AD-A091 675 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 12/1
RESEARCH IN ALGEBRAIC MANIPULATION.(U)
1980 J MOSES F49620-79-C-0200
UNCLASSIFIED AFOSR-TR-80-1118 NL

1 OF 1
AD A
091675

END
DATE
FILMED
12-80
DTIC

AFOSR-TR- 80-1118

(13)

LEVEL II

AD A091675

MASS. INSTITUTE OF TECHNOLOGY
LABORATORY FOR COMPUTER SCIENCE
FINAL REPORT
TO THE US AIR FORCE
FOR RESEARCH IN ALGEBRAIC MANIPULATION
1979-1980

CONTRACT NO. [redacted]
F49620-79-C-0200

JOEL MOSES
PRINCIPAL INVESTIGATOR

80 11 06 012

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19 REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 18 AFOSR TR-80-1118	2. GOVT ACCESSION NO. AD-A091 675	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) RESEARCH IN ALGEBRAIC MANIPULATION		5. TYPE OF REPORT & PERIOD COVERED Final 1979-1980
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Joel Moses		8. CONTRACT OR GRANT NUMBER(s) F49620-79-C-0200
9. PERFORMING ORGANIZATION NAME AND ADDRESS Massachusetts Institute of Technology Laboratory for Computer Science 545 Technology Sq., Rm. 514 Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102E, 2304/A4
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research NM Bolling AFB, Washington, DC 20332		12. REPORT DATE 1979-1980
		13. NUMBER OF PAGES 79
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Algebraic manipulation, algebraic algorithms, Greatest Common Divisor Algorithm, probabilistic algorithms, MACSYMA

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes a new probabilistic algorithm for computing Greatest Divisors in polynominal time. The algorithm solves a basic problem inherent in all prior approaches that led to exponential space and time requirements.

80 11 06 012

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT TO THE US AIR FORCE ON RESEARCH ON ALGEBRAIC MANIPULATION 1979-1980

Our research this past year concentrated on probabilistic techniques for algebraic algorithms, such as the Greatest Common Division Algorithm. These probabilistic algorithms are developed in Richard Zippel's doctoral thesis, a revised version of which is enclosed. These algorithms are in some cases exponentially better than all other known algorithms and are as close to the best algorithms one can expect for a large class of practical problems so as to essentially cause work in this area to cease.

Research on algebraic alorithms (GCD, factorization of polynomials, resultants, determinants, solutions of equations) have been going on for twenty years. The first approaches were relatively straight-forward approaches that, in the case of the GCD algorithms, were variants of Euclid's algorithm. These are dependent on divisions of polynomials or pseudo divisions and had the unfortunate property of increasing the size of intermediate results by exponential orders. These unbearable effects of the straight-forward approaches were first combated with a class of algorithms called modular algorithms and popularized by G. Collins. The modular algorithms are optimal when the polynomials are dense (and thus have an exponential number of terms in multivariate problems). They, unfortunately, turn out to assume that all problems are dense and hence are exponentially slow in practical cases (i.e., multivariate and sparse).

The major attack on the modular approaches was the use of Hensel's lemma by Moses and Yun. Hensel's lemma is the algebraic variant of Newton's method. This approach, dubbed the EZGCD algorithm in the case of GCD, worked fine in many cases but failed when the derivative became 0. Whereas Newton's method only slows down at such a point, the use of Hensel's lemma forced us to expand the problem exponentially. This problem, called the "bad zero" problem, was the one solved by Zippel's probabilistic approach.

Zippel shows that the derivative goes to zero at an infinite number of points, but the probability that a randomly chosen point is such a "bad" point can be made as low as one pleases. Zippel also generalizes Hensel's lemma so that one can start it at any point (the EZGCD algorithm always started at 0). Zippel's thesis shows that the overhead for his probabilistic GCD algorithm compares favorably with other approaches and that indeed it solves the exponential problem of EZGCD. It, of course, compares very favorably with the older approaches.

Since the completion of the thesis in September, 1979, much of the work has gone into implementing and using the algorithms in MACSYMA. In practical uses, the probabilistic aspect of the algorithm has not ever been a problem. The algorithm can check when it runs into difficulty (i.e., when the randomly chosen point is indeed a zero of the derivative) and can try another point, but it has not had to do so yet.

We have also used similar techniques in other algebraic problems. For example, R. Gosper had a set of three 10th-degree polynomials to solve. Because Gosper was only interested in rational number solutions, Zippel was able to use a Hensel method and obtained hundreds of rational solutions. In my opinion, no other known approach to the problem would have succeeded on present day computers.

Probabilistic Algorithms for Sparse Polynomials

by

Richard Eliot Zippel

Contents

Chapter I

Introduction

Scientists and engineers have used algebraic manipulation systems with significant success in many computational problems. There are numerous symbolic computations in general relativity, high energy physics and celestial mechanics that have been successfully completed using algebraic manipulation systems; yet, they would have taken years to do by hand.

