted to be a multiple of 2, then the expression $S_n(\frac{1}{x(x+1)})$ cannot be computed.

2.1.2 Field/congruence enlargement

It is always possible to compute signatures in a larger congruence i.e. $S_n(e) = S_{mn}(e) \pmod n$ for integer m. This is not advantageous in general except for the computation of some inverse functions as will be seen later. In general the computation mod mn will be more expensive than mod n.

2.1.3 Signature of unknowns

Signatures of unknowns should be random, and should not satisfy any simple relation between each other. We must beware of random number generators, which may give results which are mathematically related. Choosing the signatures of the i^{th} unknown as $S_n(x_i) = ag^i \bmod n$ is not very good, the expression $x_0x_3/(x_1x_2) - 1$ will have signature 0 for any value of a, g and n.

Polynomials are also very poor generators of signatures. For example, suppose that we use a polynomial $p(x)$ to generate the signatures of unknowns, by using $S_n(x_i) = p(i)$. It happens that

$$p(x+1) + p(x+2) + p(x+6) = p(x) + p(x+4) + p(x+5)$$

for *any* polynomial of degree 2. So $S_n(x_1 + x_2 + x_6 - (x_0 + x_4 + x_5)) = 0$ for any arbitrary quadratic polynomial and any value of n. This is highly undesirable.

A good generator for this purpose is algorithm A (Additive number generator) described in Knuth, The Art of Computer Programming, vol 2, second edition, page 27. Another good method is to use a double-mod random number generator, i.e.

$$S_n(x_i) = (ag^i \pmod N) \pmod n$$

where $N \gg n$ and $(N, n) = 1$.

Finding simple relations among consecutive polynomial values

The above relationship,

$$p(x+1) + p(x+2) + p(x+6) = p(x) + p(x+4) + p(x+5)$$

is a relation that will hold for any polynomial of degree 2. It is interesting as a side topic to be able to generate this kind of simple relationships, not just for coefficients 1 and -1, but for small coefficients.

This can be resolved with signatures of expressions and the LLL lattice reduction algorithm, and it is a very good example of both. The problem can be formalized as follows: we are looking for a polynomial $p(x) = a_d x^d + ... a_1 x + a_0$ such that $\alpha_0 p(x) + \alpha_1 p(x+1) + ... + \alpha_m p(x+m) = 0$ for any coefficients in $p(x)$ and for any value of x. Furthermore, we are interested in the case where the α_i coefficients are small.

For this we will construct the $(m+2) \times (m+2)$ matrix

$$\begin{array}{ccccc} 1 & 0 & \dots & 0 & bS_n(p(x)) \\ 0 & 1 & \dots & 0 & bS_n(p(x+1)) \\ \dots & & & & \\ 0 & 0 & \dots & 1 & bS_n(p(x+m)) \\ 0 & 0 & \dots & 0 & bn \end{array}$$

where b is an integer *blow up* factor, used to make sure that for the shortest vectors, a 0 will be in that column, and hence the corresponding relations with the polynomials are exact. The last column contains all the signatures of the unknown polynomial, which are computed (mod n). The last row serves the purpose of computing the linear combinations of the signatures (mod n).

For example, if we let $d = 2$, $n = 100000007$, $m = 6$ and $b = 10$ and the random variables assigned to the unknowns: $a_0 = 19639163$, $a_1 = 10670793$; $a_2 = 33049645$ and $x = 56110369$ then the matrix is

$$\begin{array}{cccccccr} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 574292850 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 940678620 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 968057220 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 656428650 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 5792910 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 16150070 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 687500130 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1000000070 \end{array}$$

Applying the LLL algorithm to the above produces the reduced matrix:

$$\begin{array}{rrrrrrrr} -1 & 2 & 0 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 0 & 2 & -1 & 0 & 0 \\ -1 & 1 & 1 & 0 & -1 & -1 & 1 & 0 \\ -1 & 2 & -1 & 1 & -2 & 1 & 0 & 0 \\ -8 & -5 & -2 & 0 & 0 & -1 & -4 & 50 \\ 8 & 0 & -8 & -17 & -22 & -31 & -35 & -20 \\ 63 & 20 & -10 & -21 & -14 & 11 & 50 & 10 \\ 43 & -36 & -29 & -24 & -16 & -9 & -2 & -20 \end{array}$$

From these short vectors we can conclude the previous relation from row 3 and the relation

$$2p(x+1) + p(x+4) = p(x) + 2p(x+3)$$

from row 1. These relations can be verified with a computer algebra system if necessary.

2.1.4 Rational numbers

No problems provided that the denominator and n are relatively prime. Since we can choose n, we will select it to be relatively prime to all the denominators in the expression.

2.1.5 Real numbers

Floating point numbers (real numbers in computer jargon) are a special form of rational numbers. The denominators of these rational numbers are powers of the base (2 or 10 for

IEEE standard representations). Provided that n is relatively prime to the base, we can always compute their signatures. Notice however that there is no notion of approximation, i.e. we cannot conclude anything about $|S_n(a) - S_n(b)|$ from $0 < |a - b| < \epsilon$.

2.1.6 Arbitrary powers

The signature of an arbitrary power can be computed as

$$S_n(a^b) = S_n(a)^{S_{\phi(n)}(b)}$$

where $\phi(n)$ is Euler's totient function or the number of divisors of n. This is necessary to guarantee that the relation $S_n(a^{b+c}) = S_n(a^b)S_n(a^c)$. It is a consequence of Fermat's little result, that $a^{\phi(n)} = 1 \pmod{n}$. The signatures of exponents will be computed mod $\phi(n)$, a value which may have some undesirable properties. E.g. $2|\phi(n)$ for any n, and hence $S_n(a^{\frac{1}{x(x+1)}})$ cannot be computed.

For this reason it is important to find out if $S_n(a)$ is a primitive root or not. This can be done with an efficient test. For every prime factor k of $\phi(n)$, we test whether $S_n(a)^{\frac{\phi(n)}{k}} = 1 \pmod{n}$. If it is 1, then $S_n(a)$ is a k^{th} power, and it will be a generator of at most $\phi(n)/k$ values. By trying all the primes factors of $\phi(n)$ we can find the maximum k for which $S_n(a)^{\frac{\phi(n)}{k}} = 1 \pmod{n}$. If $k \neq 1$ signatures of the exponents should be computed $(\bmod\ \frac{\phi(n)}{k})$. This is not only more efficient, but could remove unwanted factors from $\phi(n)$ which may otherwise prevent signature computation.

