UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS 282 **Prof. R. Fateman**
Spring, 2000

Additional Notes on Polynomial GCDs, Hensel construction

These notes cover a number of topics that are covered in any of the typical texts. We provide this discussion here to try to touch on some of the highlights and offer some perspective.

First we demonstrate that interpolation can be done as a special case of Garner's Algorithm by appropriately choosing our (relatively prime) moduli.

As an example, we choose the moduli m_i, to be linear polynomials,

$$\begin{aligned} m_1 &= x - b_1 \\ m_2 &= x - b_2 \\ \ldots & \\ m_n &= x - b_n \end{aligned}$$

and by $f(x)$ mod m_i, we shall mean $f(b_i)$. Recall from lecture that we compute

$$\begin{aligned} c_1 &= m_1^{-1} \bmod m_2 \\ c_2 &= (m_2 m_1)^{-1} \bmod m_3 \\ \ldots & \\ c_{n-1} &= (m_{n-1} \ldots m_1)^{-1} \bmod m_n \end{aligned}$$

If we are now given $n+1$ points $(b_1, u_1), \ldots (b_{n+1}, u_{n+1})$, we may compute the n^{th} degree polynomial which goes through these points by Garner's Algorithm.

Example: Suppose $m_1 = x$, $m_2 = x - 1$, and $m_3 = x - 2$. Then

$$\begin{aligned} c_1 &= m_1^{-1} \bmod m_2 = (x^{-1})_{x=1} = 1 \\ c_2 &= (m_2 m_1)^{-1} \bmod m_3 = (x^{-1}(x-1)^{-1})_{x=2} = 1/2 \end{aligned}$$

Now suppose that we wish to determine the polynomial of degree two passing through the points (0,5), (1,2) and (2,1). Then

$$\begin{aligned}
v_1 &= u_1 \bmod m_1 = (5)_{x=0} = 5 \\
v_2 &= (u_2 - v_1) \cdot c_1 \bmod m_2 = ((2-5)\cdot 1)_{x=1} = -3 \\
v_3 &= (u_3 - (v_1 + m_1 v_2)) \cdot c_2 \bmod m_3 \\
&= (1 - (5+x)\cdot(-3)\cdot 1/2)\Big|_{x=2} = (\frac{3x-4}{2})\Big|_{x=2} = 1
\end{aligned}$$

and now

$$u = v_3 m_2 m_1 + v_2 m_1 + v_1 = 1 \cdot x \cdot (x-1) + (-3)\cdot x + 5 = x^2 - 4x + 5$$

The above interpolation algorithm computes the unique polynomial of degree n passing through the given $n-1$ distinct points.

The interpolation algorithm given above is one of a number of alternatives. This is usually called Newton's Interpolation Formula. Other formulations (basically rearrangements with the same number of operations) include Lagrange and divided differences interpolation. Additional material on interpolation may be found in any introductory numerical analysis, or see Knuth's reference [1]. The relevant time bounds for any of these methods for interpolation at an arbitrary set of points are are $n(n+1)/2$ divisions (or evaluations) and $n(n+1)$ subtractions. The generalization of this to $v > 1$ variables represents a considerable cost in the interpolation phase of the modular GCD algorithm. It will turn out that we can improve upon this, under certain circumstances, by using another approach.

In our lectures we discussed sparse interpolation as invented by R. Zippel [1979] for his sparse polynomial GCD.

The best polynomial GCD algorithms in the early 1970s appear to be the ones based on Hensel's Lemma or variants. Let's give this a try:

Let us try to convince you of the relevance of *p-adic Representation* here.

Recall the notion of p-adic number representation as the representation of integers (or reals) in various number "systems". We are all familiar with decimal, binary and other representations of numbers (We used base-3 representations in class.). We may even construct system with somewhat more exotic bases, such as negative bases and, as we shall see shortly, polynomial bases. If p and N are integers, then the p-adic representation of N is $(a_0, a_1, \ldots, a_m)$ where for $i = 1, \ldots, m$ we have $0 < a_i < p$

$$N = a_0 p^0 + a_1 p^1 + \ldots + a_m p^m$$

For instance, the 3-adic representation of the integer 65 is (2,0,1,2).