Unfortunately, algebraic systems do have their drawbacks. Because they deal largely with exact quantities and not approximations like floating point numbers, the expressions that are dealt with become larger with almost every arithmetic operation. Consider the following sum:

$$\sum_{i=1}^{10} \frac{(-1)^i}{i^2} \approx \frac{\pi^2}{12} \approx .8225.$$

With the terms given, only the first two digits will give a meaningful approximations to $\frac{\pi^2}{12}$. Using floating point arithmetic, we might compute with four digits. This would give .8180. But if the sum is computed with exact rational numbers we would have $\frac{5194387}{6350400}$.

This illustrates the fundamental tradeoff between precision and the size of the expressions used in a computation. Because algebraic manipulation systems insist on exactness, they are always faced with large expression growth, some of which may be unnecessary. The unnecessary growth can often be fatal. For a number of problems, of which greatest common divisor calculation is the best known, the straightforward approach leads to intermediate expressions that are much larger than the final answer. Behavior of this sort is called *intermediate expression swell.*

The intermediate expression swell problem is most prominent when the problem to be solved involves a large number of variables. Assume a polynomial, P, involves v variables and each appears with degree at most d. We will call P *dense* if nearly all possible monomials are present; that is, P contains almost $(d+1)^v$ terms. If only a small portion of the number of possible terms is present then P is said to be *sparse*. Assume the expressions with which a computation begins are sparse. After a number of arithmetic operations the intermediate expressions will become more dense. This is known as *fill in*. As the steps of the computation proceed, the operations with the intermediate expressions will become more expensive. If the final answer is relatively small, possibly because it is sparse for some deep reason, then the intermediate computations can require exponential time in the size of the answer.

Over the past fifteen years, two fundamental approaches have been advanced to help contain intermediate expression swell in certain cases. In the late sixties, Brown and Collins and his students investigated algorithms based on modular arithmetic and modular homomorphisms [Bro71]. The resulting algorithms did not use large amounts of storage, but for multivariate problems with small, sparse answers the modular algorithms could require exceedingly large amounts of time.

Zassenhaus [Zas69], Wang and Rothschild [Wan75, Wan78], Musser [Mus75], Moses and Yun [Mos73, Yun74, Yun76] have investigated an alternate approach that makes use of a computational formulation of Hensel's lemma. This approach not only minimizes the storage required in the intermediate computations, but also runs in polynomial time in the size of the answer for most problems. Unfortunately there exist problems that require both exponential time and space when this method is used. Thus in the worst case, both the modular and Hensel-based algorithms require exponential time to compute the answer.

The main accomplishment of this thesis is to introduce a new probabilistic idea that allows us to modify both the modular and Hensel approaches in a manner that greatly improves the behavior of these algorithms. These modified algorithms use an internal randomization process, so the amount of time they require on any particular problem may vary from trial to trial. However, their average running time for any particular problem is polynomial. We say that these algorithms run in *probabilistic polynomial time*.

The two classical approaches to problems in algebraic manipulation utilize some of the structure inherent in multivariate polynomial problems. By replacing some of the variables by integers, the multivariate problem is reduced to a univariate problem that is much easier to solve. It is then observed that the univariate solution is very closely related to the original multivariate solution. For many problems the univariate solution is the *image* of the multivariate solution with the variables replaced. This univariate solution is then *lifted* to a bivariate solution, and then to a trivariate solution and so on. When all the variables have been recovered we should be left with the desired multivariate solution.

This version of the modular algorithm does not take advantage of any information about the "shape" of the polynomials that might be determined at the stages of the lifting process. The Hensel algorithm is generally implemented in a manner that goes from a univariate solution to the full multivariate solution in one step. Though this is very efficient in many cases, there are classes of problems for which the basic technique breaks down. (Basically because the Jacobian of a system of equations closely related the problem becomes zero. This is called the *bad zero problem.*) In this case the problem can be modified to avoid the bad zero problem, but only by making the modified answer much denser in most cases.

The fundamental idea contained in this thesis is best expressed in terms of an interpolation algorithm. Assume that we are interpolating a sequence of values of a sparse bivariate polynomial, $P(X, Y)$. This is normally done by determining a number univariate polynomials, $P(X, y_i)$, and interpolating their coefficients to compute $P(X, Y)$. Our algorithm computes the first univariate polynomial by the usual interpolation scheme, but then assumes the other univariate polynomials have the same structure. For instance, if the first univariate polynomial is $X^5 + 1$, then we assume that the other univariate polynomials have only an X^5 and constant term and that the coefficients of the X^4, X^3, X^2 and X terms are zero. If this assumption is correct then we will only need to lift the coefficients of X^5 and of the constant term, thus decreasing the number of terms that need to be computed by 67%. An assumption of this sort is utilized as each variable is introduced into the answer. This exponentially decreases the number of terms that are computed and gives us an algorithm that is polynomial in the size of the answer.