2.1.7 Complex numbers

Working with complex numbers reduces to being able to compute the signature of i, the imaginary unit. $S_n(i)$ should satisfy the basic relation of $S_n(i)^2 + 1 = 0$. This means that $S_n(i) = \sqrt{-1} \bmod n$. If $n = 4k + 1$ is a prime, then -1 is a non-residue, and we can find its square root. If $n = n_1 n_2 ...$ is composite, then -1 should be a non-residue for all n_i. As a consequence n has two possible forms, $n = 4k + 1$ or $n = 8k + 2$. Once that we have chosen n so that $S_n(i)$ has a representation, we still have two choices (plus and minus). Any choice works provided that we use it consistently. This choice is not yet a problem, but will become one once that we introduce algebraic numbers in general. This will be discussed later.

Suppose we want to test whether $(x + i)(x - i) - x^2 - 1$ is identically 0. We can work with $n = 13$, where $S(i) = 5$ is a possible choice. Selecting $S(x) = 2$ as before we obtain

$$7 \times (-3) - 4 - 1 = -26 = 0 \pmod{13}$$

When we are forced to compute signatures in a field or congruence which does not allow the representation of i, we will use field extensions. This will be described for the signatures of e^x and trigonometric functions.

2.1.8 Square roots of integers

To find the signatures of square roots of integers (everything else is recursively computed to an integer) we need to resolve two problems.

(a) Some integers do not have a square root mod n.

(b) When we find a square root, we have two choices (sign choice). We must have a consistent way of choosing the sign so that we will not make mistakes with expressions like $\sqrt{35} - \sqrt{5}\sqrt{7}$. When n is not prime there will be more than two choices for the square root.

Both of these problems are also problems with algebraic numbers and will be discussed in the next part. If we are using a field extension for the representation of i, then all square roots can be represented, either by an integer or by a multiple of the field extension.

2.1.9 Algebraic numbers

An algebraic number is a value which satisfies a certain polynomial relation. E.g. Let

$$p(x) = x^3 - 2x + 5$$

then α such that $p(\alpha) = 0$ is an algebraic number. Square roots are a special case of algebraic numbers, when $\deg_x p(x) = 2$.

If the polynomial has coefficients which are not numerical constants, then α is called an algebraic function.

To obtain signatures of algebraic numbers we need to find values of $S_n(\alpha) = r$ such that $S_n(p(r)) = 0$. This can be computed by finding all the different linear factors of $p(x)$ (mod n). This is simpler than factorization, it can be obtained from the first step of the Cantor-Zassenhaus factorization algorithm.

The same problems that we have with square roots will show with algebraic numbers:

(a) $p(x)$ may have no linear factors, hence no roots mod n, hence no possible value for r.

(b) there may be more than one different linear factor and choosing is a problem.

For algebraic numbers of degree 3 or larger it is sometimes possible to find an n which produces a factorization which has a single linear factor (possibly with some multiplicity) and hence a single root. This is extremely desirable, and justifies trying several values of n to see if this is possible. When we have a unique root we can consider it as the signature for the algebraic number. In this case, since factorization requires polynomial time in the degree of $p(x)$, the signature can be computed in linear time on the size plus polynomial time on the degrees of its algebraic numbers.

If the polynomial does not have any linear factors, we do not have any choice other than selecting a new n and trying again. For random polynomials and for large values of n, $1/2 + O(1/n)$ of all polynomials of degree 3 have a single root, $1/3 + O(1/n)$ for degree 4 and $3/8 + O(1/n)$ for degree 5.

Average number of different roots

The probability of a random polynomial (mod n) of degree k having i roots is $\frac{D_{k,i}}{k!} + O(1/n)$. $D_{k,i}$ are called the *rencontre* numbers. $D_{k,1}/k!$ is the limiting value when n grows of the number of polynomials which will have a single root, and hence suitable for computing signatures. A description of these numbers can be found in [2] pp. 65. The exponential generating function for them is

$$\frac{e^{-x}(1+x^3)}{(1-x)(1-x^2)} = \sum_{k=0}^{\infty} \frac{D_{k+1,1}x^k}{k!}$$

The limiting value, when the degree of the polynomial grows, is $\lim_{k\to\infty} \frac{D_{k,1}}{k!} = e^{-1}$. In other words, for random polynomials of high degree and for large size fields, 36.79% of the time these polynomials will have a single root suitable for use as a signature.

If we have more than one linear factor for all the values of n that we are prepared to try (notice that for polynomials of degree 2 we either have two linear factors or none), then we have multiple choices. Whenever we have multiple choices we have to compute all signatures arising from them. So if we have a square root and an algebraic number which factors in no less than 3 linear terms we will have to compute $2 \times 3 = 6$ signatures.

If all the signatures are $S_n(e) \neq 0$ the expression is not zero. If all $S_n(e) = 0$ then the expression is identically zero with some probability of error. Else we cannot decide and the procedure fails.

Example. Let α and $p(x)$ be defined as before. The following is a standard normalization of a rational expression containing algebraic numbers:

$$\frac{\alpha^5 - 1}{\alpha^2 - 2} = -4/5\alpha^2 + 21/5\alpha - 5$$

To compute the signature of these expressions, we need to find an n for which $p(x)$ factors with only one linear factor. $n = 5$ will fail to compute some fractions on the right hand side. For $n = 7$, $p(x)$ does not factor. For $n = 11$,

$$p(x) = (x + 5)(x^2 + 6x + 1) \pmod{11}$$

has a single linear factor and we decide to use it. So we set $S_{11}(\alpha) = 6$ and

$$\frac{6^5 - 1}{6^2 - 2} = -4/5 \times 6^2 + 21/5 \times 6 - 5 = 9 \pmod{11}$$

as expected.