We can extend this notion to polynomials. If $p(x)$ and $q(x)$ are polynomials, then the p-adic representation of q is $(a_1, a_2, \ldots, a_m)$ where for $i = 1, \ldots, m-1$, a_i is in an integer, a_m is a nonzero integer and

$$q(x) = a_0 p(x)^0 + a_1 p(x)^1 + \ldots + a_m p(x)^m$$

For instance the $(x-1) - adic$ representation of x^3 is (1,3,3,1).

Such items are mere curiosities, as you have also seen the representation of various numbers *approximately, and to successively higher "accuracy"* 3-adically, including $\sqrt{7}$ and $1/2$.

We shall now consider how this might be used as a way of starting from an answer we know only "approximately" modulo p to successively more accurate "higher degree" approximations modulo p^k. Eventually, for large enough k, these approximations will be complete. And if we have been able to do this faster than by interpolation, we can save time. (For sparse problems, this is what happens).

The Hensel Construction

Our major application for improving our approximations is to have a method of computing from a factorization of a polynomial in a finite field, a factorization of the polynomial in a larger computation structure. Note that this is quite close to what we do when computing GCDs via the modular algorithm, building up the image of the GCD to a polynomial of higher degree or a polynomial in more variables. The alternative we consider here is that given through the Hensel Algorithm, described here and first suggested for this application in reference [2].

The original linear Hensel Construction lifts a factorization from mod p^i to mod p^{i+1} at the i^{th} step while the more quadratic construction due to Zassenhaus lifts a factorization from mod p^{2^i} to mod $p^{2^{i+1}}$. Nevertheless, the linear version may require so much less computation at each step that it may be the algorithm of choice in lifting to a given modulus.

The Linear Hensel Algorithm

Let $u(x)$ be a monic[1] (leading coefficient = 1) polynomial in $Z[x]$ and assume $v_1(x) \cdot w_1(x) = u(x) \bmod p$ and $GCD(v_1, w_1) = 1 \bmod p$ where p is a prime. The algorithm computes a sequence of pairs of polynomials $((v_i, w_i))_{i=1,\ldots,n}$ such that

$$v_i(x) \cdot w_i(x) = u(x) \bmod p^i$$

Step I

Compute by means of the Extended Euclidean Algorithm generalized to polynomials, two polynomials $a(x)$ and $b(x)$ in $Z_p[x]$ such that

$$\begin{array}{rc} (i) & deg(a) < deg(w_1), \\ (ii) & deg(b) < deg(v_1) \\ (iii) & (a(x)v_1(x) + b(x)w_1(x) = 1) \bmod p \end{array}$$

Step II

Now suppose we are given (v_i, w_i) and we wish to compute (v_{i+1}, w_{i+1}). Compute a polynomial c_i such that

[1]This assumption is, in general, unwarranted, and overcoming it makes the algorithm messy.

$$(p^i c_i(x) = v_i(x)w_i(x) - u(x)) \bmod p^{i+1}$$

Step III

Compute (by polynomial division of $a(x)c_i(x)$ by $w_1(x)$) the quotient $q_i(x)$ and remainder $a_i(x)$ such that

$$a(x)c_i(x) = q_i(x)w_1(x) + a_i(x) \bmod p$$

and set

$$b_i(x) := (b(x)c_i(x) + q_i(x)v_1(x)) \bmod p$$

Observe that $deg(a_i) < deg(w_1), deg(b_i) < deg(v_i)$ and mod p,

$$\begin{aligned} a_i v_1 + b_i w_1 &= (a \cdot c_i - w_1 q_i)v_1 + (b \cdot c_i + v_1 q_i)w_1 \\ &= c_i(a \cdot v_1 + b \cdot w_1) \\ &= c_i \end{aligned}$$

so in $Z[x]$, (or $Z_{p^2}[x]$)

$$a_i v_1 + b_i w_1 = c_i + p \cdot d_i(x)$$

for some $d_i(x)$ in $Z[x]$.