1. Probabilistic Algorithms

Probabilistic techniques have been applied to difficult problems before. By difficult we mean problems whose best known algorithm requires exponential time. The NP-hard problems [Gar79] certainly fall into this class, as do the problems of polynomial factorization and computation of greatest common divisors. Weide [Wei78] divided probabilistic algorithms into three classes in his thesis. In the first class we have those algorithms that give an approximate answer to all problems (usually in polynomial time). The graph algorithms of Graham [Gra66] and Karp [Kar76] fall into this class. The primality testing algorithms of Solovay and Strassen [Sol77] and Rabin [Rab76] give an exact answer most of the time. Algorithms in this class can often be modified to always return an exact answer, but then will occasionally require exponential time. The final class of algorithms yields a good approximate answer most of the time.

The algorithms presented in this thesis, and those derived from the ideas given here will fall into the second class if it is possible to verify that an answer is correct. Both the GCD and polynomial factorization problems lie in this class. Some problems, like the computation of determinants and resultants, have solutions that are difficult to verify. Our algorithms for these problems fall into a subcategory of Weide's third class. These algorithms return the correct answer most of the time, but there is a chance the answer will be incorrect.

The algorithms presented in this thesis are probably best viewed as algorithms that produce candidate solutions which have a probability ϵ of being incorrect. It is possible to compute the candidate solutions in polynomial time both in terms ϵ and the size of the candidate. It is possible to make ϵ as small as desired. In practice, ϵ can be small enough that only one candidate need be computed.

2. Factoring Polynomials and Computing GCDs

Factoring integers has been a favorite pastime of human and electronic computers for some time. With the advent of symbolic algebra systems in the sixties, it was not surprising that interest would arise in factoring polynomials. The ability to factor polynomials has proven to be an exceedingly useful tool in other problems. Of course, this ability can make solving a polynomial equation much easier and thus lies at the heart of most packages to solve equations and systems of equations. Factoring polynomials also is part of most algorithms for computing integrals of rational and algebraic functions.

Algebraic number theory computations abound with factoring problems. The determination of the structure of Galois groups, the computation of the degree of an algebraic number over a field and the investigations of class groups are just a few of the problems that require factoring of polynomials.

Many users of algebraic manipulation systems have discovered that factoring programs can be among their most powerful simplifiers. The factorization of a result will often yield a more succinct expression and more insight into its physical interpretation than the expanded representation that many algorithms return.

The first algorithm for determining the factors of a polynomial over the integers was invented by von Schubert in 1793 and rediscovered by Kronecker in the late nineteenth century. This algorithm is now probably best known for its highly exponential behavior. The truly practical algorithms that have been developed in the seventies are based on Berlekamp's algorithm [Ber69] for factoring univariate polynomials over a finite field and Zassenhaus' version of Hensel's lemma [Zas69] for lifting a factorization over a finite field to one over the integers and then to a multivariate factorization.

As was mentioned earlier, there are certain classes of problems for which the Hensel approach takes an extremely long time. These problems are generically called "bad zero" problems. Characteristically these problems require time that is exponential in the number of variables.

Recently, Paul Wang [Wan78] introduced some ideas on how to reduce the impact of the bad zero problems on algorithms using Hensel's lemma. The heuristics he uses yield a very significant reduction of the exponential cost of some bad zero type problems, but they do not completely eliminate the exponential behavior. The precise dependence of the exponential behavior of Wang's algorithms has not yet been determined, due in part to the fact that they are exceedingly complex algorithms.

The techniques presented in this thesis are a direct result of Wang's ideas when applied to our formulation of Hensel's lemma. A significant advantage of our framework for the Hensel lemma is that Wang's ideas, when incorporated, yield a demonstrably polynomial time algorithm. Actually, the work for this thesis began by trying to show that Wang's ideas could be applied to the formulation of Hensel's lemma presented here.

Though probably not as exciting a problem as factoring polynomials, computing GCD's is certainly at least as important. GCD's are implicitly used in computations to keep rational functions reduced to lowest terms. Unlike the situation with factoring polynomials where there is essentially only one realistic approach, there are four approaches that have been implemented and are used in algebraic manipulation systems. Each works quite well for some class of problems. The subresultant GCD algorithm [Col67] is rather difficult to analyze, but it seems to perform well when the GCD is a large factor of the original polynomials. When the two polynomials have a GCD of 1, the subresultant algorithm will often require unreasonably large amounts of time and space. Recently Hearn [Hea79] introduced a new algorithm along the same lines as the subresultant algorithm, which seems to behave somewhat better than the subresultant algorithm but still exhibits exponential behavior. The modular algorithm [Bro71] does not require a large amount of space but it still runs in exponential time in terms of the number of variables. This is reasonable for completely dense polynomials. The Hensel version of the GCD algorithm [Mos73], which is called the EZGCD algorithm, has the same problem with bad zeroes that plagues the Hensel factorization algorithms.

Our improvement to the Hensel factoring algorithm also yields a polynomial time algorithm for GCD computations. However, we have also been able to apply our techniques to the modular algorithm to produce a probabilistic procedure that computes polynomial GCDs in time that is polynomial in the size of the answer. This new modular algorithm, which was discovered in discussions with Barry Trager, is actually significantly simpler than the Hensel algorithm and illustrates the key ideas in our approach more clearly. Thus we discuss the sparse modular algorithm first, even though our version of the Hensel algorithm was discovered first.