2.1.10 Exponentials, e^x

Next we will add exponentials to our repertoire of expressions for which we can compute signatures. The signature of e^x will be computed as

$$S_n(e^x) = S_n(e)^{S_{\phi(n)}(x)} \pmod{n}$$

where $S_n(e)$ is arbitrarily chosen to be a primitive root (mod n) and $\phi(n)$ is Euler's totient function.

Over rational expressions of polynomials, the only relations that $S(e^a)$ has to follow are: $S(e^0) = 1$ and $S(e^{a+b}) = S(e^a)S(e^b)$. These are satisfied with the above definition.

$S(e^a)$ should also satisfy a negative condition, which is that there is no relation $e^x \neq P(x)/Q(x)$ for any finite-degree polynomials $P(x)$ and $Q(x)$. This can be proved over the reals by showing that the expansion of $Q(x)e^x$ will have a term with degree higher than the degree of $P(x)$, and hence for sufficiently large x, $|Q(x)e^x| > |P(x)|$.

This property should also be maintained over (mod n). To prove this we make use of the operator $\Delta P(x) = P(x+1) - P(x)$. Now, each time that we apply Δ to a polynomial, we reduce its degree by one. So

$$\Delta^{d+1} P(x) = 0$$

where d is the degree of $P(x)$. Δ applied to powers $\pmod{n}$ gives:

$$\Delta E^x = E^{x+1} - E^x = (E-1)E^x$$

or

$$\Delta^{d+1} E^x = (E-1)^{d+1} E^x \neq 0$$

Suppose that $P(x)$ and E^x are claimed to be identical, then if the degree of $P(x)$ is not higher than $n-2$, we compute the sequence $P(0), P(1), ..., P(d+2)$. The Δ^{d+1} of this sequence is 0, and if it were equal to the sequence $E^0, E^1, ...E^{d+1}$ it would be non-zero, so there cannot be a polynomial that has identical signatures to E^x. The extension to rational polynomials is by observing that the $\Delta^{d+1}(Q(x)E^x) \neq 0$ too.

Choosing $S(e)$ to be a primitive root guarantees that the range of the signatures $S(e^x)$ is as large as possible, i.e. there will be $\phi(n)$ different possible values for the signature. An extremely small range would force us to reconsider the probability bounds, in particular if we were computing with polynomials on exponentials.

2.1.11 Trigonometrics and complex arguments of exponentials

The problem of computing signatures of trigonometrics is closely related to the problem of computing exponentials of complex numbers, since if signatures are computed correctly, $S(\sin x) = S(\frac{e^{ix} - e^{-ix}}{2i})$.

Exponentials of complex numbers pose a more immediate problem. We have seen that to represent i, n has to be either of the form $4k+1$ or $8k+2$. In both cases, $\phi(n)$ is a multiple of 4 and it is not possible to represent $S_{\phi(n)}(i)$. Since $\sin x$ requires i to be represented in the base and in the exponent, we need to resolve this problem if want to compute signatures of trigonometric functions.

To represent i and the square root of other non-residues we will use one field extension. We will call this extension α, and since it has no representation $\pmod{n}$ we will have to operate with it in symbolic form. Without loss of generality, we will assume that it satisfies $\alpha^2 + 1 = 0$. So when this is needed, our signatures will be of the form $a + b\alpha$ and called α-forms. (It is easy to see that the signatures of the square roots of non-residues are of the form $k\alpha$, so as a side effect we can represent the square roots of all the integers.) Arithmetic is done as with polynomials in α and every time that we generate α^2 we substitute it for -1. Inverses and divisions are done by multiplying the numerator and denominator by the conjugate with respect to α. The only problem to resolve is how to compute exponentials of α-forms, and here is where we will introduce signatures of trigonometrics.

Let $T = S(e)^r \pmod{n}$, where r and $\phi(n)$ are relatively prime, neither r nor $1/r$ $\pmod{\phi(n)}$ are *small* and r is otherwise randomly chosen. With such an r, T is also a primitive root $\pmod{n}$. Since neither r nor $1/r$ are small, there is little chance, $O(1/n)$, of polynomials in $S(e)^x$ being confused with polynomials in T^x. In what follows, n will be chosen so that i can be represented. The signatures of the trigonometric functions will be computed as follows.

$$S_n(\sin x) = \frac{T^{S_{\phi(n)}(x)} - T^{-S_{\phi(n)}(x)}}{2i} \pmod{n}$$

$$S_n(\cos x) = \frac{T^{S_{\phi(n)}(x)} + T^{-S_{\phi(n)}(x)}}{2} \pmod n$$

The rules for computing exponentials and trigonometrics with α-forms are:

$$S_n(e^\alpha) = T$$

$$S_n(T^\alpha) = 1/S_n(e) \pmod n$$

$$S_n(e^{a+b\alpha}) = S_n(e)^a T^b \pmod n$$

$$S_n(\sin(a+b\alpha)) = \frac{T^a S_n(e)^{-b} - T^{-a} S_n(e)^b}{2i} \pmod n$$

etc. In simpler terms, the symbol $S(e)^\alpha$ is assigned an arbitrary value and forced to respect the only simplification restriction which it has to follow: $(S(e)^\alpha)^\alpha = S(e)^{-1}$.

It is not difficult to verify that with these definitions, $S(\sin 0) = 0$, $S(\cos 0) = 1$, $S(\sin^2 x + \cos^2 x) = 1$, $S(\sin(a+b)) = S(\sin a \cos b + \cos a \sin b)$, etc.

2.1.12 A better method for exponentials and trigonometrics

The previous reasoning, in particular the desire of computing the signatures of exponentials with the largest possible range, prevents us from representing i in the base and in the exponent. This can be overcome if we choose $S(e)$ to be a non-primitive root.

Lets assume that $n = 4kn_2 + 1$ is a prime, and that $n_2 = 1 \pmod 4$ is also a prime. This condition is possible to satisfy, and these numbers are similar in concept to *safe primes*. With them we can represent $S_n(i)$ and also $S_{n_2}(i)$. Now we will choose $S(e) = p^{4k}$, where p is a primitive root $\pmod n$. By construction, $\phi(n) = n - 1 = 4kn_2$, so $S(e)^{n_2} = p^{\phi(n)} = 1 \pmod n$ for any choice of $S(e)$. This means that the exponent arithmetic can be done $\pmod{n_2}$, which implies that we can compute the signatures of the exponents in a field. This has many advantages, among others the ability to represent i and being free of any bad divisors. The counterpart is that the signatures of e^x will have a smaller set of possible values, n_2 instead of $n-1$, or a factor of $4k$. Assuming that k is not too large (it could just be 1), we need to choose n a factor of $4k$ larger than what we would have done. This is a small price to pay for being able to compute exponent signatures in a field.