Step IV

Set

$$\begin{aligned} v_{i+1}(x) &:= v_i(x) - p^i b_i(x) \bmod p^{i+1} \\ w_{i+1}(x) &:= w_i(x) - p^i a_i(x) \bmod p^{i+1} \end{aligned}$$

Observe that, over the integers

$$\begin{aligned} v_{i+1}(x)w_{i+1}(x) &= v_i(x)w_i(x) - p^i[a_i(x)v_i(x) + b_i(x)w_i(x)] + p^{2i} \cdot a_i(x)b_i(x) \\ &= u(x) + p^i \cdot c_i(x) - p^i[c_i(x) + p \cdot d_i(x)] + p^{2i} \cdot a_i(x)b_i(x) \\ &= u(x) - p^{i+1}[d_i(x) - p^{i-1} \cdot a_i(x)b_i(x)] \\ &= u(x) + p^{i+1} \cdot c_{i+1}(x) \end{aligned}$$

that is,

$$v_{i+1}(x)w_{i+1}(x) = u(x) \bmod p^{i+1}$$

Example

Suppose $u(x) = x^2 + 27x + 176$. In order to improve readability, we will do arithmetic modulo a conveniently small prime, namely 3. Knowing that we will have only positive coefficients in the factors, we can use a positive representation of the elements, (0,1,2) rather than the balanced representation (-1,0,1). Really, we do this to reduce the number of iterations needed to reach the right p-adic answer. Then,

$$\begin{aligned} u(x) &= x^2 + 2 \bmod 3 \\ &= (x+1)(x+2) \bmod 3 \end{aligned}$$

and so let $v_1(x) = x + 1$ and $w_1(x) = x + 2$.

We now compute an a and b such that, $a(x+1) + b(x+2) = 1 \bmod 3$ and find $a = 2$ and $b = 1$.

We compute $c_1(x)$ such that $3c_1(x) = v_1(x) - u(x) \bmod 9$, that is,

$$\begin{aligned} 3c_1(x) &= (x+1)(x+2) - (x^2 + 27x + 176) \bmod 9 \\ &= 3x + 6 \bmod 9 \end{aligned}$$

and so,

$$c_1(x) = x + 2$$

Now,

$$\begin{aligned} q_1(x) &= 2, \\ a_1(x) &= 0 \\ b_1(x) &= 1 \end{aligned}$$

We compute

$$\begin{aligned} v_2(x) &= (x+1) - (3) \cdot (1) \bmod 9 \\ &= x + 7 \\ w_2(x) &= (x+2) = (3) \cdot (0) \bmod 9 \\ &= x + 2 \end{aligned}$$

so that

$$x^2 + 27x + 176 = (x+7) \cdot (x+2) \bmod 9$$

To raise the factors from mod 9 to mod 27, we compute $c_2(x)$ such that

$$\begin{aligned} 9c_2(x) &= (x+7)(x+2) - (x^2 + 27x + 176) \bmod 27 \\ &= (x^2 + 9x + 14) - (x^2 + 14) \bmod 27 \end{aligned}$$

that is,

$$c_2(x) = x$$

Now,

$$\begin{aligned} q_2(x) &= 2, \\ a_2(x) &= 2 \\ b_2(x) &= 2 \end{aligned}$$

We compute

$$\begin{aligned} v_3(x) &= (x+7) - (9)(2) \bmod 27 \\ &= x + 16 \\ w_3(x) &= (x+2) - (9)(2) \bmod 27 \\ &= x + 11 \end{aligned}$$

Now each of the coefficients in the factors of $u(x)$ is less than 27, so we have reconstructed them. We can check by division in case we are unsure, but we have in any case found that $u(x) = x^2 + 27x + 176 = (x+16)(x+11)$ over $\mathbf{Z}$.