The following table lists the times required by a number of algorithms to compute the GCD of the second set of polynomials listed in the appendix. It is intended to give an indication of the possible performance of the algorithms presented in this thesis. In section VII.4 we discuss this example more fully. The times are listed in seconds and the asterisks indicate that MACSYMA ran out of storage.

v	EZ	Modular	Reduced	EEZ	Sparse Mod
2	.614	.481	.710	.108	.312
3	2.938	6.092	1.876	2.908	1.074
4	14.935	64.963	*	5.906	1.413
5	*	282.373	*	9.075	2.394
6	*	*	*	60.417	4.153
7	*	*	*	*	5.145
8	*	*	*	*	4.953
9	*	*	*	*	8.699
10	*	*	*	*	8.811

3. Determinants, Resultants and Linear Equations

The computation of determinants of large sparse matrices is one of the more difficult problems in computing. Using numerical methods, "large" means matrices with thousands of rows and columns. In algebraic manipulation a matrix with 10 to 20 rows is quite large. This is one of the problems that quite graphically points out the difference between numerical and symbolic calculations. The determinant of a general 20×20 symbolic matrix contains $20! \approx 2.5 \times 10^{18}$ terms far more than can be handled by any existing computer or even any being designed now. On the other hand, the determinant of any numerical matrix is a single number.

One result that points to the difficulty in computing determinants has been recently discovered by Papadimitriou [Pap79]. He has shown that for certain $n \times n$ matrices involving $\sqrt{n}$ variables, the computation of the coefficient of a single monomial in the determinant is NP-hard.

If the only problem being considered was computing the entire determinant then determinant calculations would be quite uninteresting. On occasion, however, something is known about the structure of the determinant beforehand. It would be very useful if some use can be made of this information to speed the computation.

Commonly, in the course of a problem, a system of equations is constructed. Of all the variables, only one or two may actually be interesting. All the others were merely introduced to set up the system of equations. Using numerical methods, the computation of the values of these *auxiliary variables* is not really a devastating problem. Though their computation may require additional time, the space required is only linear in their number.

In symbolic problems each of the auxiliary solutions can be very large. If there is any way to eliminate the computation of the auxiliary variables it will be very worthwhile. Until now, the only technique available for avoiding these auxiliary computations was the modular interpolation algorithm. But this algorithm does not take advantage of any sparsity in the answer. In a later section of this thesis we will discuss the first algorithms for solving these sorts of problems that do take advantage of the sparsity of the answer.

The computation of polynomial resultants and inversion of a matrix containing polynomial entries can both be considered to be special cases of determinant calculations. They also seem to have the same problems that plague determinants. It is very difficult to check an answer with all of these problems. Thus the algorithms we present can possibly give an erroneous result, but the probability that an answer is erroneous can be set arbitrarily low before the computation begins.

4. A Roadmap

This thesis is divided into two essentially independent pieces. Part I is devoted to the study of the modular algorithm, its derivatives and applications. Part II is concerned with the algorithms based on Hensel's lemma. In each portion the core of the results is contained in chapters that discuss the use of sparsity in each of these approaches.

The main example that is used to illustrate both the modular and Hensel algorithms is the computation of the GCD of two polynomials. This algorithm is used frequently in algebraic manipulation and is implicit in virtually all computations involving rational functions. In fact, during the sixties before Collins' and Brown's [Bro71] work on GCD computations, the dominant cost in rational function operations was the GCD calculations used to ensure that rational functions were reduced to lowest terms.

Because of their importance we begin Part I with a review of the what are now the "classic" GCD algorithms. This review, which is contained in chapter II, begins with a discussion of the principles underlying Euclid's algorithm for polynomials and the various improvements of it, which culminated in Collins' subresultant algorithm [Col71]. The chapter concludes its survey of the algorithms of the sixties with Collins' and Brown's version of the modular GCD algorithm.

In chapter III we present our sparse version of the modular algorithm. This chapter presents the fundamental idea advanced in this thesis. In chapter III we also give a relatively detailed analysis of the behavior of the probabilistic sparse modular algorithm.

In chapter IV we apply the ideas of the previous chapters to a number of problems. Of particular interest is the sparse resultant algorithm. This is the first algorithm of any form that computes the resultant of two polynomials in time that is polynomial in the size of the answer. We also indicate how the sparse ideas may be applied to more involved and complex problems in algebraic manipulation.

In the second portion of this thesis we illustrate how the basic idea of a sparse lifting can be applied to Hensel's lemma. Before this can be done we need to cast Hensel's lemma in a framework that is somewhat different from that presented by other researchers in algebraic manipulation. To fix the notation and make this a bit more self-contained, the basic ideas that are taken from mathematics are presented in chapter V. Those who are familiar with the results of valuation theory can skip this chapter.

Chapter VI gives a detailed discussion of Hensel's lemma. We present our "new" framework and trace its development from Newton's work through Hensel's. We then present the "old" version that was popularized by Zassenhaus [Zas69], Wang and Rothschild [Wan75], Musser [Mus75] and Moses and Yun [Mos73, Yun75, Yun 76]. This version is shown to be equivalent to our version. The results of chapters V and VI are quite old and we make no claim to originality here.