As before, we will select $S(e)$ so that it does not satisfy any simple algebraic relationship. In this case we want to select it also such that $S(e)^i$ does not satisfy any simple relation. For example, if we choose $n = 4000133$ and $n_2 = 1000033$, which satisfy the above constraints, $p = 1351050$ is a primitive root $\pmod n$ which gives $S(e) = 266466$. Neither $S(e)$ nor $S(e)^i$ satisfy any simple algebraic relation.

Simple algebraic relations

It is important to check that these numbers, like $S(e)$ above, do not satisfy simple algebraic relationships. It would be very unfortunate that we randomly select an $S(e)$ which happens to be, for example, $S(e) = 1/3$. This can be checked quite easily using the LLL lattice reduction algorithm. To do this we construct a matrix of dimension $m \times m$ with the following structure:

$$
\begin{array}{ccccc}
1 & 0 & \dots & 0 & -S(e) \\
0 & 1 & \dots & 0 & -S(e)^2 \\
\dots & & & & \\
0 & 0 & \dots & 1 & -S(e)^{m-1} \\
0 & 0 & \dots & 0 & n
\end{array}
$$

The last row in the matrix is used to reduce the values by integer multiples of n, as should be for (mod n) computations. Every vector which is a linear combination of some of the rows, will give a polynomial in $S(e)$ of degree at most $m-1$ (mod n) that is satisfied. A *short* vector will give a simple polynomial, hence a suspicious relationship which may be better avoided. For the above choices of $S(e)$ and $m = 5$, the reduced lattice is

$$
\begin{array}{rrrrr}
5 & 0 & -15 & 3 & 14 \\
15 & 10 & -14 & 16 & -1 \\
2 & -11 & -7 & -17 & -7 \\
-7 & -11 & 7 & 18 & -5 \\
-9 & -6 & 5 & -3 & 15
\end{array}
$$

So the simplest polynomial relationship of degree four or less satisfied by $S(e)$ (from the last row) is $-9S(e) - 6S(e)^2 + 5S(e)^3 - 3S(e)^4 + 15 = 0 \pmod n$.

Under this scheme, we can now compute the trigonometric functions with the standard complex-exponential formulas:

$$T = S(e)^{S_{n_2}(i)} \pmod n$$

$$S_n(\sin x) = \frac{T^{S_{n_2}(x)} - T^{-S_{n_2}(x)}}{2i}$$

$$S_n(\cos x) = \frac{T^{S_{n_2}(x)} + T^{-S_{n_2}(x)}}{2}$$

All the trigonometric relationships, including the relations with the complex exponentials will be verified. For the numerical example, $S_n(i) = 917163$, $S_{n_2}(i) = 649529$ and $T = 2888692$.

Primes suitable for nested exponentials

In general, if we want to handle several levels of nested exponentials, we will have to precompute several primes $n_1, n_2, \dots$ so that $n_{i+1} = 4k_in_i + 1$. This will make it possible to represent i at all levels. The values of k_i should be chosen as small as possible, as we want to keep the ratio between the largest and smallest primes small. The following Maple code generates a sequence of 5 primes with this property. It should be noticed that the ratio between $n_5/n_1 = 1536$. The minimum ratio, for all $k_i = 1$, or 256 is not attainable.

```
n1 := 10^9:
maxn5 := n1*10^10:

to 10000 do
  n1 := nextprime(n1);
  if modp(n1,4) = 1 then for n2 from 4*n1+1 by 4*n1 while 64*n2+21 < maxn5 do
    if isprime(n2) then for n3 from 4*n2+1 by 4*n2 while 16*n3+5 < maxn5 do
      if isprime(n3) then for n4 from 4*n3+1 by 4*n3 while 4*n4+1 < maxn5 do
        if isprime(n4) then for n5 from 4*n4+1 by 4*n4 while n5 < maxn5 do
          if isprime(n5) then
```

```
      maxn5 := n5;
      lprint(n1,n2,n3,n4,n5)
    fi od fi od fi od fi od fi od:
```

The best set of primes found with the above is 1000077157 4000308629 48003703549 192014814197 1536118513577.

2.1.13 Computing signatures of logarithms and inverse trigonometrics

Computing signatures of logarithms is more complicated in the sense that we do not know how to compute them in time polynomial in the size of the problem. The signatures will be computed as

$$S_n(\log a) = \log_E(S_m(a)) \pmod n$$

where n and $m = kn+1$ are primes, $E = S_m(e) = p^k$ is the base of the exponential function and p is a primitive root $\pmod m$. The log is the discrete logarithm, sometimes called the index function. It is defined by

$$\log_E b = c \pmod n \iff E^c = b \pmod m$$

Notice the different ground fields on which we are computing here. This is similar to the exp function, where the exponent arithmetic was done on a different field, but in reverse.

If the signature of the argument is 0, we cannot compute the discrete logarithm. This is a case analogous to a division by zero, and we will proceed as with it. I.e. we will recursively test whether the argument is 0. If it is, then the computation does not make sense and it is aborted. If it is not, then we have found an n for which the signature is non-zero, and we continue the computation in this field.