Such "lifting" of factorizations by means of the Hensel Lemma replaces Garner's algorithm, and by a suitable generalization to multivariate coefficients, interpolation. Detailed analysis of this algorithm was first done in David Yun's MIT Ph.D. thesis: The Hensel Lemma in Algebraic Manipulation. Empirical tests demonstrate its speed over interpolation when the number of non-zero coefficients is small compared to the worst-case dense situation. If the GCD is sparse, this so-called EZ GCD "extended Zassenhaus" algorithm appears to dominate the earlier polynomial remainder sequence or modular GCD algorithms in terms of typical speed.

There are other approaches, even within the Hensel construction framework. An important practical variation by Paul Wang, named EEZGCD, provides better performance for sparse multivariate GCD computations by preserving sparseness in intermediate expressions, lifting the results one variable at a time. (Not discussed further in these notes)

An alternative technique first developed by Zippel in his 1979 MIT Ph.D. thesis: Probabilistic Algorithms for Sparse Polynomials, is rather complicated in practice, but is based on a reasonably simple idea: We start in the same way as a Hensel GCD algorithm. We are given an image of some multivariate polynomial GCD calculation in some univariate image domain. We now *assume* that any coefficients that are zero in a factor in the simple domain are zero even after lifting. That is, if some factor is (say, mod some large prime p and with y and z removed by the substitution of values for those indeterminates) $3x^4 + 2x^2 + 1$, then not only are the coefficients of the x^3 and x terms zero in this image, but they are zero *in the final answer*. There are three issues:

1. Making such assumptions, can we interpolate *faster* to get the complete expression of the non-zero coefficients? (Yes, quite a bit faster sometimes: because the cost depends on the number of non-zero terms, not the degree, this can be a big win.)

2. How likely is this assumption to be true? (Probabilistically speaking, the chance that a random substitution into a polynomial will come out zero is quite small; that several random substitutions will be zero is much smaller. A rather large literature on this question has developed since 1979.)

3. In case the assumption does not hold, can one repair the algorithm so that it gets the right answer (yes, Zippel's thesis showed how), and is not degraded in asymptotic running time. (well, it obviously takes longer, but Zippel has improved the repair steps so that it is still extremely probable to get the answer; in some hypothetical worst case it has been proved to be polynomial-time. (More details in lecture slides)

The final word in GCDs?

Finally, the Heuristic GCD of the Maple system is worthy of consideration. Although it clearly cannot work well for problems in too many variables because, as we indicated in class, the numbers you substitute in make the work grow exponentially as a function of the number of variables, it works so well (especially in Maple) on the vast majority of small- to medium- sized problems that it makes sense to try it first. Its speed relies on the assumption that modern computer algebra systems have relatively efficiently-coded arbitrary-precision arithmetic packages. This is especially true with respect to Maple, where algorithms not implemented in the system kernel are interpreted; moving the GCD algorithm from the interpreted library to the machine-coded kernel would be one way of speeding it up, but at the expense of enlarging the kernel. The alternative of mapping the GCD algorithm into computations largely in the kernel *already* seems especially beneficial.

The basic idea is that if you want to determine the coefficients of a polynomial $p(x) = c_0 + c_1x + \cdots + c_kx^k$ and you have a bound on the size of c_i you need not have $k+1$ different values of $p(x_i)$ for interpolation. As we saw in assignment 1, you can reconstruct p from only a single *sufficiently large* value $p(B)$. This *radix interpolation* idea was used on your first problem set. There is a Maple command `genpoly` that solves this problem, if have maple available and wish to experiment. To compute the polynomial GCD(f, g), GCDHeu computes a suitable bound B (heuristically chosen – the value is not a true bound which in practice is uncomfortably large and unlikely to be achieved, but a smaller estimated bound), evaluates $f(B)$ and $g(B)$, computes the GCD of these two integers, say $h(B)$ where one hopes that $h(x)$ is the GCD. It then uses radix interpolation as you used in assignment 1 to reconstruct $h(B)$. If such a construction is possible, and if h divides both f and g, then we have the GCD. By using evaluation to eliminate $n-1$ of the n variables in a problem, this can be extended to a general algorithm, suitable at least for modest-sized problems. One advantage of this program is its relative simplicity to implement, and the fact that for small inputs it is fast.