Chapter VII shows how the ideas on sparsity that were applied to the modular algorithm in chapter III can be applied to Hensel's lemma. We also show how a trick due to Wang may be used to increase the speed of the algorithm even further. The final section of this chapter gives an analysis of the sparse Hensel algorithm. The Hensel algorithm, when applied to the computation of polynomial GCDs, can be analyzed in much the same manner as the sparse modular algorithm.

Finally, chapter VIII presents our conclusions and some points that deserve further study.

Throughout this thesis the following conventions are observed with respect to cross references. Any reference to a section in different chapter will be of the form V.2.3. The other chapter's number is given as a capital Roman numeral and is followed by the section number and then the subsection number. References to sections in the same chapter omit the chapter number. Bibliographical references consist of the first three letters of the first author's last name and the last two digits of the year of publication.

Chapter II

Classical GCD Algorithms

All practical algorithms for computing the greatest common divisor of two polynomials are based on Euclid's algorithm (perhaps the first usage of Fermat's principle of infinite descent [Har68]). Assume we are given two positive integers $p_1 > p_2$ whose greatest common divisor (GCD) we wish to calculate. We can determine two integers, q_1 and p_3, that satisfy

$$p_1 = q_1 p_2 + p_3 \qquad p_2 > p_3 > 0$$

by integer division. It is clear that any integer that divides p_1 and p_2 also divides p_3. Furthermore, any integer that divides p_2 and p_3 must divide p_1 so the GCD of p_2 and p_3 is also the GCD of p_1 and p_2. Assume $p_3 \neq 0$. We now compute a new quotient and remainder

$$p_2 = q_2 p_3 + p_4 \qquad p_3 > p_4 > 0.$$

This can be continued to give a *remainder sequence*

$$p_1 > p_2 > p_3 > \cdots > p_n > p_{n+1} = 0,$$

thus p_n must be the GCD of p_1 and p_2.

In this chapter we will examine how this algorithm may be extended to compute the GCD of two polynomials. The remainder sequence of Euclid's algorithm then becomes a sequence of polynomials called a *polynomial remainder sequence*, or PRS. Again the GCD will be the last non-zero term in the sequence. For polynomials, the term before the GCD, p_{n-1}, may be much larger than necessary. The main purpose of the algorithms in the first two sections of this chapter is to reduce the size of the elements of the PRS.

In the final section, we discuss the modular GCD algorithm due to Brown and Collins [Bro71]. This algorithm is the first of the GCD algorithms that deviates radically from the classical approach.

1. Euclid's Algorithm

We will use the following conventions and terminology throughout this thesis. Capital letters will be used to represent symbolic literals. Lower case letters will be used to represent unknowns, integers and, on occasion, other expressions. Let

$$F(X) = f_0X^n + f_1X^{n-1} + \cdots + f_n$$

be a polynomial in X. The *degree* of F is n. This is denoted by $\deg F = n$. If $f_0 = 1$ then F is a *monic* polynomial. The greatest common divisor of $f_0, \ldots, f_n$ is the *content* of F. If the content of a polynomial is 1, then it is said to be *primitive*. Given the polynomial F we can compute its *content* by computing the GCD of f_0 and f_1, call it g_1; then computing the GCD of g_1 and f_2 *and repeating*. The content will then be g_n. The *primitive part* of F is $f(X)/g_n$, which is a primitive polynomial.

Let $F_1(X)$ and $F_2(X)$ be polynomials over a field. $F_1(X)$ can be divided by $F_2(X)$ to obtain a quotient $q_1(X)$ and a remainder $F_3(X)$ that satisfy

$$F_1(X) = q_1(X) \cdot F_2(X) + F_3(X) \qquad \deg(f_3) < \deg(f_2). \tag{1}$$

Just as with integers, the GCD of $F_2(X)$ and $F_3(X)$ must also be the GCD of $F_1(X)$ and $F_2(X)$. This process can be repeated with $F_2(X)$ and $F_3(X)$ to obtain a quotient $q_2(X)$ and a remainder $F_4(X)$.

$$\begin{gathered} F_2(X) = q_2(X) \cdot F_3(X) + F_4(X) \\ \vdots \\ F_{n-3}(X) = q_{n-3}(X) \cdot F_{n-2}(X) + F_{n-1}(X) \\ F_{n-2}(X) = q_{n-2}(X) \cdot F_{n-1}(X) + F_n(X) \end{gathered}$$

The degrees of the $F_i(X)$ are decreasing, hence, for some i, the degree of $F_i(X)$ must be zero. If $F_i(X)$ is non-zero then the next remainder in the sequence, $F_{i+1}(X)$, is zero. Without loss of generality we may assume that $F_{n+1}(X) = 0$. The GCD of $F_1(X)$ and $F_2(X)$ will be $F_n(X)$. Notice that the degree of the polynomial is being used in place of the magnitude metric used for integers.