With such a value for E, not being a primitive root, there will be discrete logarithms that will not be possible to compute. This problem is tricky to solve, but we can use the fact that the range of the discrete logarithm is normally $0...m-2$, and we fold this range to $0...n-1$. Let $r = p^n$, then r is a k^{th} root of unity, and $\log_p r = n$. Multiplying the arguments of the logarithms by any power of r will not change their signature. If the base of the logarithms is p, then it is clear that

$$S_n(\log rx) = n + S_n(\log x) = S_n(\log x) \pmod n$$

In other words, the signatures of logarithms group values of the arguments in equivalence classes of size k. The generator of these classes of equivalence is r. When we compute discrete logarithms base E, then this class of equivalence will have one member having a logarithm, and all the others being not representable. We will use as a signature of any argument, the signature of any of the members of its equivalence class. It turns out that this is equivalent to computing the discrete logarithms as:

$$S_n(\log a) = \frac{\log_p a}{k} \pmod n$$

A numerical example helps to understands all these computations and problems. Assume that $n = 13$ and $m = 53$ and hence $k = 4$. Let $p = 2$ which is a primitive root (mod

53). Then $E = S_{53}(e) = p^k = 16 \pmod{53}$ and $r = p^n \pmod{53} = 30$. We would like to prove that

$$\log 8e^9 = \log 8 + 9 \implies S_n(\log S_m(8e^9)) = S_n(\log S_m(8)) + 9$$

The signatures of the arguments are $S_m(8e^9) = 8E^9 = 23 \pmod{53}$ and $S_m(8) = 8$. Now, $\log_E 8 \pmod{53}$ does not exist. I.e. there is no value c such that $E^c = 8 \pmod{53}$. But 8 is in a class of equivalence $\{8, 8r, 8r^3, 8r^3\} = \{8, 28, 45, 25\} \pmod{53}$. Of the four components of this equivalence class, only 28 has a discrete logarithm base E, $\log_E 28 = 4 \pmod{53}$. Similarly for 23, $\{23, 23r, 23r^3, 23r^3\} = \{23, 1, 30, 52\} \pmod{53}$, and only 1 has a logarithm, $\log_E 1 = 0$. Notice that $\frac{\log_p 8}{4} = 3/4 = 4 \pmod{13}$ and $\frac{\log_p 23}{4} = 39/4 = 0 \pmod{13}$. This gives $0 = 4 + 9 \pmod{13}$ which verifies the initial equation.

2.2 Non-standard uses of signatures for general problems

The game of Mühle (9 Men Morris) is a common game in Europe. It is played on a square board consisting of three concentric squares. Black and white pebbles (or men) are placed on the crossings and later moved around. The following is a picture of the board with the playing positions numbered from 1 to 24.

```
1--------------------2--------------------3
|                    |                    |
|                    |                    |
|      9-------------10------------11     |
|      |             |             |      |
|      |             |             |      |
|      |    17------18-------19    |      |
|      |    |                 |    |      |
|      |    |                 |    |      |
8-----16----24               20----12-----4
|      |    |                 |    |      |
|      |    |                 |    |      |
|      |    23------22-------21    |      |
|      |             |             |      |
|      |             |             |      |
|     15------------14-------------13     |
|                    |                    |
|                    |                    |
7--------------------6--------------------5
```

For the purposes of building a table of boards and searching particular configurations in this table, it is extremely appealing to have a signature for each instance of the board. By a particular instance we mean several black and white pebbles in various positions.

Notice that the board instances, for the purposes of strategy, validity, moves, etc. form classes of equivalence. Each class has a maximum of 32 different boards and these are generated by:

4 rotations

2 axial symmetries

2 inner-outer square swaps

2 colour swaps

(Whether it is a white move or a black move is not part of the board, and hence it should be handled separately.)

Because of the large number of possible boards, it is assumed that that each equivalence class of boards will be stored only once. This will have an extra cost that will be paid at searching and/or insertion time. For this there are two alternatives

(1) Insert any of the boards at insertion time. At searching time we will have to search all of the members of the equivalence class, that is up to 32 searches.

(2) Insert a canonical one (it is quite easy to select a canonical one when we use signatures, we can select the one with the smallest signature). At searching time we search only once after having selected the signature of the canonical one.

The second policy is superior as we do only one search in the database. Notice that following a successful signature search we still have to compare the boards. Different boards may have equal signatures. This comparison will seldom fail, it will almost always return true. I.e. signature collisions are rare, they occur with probability $O(1/n)$.

Additive signature functions for Mühle.

Let the positions in the board be numbered from 1 to 24 as shown above. Let W_i and B_i be random numbers chosen uniformly from $[0, n-1]$ unless otherwise stated. A linear signature is defined by

$$S(board) = \sum_{i=1}^{24} (W_i \delta_i + B_i \gamma_i) \pmod{n}$$

where $\delta_i = 1$ when there is a white pebble in position i and it is 0 otherwise and similarly $\gamma_i = 1$ for the black. The main operation in a linear signature is multiplication by a factor, i.e. αS, which is equivalent to multiplying each B_i and W_i by α. If the number of men, k, is known, a linear transformation of the form $\alpha S + k\beta$, will map every B_i to $\alpha B_i + \beta$, and similarly for W_i.

Rotations of the board.

To do rotations by multiplication the following equations must hold:

$$\alpha B_1 = B_3, \quad \alpha B_3 = B_5, \quad \alpha B_5 = B_7, \quad \alpha B_7 = B_1$$

etc. This means that $\alpha^4 - 1 = (\alpha + 1)(\alpha - 1)(\alpha^2 + 1) = 0 \pmod{n}$. We can choose α to be a non-trivial root of the above, for example for $n = 123457$, $\alpha = 7738$, and if we compute all the W_i from $W_1, W_2, W_9, W_{10}, W_{17}, W_{18}$ and similarly for B_i we can rotate the board 90 degrees by multiplication by α. This has a flaw which is difficult to cure, as even for the non-trivial α, $\alpha^2 = -1$. This means that diametrically opposed men of the same colour will cancel out. This cancellation is bad as it happens for any possible value of n and confuses a board with no men in two corners with a similar board with two equally coloured men in these corners. Other cancellations are also possible. Consequently we will not use this technique to compute rotations.

Colour swaps.

If we select $W_i = -B_i \pmod n$, colour swap is achieved by complementing the signature. Cancellations cannot happen as we cannot have two men in a single position. It we have no other conditions on the W_i and B_i this is a good solution.

Inner-outer square swaps.