Review on "modern" polynomial GCDs

The dense modular GCD algorithm finds the GCD of two polynomials by evaluating the polynomials at a number of points, performing a modular GCD calculation at each point and then interpolating to obtain the result. If the GCD of two polynomials is a polynomial in n

variables of degree d each, then the number of evaluations and interpolations performed by the Modular algorithms is $(d+1)^{(n-1)}$.

Hence, if the GCD of two polynomials were $x_1^{100} + x_2^{100} + \ldots + x_{10}^{100}$, the dense modular algorithm might take months to compute it on existing machines. The best performance of the Modular algorithm occurs when the original polynomials are univariate or when the GCD is small, say 1, since this case involves few evaluations and univariate modular GCD calculations (perhaps only one!).

Zippel's sparse modular algorithm should do much better on cases such as this, where the GCD is of high degree yet sparse. Comparisons of this sparse modular algorithm over variations of the Hensel-based algorithm are less clear: it appears difficult to predict whether, in any particular case whether some Hensel-based algorithm, especially the variation called EEZGCD, is better.

GCDHeu serves well enough for many uses, and certainly for the typically small demonstrations that might occur in sales literature, or even benchmarks. Actually, for small to medium problems, our tests suggest the superiority of a more old-fashioned subresultant PRS (polynomial remainder sequence). GCDHEU is, however, especially well suited to Maple's implementation strategy.

In combination with some clever heuristics (including avoiding GCDs by maintaining factors throughout manipulations), the choice of "the best" GCD may not be critical for a large majority of computer algebra system duties. When the going gets tough, some of the algorithms shine compared to others; which shines depends on the nature of the *answer* (sparse/dense small/large) rather than the *inputs.* One rarely has the advantage of knowing the characteristics of the answer before beginning, and so the choice of algorithm is difficult.

What do systems use?

Macsyma has quite a collection of algorithms, including all but the GCDHeu algorithm. For small problems, simple checks are possible which suggest the use of very simple "low overhead" algorithms, including PRS methods. The data-structure complications of setting up these advanced algorithms (copying over polynomials reduced mod p, keeping extra evaluation) are then avoided.

After some simple checking, Maple uses the GCDHeu algorithm unless it cannot get a reasonable value for B, in which case it uses a Hensel algorithm (presumably one can look at the algorithm on-line).

In other computer algebra systems, including Reduce and Derive, variations of the polynomial remainder sequence, usually the subresultant GCD are used. Mathematica says it "usually uses modular algorithms, including Zippel's sparse interpolation algorithm, but in some cases uses subresultant PRS." Trying to cover all bases, I guess.

I have not found any specification of the algorithms used by MuPad or other general computer algebra systems. I suspect that they use a subresultant PRS as a reasonable compromise between simplicity of implementation and performance on a general class of inputs.

References

1. Knuth, D.E., *The Art of Computer Programming, vol.2; Seminumerical Algorithms*, Addison Wesley, Reading, Massachusetts 1969.

2. Moses, J. and Yun, D., "The EZ GCD Algorithm," *Proceedings of The 1973 ACM National Conference.*

3. Miola, A. and Yun, D., "Computational Aspects of Hensel-type Univariate Polynomial GCD Algorithms,"*Proceedings of Eurosam '74.*

4. Yun, D."A p-Adic Division With Remainder Algorithm," *SIGSAM Bulletin*, vol. 8, no. 4, 1974.

5. Zippel, R., *Effective Polynomial Computation*, Kluwer Scientific, 1993.