As an example consider calculating the greatest common divisor of

$$\begin{aligned} F_1(X) &= X^8 + X^6 - 3X^4 - 3X^3 + 8X^2 + 2X - 5 \qquad \text{and} \\ F_2(X) &= 3X^6 + 5X^4 - 4X^2 - 9X + 21. \end{aligned}$$

(This is the traditional example used to illustrate polynomial GCD algorithms first used by Knuth [Knu69].) Using the Euclidean algorithm we obtain the following sequence of polynomials. We write only the coefficients of $F_i(X)$.

$$\begin{aligned} F_1 &= 1, 0, 1, 0, -3, -3, 8, 2, -5 \\ F_2 &= 3, 0, 5, 0, -4, -9, 21 \\ F_3 &= -\tfrac{5}{9}, 0, \tfrac{1}{9}, 0, -\tfrac{1}{3} \\ F_4 &= -\tfrac{117}{25}, -9, -\tfrac{441}{25} \\ F_5 &= \tfrac{233150}{19773}, -\tfrac{102500}{6591} \\ F_6 &= \tfrac{1288744821}{543589225} \end{aligned}$$

Since the last non-zero term in the *polynomial remainder sequence* is a constant, the GCD of these two polynomials is 1.

Intermediate expression swell usually becomes more severe as the number of variables increases. In the following PRS we have used bivariate polynomials of smaller degree than the previous example, yet the polynomials that result are still quite large.

$$\begin{aligned}
F_1 &= X^4 + X^3 - W \\
F_2 &= X^3 + 2X^2 + 3WX - W + 1 \\
F_3 &= (-3W + 2)X^2 + (4W - 1)X - 2W + 1 \\
F_4 &= \frac{27W^3 - 2W^2 - 11W + 3}{9W^2 - 12W + 4} X + \frac{9W^3 - W^2 + 4W + 1}{9W^2 - 12W + 4} \\
F_5 &= \frac{-729W^7 - 738W^6 - 471W^5 + 725W^4 + 81W^3 - 162W^2 + 68W - 8}{729W^6 - 108W^5 - 590W^4 + 206W^3 + 109W^2 - 66W + 9}
\end{aligned}$$

Had we used three, four or more variables the elements of the PRS would have filled many pages.

In practice it is not reasonable to use rational coefficients in the PRS. The number of GCD's required to keep these coefficients reduced to lowest terms is just too great, and not reducing them leads to horrendous expression growth. If we relax (1) slightly we can obtain a "pseudo-quotient" and "pseudo-remainder" that will always have integral coefficients. Let $F_1(X)$ and $F_2(X)$ be primitive polynomials over a ring R of degrees n_1 and n_2 respectively. (A polynomial is *primitive* if its coefficients are pairwise relatively prime.) Denote the leading coefficient of $F_i(X)$ by f_i. The pseudo-remainder $r(X)$ and pseudo-quotient $q(X)$ satisfy

$$f_2^{n_1 - n_2 + 1} F_1(X) = F_2(X) \cdot q(X) + r(X). \qquad \deg(r) < n_2$$

Using $r(X)$ as $F_3(X)$, the next term in the PRS leads to rather severe expression swell as we see:

$$\begin{aligned}
F_1 &= 1, 0, 1, 0, -3, -3, 8, 2, -5 \\
F_2 &= 3, 0, 5, 0, -4, -9, 21 \\
F_3 &= -15, 0, 3, 0, -9 \\
F_4 &= 15795, 30375, -59535 \\
F_5 &= 1254542875143750, -1654608338437500 \\
F_6 &= 12593338795500743100931141992187500
\end{aligned}$$

This growth is clearly unacceptable. Most of these polynomials are not primitive. If we remove their content at each step we get the following PRS:

$$\begin{aligned}
F_1 &= 1, 0, 1, 0, -3, -3, 8, 2, -5 \\
F_2 &= 3, 0, 5, 0, -4, -9, 21 \\
F_3 &= -5, 0, 1, 0, -3 \\
F_4 &= 13, 25, -49 \\
F_5 &= 4663, -6150 \\
F_6 &= 1
\end{aligned}$$

This is known as the *primitive PRS* and the algorithm that uses it is called the *primitive* GCD algorithm. It is as good as can be done with respect to the growth of the terms in the PRS;

unfortunately the amount of time required to ensure that each term in the PRS is primitive is exceedingly great.

The terms of a PRS always satisfy the following relationship

$$f_{i+1}^{n_i - n_{i+1}+1} F_i(X) = F_{i+1}(X) \cdot q(X) + \beta_i F_{i+2}(X).$$

For the Euclidean PRS, $\beta_i = 1$. We will denote the pseudo-remainder of $F_i(x)$ and $F_{i+1}(x)$ by $\mathrm{prem}(F_i, F_{i+1})$ and the content of F_i by $\mathrm{cont}(F_i)$. Then the primitive PRS uses $\beta_i = \mathrm{cont}(\mathrm{prem}(F_i, F_{i+1}))$.