This is not possible to achieve by simple multiplication, as

$$\alpha B_1 = B_{17}, \quad \alpha B_{17} = B_1, \quad \alpha B_9 = B_9$$

has only one non-interesting solution. A relatively general technique may be used to solve this problem. If we define a new signature over a single square as

$$S^*(board) = \sum_{i=1}^{8}(W_i\delta_i + B_i\gamma_i) \pmod n$$

then the signature of the entire board can be represented by a triplet

$$S(board) = (S^*(outer), S^*(middle), S^*(inner))$$

The inner-outer swap of the signature (a, b, c) is (c, b, a). If this is combined with the colour swap as described above, then the four combinations of colour swap and inner-outer swaps are

$$(a, b, c), \quad (c, b, a), \quad (-a, -b, -c), \quad (-c, -b, -a)$$

Since we are interested in the minimum signature (in the case of a tuple of values, the ordering will be a lexicographical ordering of the elements left to right), we first choose the minimum between $a, -a, c, -c$ and only in cases of multiple minima we need to compare further.

Axial symmetries.

There is again no obvious solution for symmetries, as a vertical axis symmetry would have to satisfy:

$$\alpha W_1 = W_3, \quad \alpha W_3 = W_1, \quad \alpha W_2 = W_2$$

which gives non-interesting solutions.

Question / exercise.

What are the advantages and disadvantages of the signature defined by the tuple:

$$S(board) = (S^*(outer)S^*(middle), S^*(inner)S^*(middle))$$

As a final remark, it should be noticed that in reducing the signatures of the entire board to 3 signatures over squares, the cardinality of each individual S^* is only $3^8 = 6561$. So at this point it is probably more useful to use an exact mapping between squares and integers, for example

$$S^*(square) = \sum_{i=1}^{8} 3^{i-1}(\delta_i + 2\gamma_i)$$

Rotations by 90 degrees and symmetries can now be precomputed and stored in tables, their size will be reasonable for any computer. Then all the computation of signatures is reduced to table lookups and the operations done for colour and inner-outer swaps. The resulting signatures will also be unique, so this will save the comparison for equal boards once that the signatures match. This is interesting because with the ideas of signatures we have gone back to a better deterministic algorithm.

2.3 Open problems with signatures

There are several open problems with signatures. Some we know that are likely to be impossible to solve, as their solution would imply that P=NP. Others are still open and their solution may have a truly profound impact in practical and effective computer algebra.

2.3.1 Signatures of boolean expressions

The signature of a boolean expression, that is an expression using unknowns and the operators `and`, `or` and `not` may be easily computed mod 2. Unfortunately we do not know how to compute these signatures for general n. Computing mod 2 is equivalent to assigning random values to the variables and testing. Such test may have an exponentially close to 1 probability of failure, and hence useless. These tests can be applied in parallel, for example using bitwise boolean operations, on words of 32 or 64 bits. This is just equivalent doing so many tests sequentially. Finding a good signature, i.e. one with a probability of failure less than 1/2, would imply that equivalence of boolean expressions is in RPT (random polynomial time algorithm) and hence P=NP.

2.3.2 Signature of π

We do not know how to compute the signature of π within arguments of trigonometrics. π plays a very specific role in trigonometric functions. So far all attempts to find $S_n(\pi)$ so that the trigonometric functions compute correctly yield values which make too many mistakes. The reason for the apparent failures to find a signature are illustrated in the following. Since $\sin(k\pi) = 0$ for integer k and $\sin(e\pi)$ is reduced to $\sin(S_n(e)S_n(\pi))$ and all signatures are integers, the signature of sin of any expression times π becomes 0. In other words, the condition k is an integer is not unique when we are dealing with signatures, all expressions are mapped to integers. New ideas are needed to resolve this problem.

2.3.3 Signature of $\Gamma(k)$

The signature of $\Gamma(k)$ or at least the signature of $k!$ are very expensive, $O(k)$ operations, to compute. This is not good, as for an unknown x, $\Gamma(x)$ would require the computation of $S_n(x)!$ which is $O(n)$. This question in itself is very interesting. Is it possible to compute $k! \bmod n$ in time less than $O(k)$? This question is open, and very doubtful. If it would be possible, we could do integer factorization in the same time. Non-trivial factors of n can be computed from by $\gcd(k! \bmod n, n)$. Is it possible to fake the signature of $\Gamma(k)$? The only relations between $\Gamma(n)$ and $\Gamma(m)$ are when $n-m$ is an integer and the expression is of size $O(|n-m|)$. Additional care has to be taken with the half argument relations,

$$\Gamma(2n) = \frac{\Gamma(n)\Gamma(n+1/2)}{2\sqrt{\pi}}$$

its extensions and with the reflection formula.

$$\Gamma(z)\Gamma(1-z) = \frac{\pi}{\sin(\pi z)}$$

2.3.4 Signature of a derivative

A partially resolved problem is the computation of the signature of the derivative of an expression with respect to a given variable, say x. This can be done by recursing on the original expression, but it is not possible to do (or at least not known to us) just from the signatures. The rules for computing the signature of a derivative are straightforward, i.e.

$$S_n(D_x(ab)) = S_n(D_x(a))S_n(b) + S_n(a)S_n(D_x(b))$$

$$S_n(D_x(\text{constant})) = 0$$

$$S_n(D_x(x)) = 1$$

etc. In general we want to compute signatures from signatures, and not recursively on the original expression. Just from the complexity point of view, if intermediate results are not stored, the above computation would be exponential in the number of nested products. This is clearly undesirable. In other words, an optimal signature is one which can be computed as

$$S_n(g(a,b,...)) = g^*(S_n(a), S_n(b), ...)$$

This does not appear to be possible for the derivative. Computing the signature of an indefinite integral does not make sense as there is an arbitrary integration constant which makes any result possible. Notice that the integration constant may be hidden in the expression, and hence impossible to isolate. For example

$$\int \frac{1}{x} dx = \ln 2x$$

is a correct indefinite integral with a hidden integration constant. Although this is still partly open, it is possible to verify that a given expression is the integral of another. All what has been said about integration and differentiation can be mapped one-to-one to summations and the Δ difference operator.

2.3.5 Definite integrals

The notion of limiting sum cannot be expressed as a signature. There is no notion of approximation with signatures, arbitrarily close rational numbers can have arbitrarily different signatures. Consequently, under our present knowledge of signatures, it does not appear possible to compute signatures of definite integrals, definite sums, limits and similar operations. Additionally, integrals or summations with infinite limits are completely outside the reach of signatures for an even stronger reason.

2.3.6 Signatures of sets

It is possible to compute the signature of a set with non-repeated elements, but it appears to be impossible to compute the signatures of sets with repeated elements. Also, no signatures are known for the set operations (union, intersection, negation, set difference). This last fact is not surprising, as there is an equivalence between boolean operations and set operations.

Chapter 3

Computation with algebraic numbers

The problem that we want to address here is how to compute effectively with algebraic numbers and algebraic functions. Part of the algorithms presented here were studied and developed by Trevor J. Smedley for his thesis dissertation. His thesis and related papers are the main references for this chapter.

What do we mean by computation with algebraic numbers?

- Normalizing expressions, i.e. representing every algebraic number of arithmetic expression involving algebraic numbers, into a canonical, simplified form. For algebraic numbers of order k, all powers k or larger will be reduced.
- Rationalize expressions. Remove the occurrence of the algebraic numbers in the denominator of expressions.
- Gcd computations.
- Polynomial factorization.
- Other polynomial operations, e.g. resultants.

There are deterministic conventional algorithms for all of the above problems, but these are, in some cases, very difficult to implement and always extremely expensive to run.

The heuristic algorithms presented in this chapter are two variations of the same flavour. In both cases we reduce our domain from polynomials over the integers with the addition of algebraic numbers to a domain of polynomials over the integers mod n and without algebraic numbers. In the first method, the one described in Smedley et al., we select the representation of α so that it is easy to lift the results. For the second method, we rely on the LLL algorithm for lifting the results. Complete algorithms for the first method are currently implemented and used in Maple.

Let α be an algebraic number defined by the polynomial $p(x)$ with $p(\alpha) = 0$. Without loss of generality we will assume that $p(x)$ is an primitive, irreducible polynomial over the integers of degree 2 or higher. The degree of the algebraic number α, is the degree of the polynomial, $d = \deg p(x)$. If the leading coefficient of $p(x)$ is 1 or -1, then α will be called an algebraic integer.

From the above definitions and the proposed mapping, it is obvious that $S_n(\alpha)$ should be selected so that $p(S_n(\alpha)) = 0 \bmod n$, or that $S_n(\alpha)$ is a root of $p(x) \bmod n$ (or that $(x - S_n(\alpha))$ is a factor of $p(x) \bmod n$).

Example. Suppose we want to simplify α^5, where α is an algebraic number of degree 3, i.e. $d = 3$. We know that the answer will be of the form

$$\alpha^5 = a_0 + a_1\alpha + a_2\alpha^2$$

If we choose α to be a root of $p(x) \bmod n$, the resulting signature for α^5 is typically a number as large as n and it is not posible to easily retrieve a_0, a_1 and a_2. Using the same ideas that we used in GCDHEU, we should try to obtain an $S_n(\alpha)$ which is very small with respect to n, so that then the coefficients a_i can be lifted with the mods algorithm described before.

It is not very difficult to find an n such that $p(S_n(\alpha)) = 0$ for small $S_n(\alpha)$. But first lets be a bit more precise. Lets assume that $\max_i |a_i| = a$. So $S_n(\alpha) > 2a$ just to be able to lift the coefficients. In particular, $S_n(\alpha)^{d-1}a < n$ to be able to lift the last coefficient. In summary this means that

$$(2a)^d < 2n$$

The procedure to select $S_n(\alpha)$ and n witht the above conditions is as follows:

You can experiment with other values for `a` by changing its assignment, or with other expressions in α by changing the assignment to
tt S.

There is an alternative approach to solve this problem which does not require the searching of a suitable prime (which in turns requires a partial factorization). This alternative also resolves the problem of common denominators which are very difficult to estimate correctly. It is based on the LLL algorithm and proceeds as follows.

For some arbitrary expression, e involving an algebraic number α, we compute its signature using some prime number n. This prime number n is chosen so that the characteristic polynomial $p(x)$ has at least one linear factor $\pmod n$. (Obtaining the linear factors of a polynomial $\pmod n$ is much easier than factoring.) We assume that our expression e can be expressed in terms of the algebraic number α as

$$e = r_0 + r_1\alpha + r_2\alpha^2$$

for rational numbers r_i (usually with a common denominator). For our use of the LLL algorithm, we assume that these r_i rationals are composed of "small" numerators and denominators, at least small compared with n. The matrix is set up as usual. The first column should satisfy the equality of the signature of e and the signature of the expression in the r_i. We multiply this column by a large integer to insure that most of the answer vectors will have a 0 (exact satisfaction) in this position. The second column will have the multiplier of the signature of e, that will play the role of the common denominator of all the r_i. The remaining colums give the numerators of the r_i.

Chapter 4

Algorithms for algebraic pattern matching

Pattern matching is one of the most desirable facilities in a computer algebra environment, yet one of the most difficult ones to implement. Part of the difficulty lies in the definition itself. There are two basic definitions and many grades in between. What is desirable for some users may not be for others, and what may be desirable for others may be too complicated to compute effectively.

4.1 Definitions

We first define algebraic pattern matching, the one which interests us in a computer algebra environment. Then we will define structural pattern matching, a simpler and different type of matching.

Algebraic pattern matching is defined as solving the equation

$$F(x, y, ...a, b, ...) \equiv G(x, y, ...m, n, ...)$$

by finding

$$m = S_m(a, b, ...)$$
$$n = S_n(a, b, ...)$$

etc. such that substituting m, n, etc. by the above expressions gives an identity in x, y, etc. Or

$$F(x, y, ...a, b, ...) - G(x, y, ...S_m(a, b, ...), S_n(a, b, ...), ...) \equiv 0$$

x, y, ... are called the *main* variables,
a, b, ... are called the *parameters* and
m, n, ... are called the *pattern* variables.

The problem is often extended to dictionaries of patterns. In this case we are searching among a set of patterns G_i and we would like to find all the ones which match a given F. In such a situation, we expect to do better than just a sequential search over each pattern.