To avoid the GCD's needed by the primitive PRS, Collins introduced the *reduced PRS* [Col67, Bro71]. This algorithm uses 1 for β_1 and $f_{i+1}^{n_i - n_{i+1}+1}$ for β_i. For the same problem, the coefficients of the reduced PRS are

$$\begin{aligned}
F_1 &= 1, 0, 1, 0, -3, -3, 8, 2, -5 \\
F_2 &= 3, 0, 5, 0, -4, -9, 21 \\
F_3 &= -15, 0, 3, 0, -9 \\
F_4 &= 585, 1125, -2205 \\
F_5 &= -1885150, 24907500 \\
F_6 &= 527933700
\end{aligned}$$

Through better than the Euclidean PRS, the coefficients of the reduced PRS can still grow exponentially. In particular, if the two polynomials are relatively prime, the cost involved in the PRS can be prohibitive. This is probably the most common case for the GCD's involved in rational function computations.

2. Subresultant PRS

The best of the GCD algorithms that use a full polynomial remainder sequence is Collins' subresultant algorithm. For sufficiently sparse polynomials, Brown's analysis [Bro78] indicates that even the subresultant algorithm will require exponential time to compute the GCD. Again the way of decreasing the growth of the terms in the PRS is to pick a better value for β_i. The proof that the β_i chosen actually leads to a valid PRS is not easy. The interested reader should examine [Bro78], which is the definitive reference for the subresultant algorithm.

Let $\delta_i = n_i - n_{i+1}$. For the *subresultant* PRS, β_i can be chosen as follows

$$\begin{aligned}
\beta_3 &= (-1)^{\delta_1+1} \\
\beta_i &= (-1)^{\delta_{i-2}+1} f_{i-2} h_{i-2}^{\delta_{i-2}}, \qquad i = 4, \ldots, k+1;
\end{aligned}$$

where

$$\begin{aligned}
h_2 &= f_2^{\delta_1} \\
h_i &= f_i^{\delta_{i-1}} h_{i-1}^{1-\delta_{i-1}}, \qquad i = 3, \ldots, k.
\end{aligned}$$

The subresultant PRS for our standard example is

$$\begin{aligned}
F_1 &= 1, 0, 1, 0, -3, -3, 8, 2, -5 \\
F_2 &= 3, 0, 5, 0, -4, -9, 21 \\
F_3 &= 15, 0, -3, 0, 9 \\
F_4 &= 65, 125, -245 \\
F_5 &= 9326, -12300 \\
F_6 &= 260708
\end{aligned}$$

If δ_i is 1 then the PRS step that divides F_i and F_{i+1} to produce F_{i+2} is called *normal*. Otherwise, the step is said to be *abnormal*. At each normal step of a PRS the size of the coefficients tends to grow linearly. Abnormal steps lead to faster growth.

The subresultant PRS algorithm does an admirable job of minimizing intermediate expression swell in the computation of the polynomial remainder sequence. If we are interested in the GCD of the two polynomials, only the last term of the PRS will be of interest. If the PRS is relatively short then it will not have a chance to grow too much. In this case the GCD of the two polynomials will tend to be a large factor of one of the two polynomials. However, if the GCD of two polynomials is small the PRS involved will tend to rather long. In this case the swell involved in the PRS can be extremely costly.

3. The Modular GCD algorithm

In this section we discuss a GCD algorithm that does not try to reduce intermediate expression swell by controlling the content introduced in the remainders of the PRS. Instead this algorithm maps the problem into a univariate polynomial ring over a field in which GCDs can be computed simply, easily and without intermediate expression swell. After doing a number of GCDs in that ring, it is possible to interpolate the results to compute the original, polynomial answer. Thus there is no need to compute the large terms of the PRS.

This idea was developed by Brown and Collins [Bro71]. With some hints, the modular algorithm was independently discovered by Knuth [Knu69]. This led to an interesting method of controlling the intermediate expression swell for GCDs, but at a cost in the time required by the algorithm.

Brown and Collins observed that computing univariate GCD's over a finite field led to no swell since the coefficients could not grow beyond the size of the modulus. Thus they convert a multivariate GCD problem to a number of univariate GCD problems modulo some prime. The answers to these problems are then interpolated to produce the true, multivariate GCD using the Chinese remainder algorithm or Newtonian or Lagrangian interpolation.

This section is divided into two parts. In the first we discuss the Chinese remainder algorithm and some of its features. This is the key to doing the interpolation. In the second we indicate how it is applied to produce the modular GCD algorithm.

3.1. Modular Arithmetic

In the next section we will be using modular arithmetic rather heavily. In this section we will review some of the basic principles involved when performing computations with modular arithmetic. We will write

$$x \equiv y \pmod{m}$$

if x and y differ by a multiple of m, and say that x and y are *equivalent modulo m* if this is the case. It is conventional to pick an element from each equivalence class of the rational integers modulo m and do the arithmetic with these representatives. Two sets of representatives are commonly used. On most current computers it is somewhat easier to perform modular arithmetic with representatives from the set $\{0, 1, 2, \ldots, m-1\}$. We call this the *non-negative representation* of the integers modulo m [Knu69]. The *balanced representation* of the integers modulo m uses the set

$$\{\lceil -(m-1)/2 \rceil, \ldots, -1, 0, 1, \ldots, \lceil (m-1)/2 \rceil\}$$

(The ceiling operations are needed because m may be even.)