Some examples,

expression	pattern	result
$3x^2+x-1$	$a_2x^2+a_1x+a_0$	$a_2=3,\ \ a_1=1,\ \ a_0=-1$
$\cos x$	$\sin ax+b$	$a=\pm 1,\ \ b=\pi/2$
e^x	$a_0\sin a_1x+b_0\cos b_1x$	$a_0=\pm i,\ a_1=-a_0,\ b_0=1,\ b_1\pm i$
x^3+x-2	$(x+a_0)(x^2+b_1x+b_0)$	$a_0=-1,\ b_1=1,\ b_0=2$
x	$x^r\sin^p x\cos^q x$	$p=0,\ \ q=0,\ \ r=1$
$\frac{1}{x^2+3x+2}$	$\frac{A}{x+a_0}+\frac{B}{x+b_0}$	$B=1,\ A=-1,\ a_0=2,\ b_0=1$

From the above examples, it is easy to see that the power of algebraic pattern matching is quite substantial and will include most of other operations, like working with complex numbers and algebraic extensions, factoring, gcds, etc.

Structural pattern matching, at the other end, attempts to match a particular form to a mathematical expression. It is like a data structure problem, or a typing problem. Often structural pattern matching does not require the definition of main variables. Only the first of the above examples would record a structural match. The following examples illustrate structural matching more precisely

pattern	matches	does not match
$m_2x^2+m_1x+m_0$	$3x^2+x-1$	$3x^2$
$\cos(m)$	$\cos(3x^2-1)$	$\sin(3x^2-1+\pi/2)$
$m+np$	$1+2x$	$x+y$
m:int $+\ n$:float $\times\ p$:var	$1.2a-1$	$12a-1$

Often the pattern variables are allowed to carry typing information, as it is shown in the last line.

There are many possibilities in between these two extremes. Systems will normally implement more than just purely structural matching, often under a form that allows for the matching of missing arguments making the appropriate zero assignments.

Except for the simplest structural matching, all the matching problems are very difficult and in many cases unsolvable. For this reason, heuristic algorithms are the only hope of solving some classes of subproblems.

4.2 Various methods for algebraic pattern matching

Without trying to survey the field, we describe here some of the most successful heuristics for algebraic pattern matching.

4.2.1 Series method

This method expands the expression

$$F(x, y, ...a, b, ...) - G(x, y, ...m, n, ...)$$

as a multivariate series in x, y, ... (all the main variables). If a match exists, all the coefficients must be equal to zero. So the matching problem is reduced to solving a system of equations in the parameter variables. Notice that each coefficient of the multivariate series will not contain any occurrence of the main variables, hence the coefficients are expressions in the parameters and the pattern variables. A solution of equating several coefficients to 0 in terms of the pattern variables should be substituted and checked in the original pattern. If at any point, a coefficient cannot be equated to zero, then there is no solution (provable) to the matching problem.

For example, to solve the matching of

$$\cos x \equiv \sin ax + b$$

we expand $\cos x - (\sin ax + b)$ around $x = 0$ and obtain

$$1 - \sin(b) - \cos(b)ax + \left(1/2\sin(b)a^2 - 1/2\right)x^2 + O(x^3)$$

The coefficient of the series must be equal to zero, this means

$$1 - \sin(b) = 0$$

$$-\cos(b)a = 0$$

$$1/2\sin(b)a^2 - 1/2 = 0$$

From the first equation we conclude that $b = \pi/2 + 2n\pi$ for any integer n. This value of b satisfies the second equation. The third equation becomes $a^2 - 1 = 0$ which has two solutions $a \pm 1$. Plugging in any of these solutions in the original pattern, and simplifying, we arrive at $\cos x$ and hence they are a solution of our matching problem.

This heuristic is very powerful, but it has several pitfalls. Some of them are avoidable.

Suitable point for expansion

In the above example we expanded the series around 0, and it worked. Expanding around any arbitrary point should also work, but the complexity of the resulting equations may be too high to produce a solution.

In other cases, the series may not be possible to expand at the given point. For example

$$x - x^r \sin^p x \cos^q x$$

cannot be expanded around $x = 0$ because the pattern part has its leading coefficient powered to $r + p$, which is unknown. In such cases, most systems fail to expand, and the algorithm mentioned above will not work. Expanding around $x = \pi/2$ does not work either, as $\cos x$ will have an unknown exponent in its leading term. Expanding around $x = 1$ (or around any other non-special value) gives a system of equations which is virtually impossible to solve.

The matching $\ln 2x \equiv a \ln x + b$, will fail if expanded at $x = 0$ because there is no series in x (the series is in $\ln x$). In this case, expanding around $x = 1$ works well

$$\ln 2x - (a \ln x + b) = \ln(2) - b + (1 - a)(x - 1) + O((x - 1)^2)$$

giving the solution $b = \ln 2$, $a = 1$ from the first two coefficients.

For simplicity, in particular for simplicity of the resulting system of equations, it is best to expand at *simple* values. If these do not work, expanding at a random rational should work. The main drawback of using random points of expansion is that the corresponding system of equations may be too complicated to solve.

How many terms to use

It is not clear how many coefficients should be used. We may need more, the same, or less than the number of pattern variables. Coefficients in a multivariate series typically became larger hence we would like to use the smallest first few coefficients. This suggests an iterative approach. Compute the first non-zero coefficient of the series, and solve it for any of the parameters. Plug in the result in the original pattern and expand again. This guarantees to use the right number of coefficients. On the other hand it forces several invocations of the solver, and sometimes it is better to give the whole system to the solver and let it do the best possible job.

4.2.2 Heuristics based on transcendentality

This is a useful heuristic, quite effective, but not widely applicable. The idea is best illustrated with an example. Suppose that our expression to match is a rational polynomial in x. Suppose that the pattern has terms in x and in e^x. Since we know that no polynomial in x is equal to any polynomial in e^x, then then only possible matching is one which eliminates all appearances of e^x in the pattern.

4.2.3 Top-down heuristics

4.3 Reduction-functions heuristics

Bibliography

[1] Bruce W. Char, Keith O. Geddes, and Gaston G. Gonnet. Gcdheu: Heuristic polynomial gcd algorithm based on integer gcd computations. In *Proceedings of EUROSAM 84*, volume LNCS 174, pages 285–296. Springer Verlag, 1984.

[2] John Riordan. *An Introduction to Combinatorial Analysis*. Princeton University Press, 1978.