Regardless of the representation chosen, any integer k will be equivalent to only one element of the set of representatives. Thus representative r is called the *residue* of k modulo m. We will also say *that k is congruent to r modulo m.*

Arithmetic with these equivalence classes modulo m is called *modular arithmetic*; m is called the *modulus*. Modular addition, subtraction and multiplication are all performed by combining the two operands using integer arithmetic and then reducing the result to the appropriate range, usually by a remainder computation. Raising a number to a power can be performed by repeated squaring using modular multiplication. This is especially effective since the numbers used are all about the same size as the modulus.

If p is an integer modulo m we can sometimes compute $1/p$ modulo m. Denote $1/p$ modulo m by u. Then

$$pu \equiv 1 \pmod{m},$$

and pu and 1 differ by a multiple of m. That is

$$pu - 1 = mv \qquad \text{or} \qquad pu - mv = 1.$$

If p and m have a common divisor $d > 1$, then d would divide $pu - mv$ and would thus have to divide 1. Since this is impossible there cannot be solutions to $pu - mv = 1$ and thus p cannot possess an inverse modulo m.

If p and m are relatively prime then it is not too hard to demonstrate that $pu - mv = 1$ must have solutions. To see this, compute the GCD of p and m using the Euclidean algorithm. For uniformity we will let f_0 and f_1 denote m and p respectively. We will also assume that p is in the non-negative representation. Then

$$\begin{aligned} f_0 &= q_1 f_1 + f_2 \\ f_1 &= q_1 f_2 + f_3 \\ &\vdots \\ f_{\ell-2} &= q_{\ell-2} f_{\ell-1} + f_\ell \\ f_{\ell-1} &= q_{\ell-1} f_\ell + 1 \end{aligned}$$

The final remainder must be 1 since f_0 and f_1 (p and m) are relatively prime. It is not hard to show that if

$$a_i f_0 = b_i f_1 + f_i$$

then

$$\begin{aligned} a_{i+1} &= q_{i-1} a_i + a_{i-1} \\ b_{i+1} &= q_{i-1} b_i + b_{i-1} \end{aligned}$$

Since $f_{\ell+1} = 1$ we have $u = b_{\ell+1}$. Thus, if p and m are relatively prime, p has an inverse modulo m, and in which case p is said to be a unit. More generally a unit is some element of a ring that has a multiplicative inverse. Furthermore, we have developed a technique for solving the equation

$$af - bg = 1$$

for integers a and b when f and g are relatively prime. This *equation figures prominently in the* Hensel algorithms discussed later.

3.2. The Chinese Remainder Algorithm

We now begin outlining the Chinese remainder algorithm for integers. Let $p_1, \ldots, p_k$ be relatively prime rational integers. Denote the product of the p_i by P. Assume we know that some integer w satisfies the system of equations

$$w \equiv m_i \pmod{p_i} \qquad i = 1, \ldots, k. \tag{2}$$

We do not assume that the p_i are primes. However, if they are not pairwise relatively prime then the *system of equations* may not have a solution. Note that this is when P is not square-free, i.e. when there is no integer Q such that Q^2 divides P. Regardless of whether P is square-free, if there is a solution it will not be unique. If a multiple of P is added to any solution of (2) then the result will also be a solution. If P is square-free, then there all the solutions of (2) will lie in exactly one equivalence class of the integers modulo P.

The Chinese remainder algorithm provides a means of determining an integer $\hat{w}$ which satisfies these modular equations, provided P is square-free. If there are only two equations then the appropriate value for $\hat{w}$ is apparent, $\hat{w} = m_1 + p_1(m_2 - m_1)p_1^{-1}$, where the inverse of p_1 is computed modulo p_2. Thus we can replace (2) by the slightly smaller system

$$\begin{aligned} w &\equiv m_1 + p_1(m_2 - m_1)p_1^{-1} \pmod{p_1 p_2}, \\ w &\equiv m_3 \pmod{p_3}, \\ &\vdots \\ w &\equiv m_k \pmod{p_k}. \end{aligned} \tag{3}$$

We can now repeat this process with the first two equations of (3) since $p_1 p_2$ and p_3 are relatively prime. The following *specification*, which follows the model of [Knu69], makes this algorithm precise. We will use this format throughout this thesis.

Algorithm C takes as input a set of pairwise relatively prime rational integers $p_1, \ldots, p_k$ and a set of integers $m_1, \ldots, m_k$. It returns an integer $m < p_1 p_2 \cdots p_k$ such that $m \equiv m_i \pmod{p_i}$ for $1 \leq i \leq k$.

```
C1. [Initialize] Set $q \leftarrow p_1$, $m \leftarrow m_1$.
C2. [Loop] For $i = 2, \ldots, k$ do step C3.
    C3. [Determine new $m$] Set $q' \leftarrow q^{-1} \pmod{p_i}$, $m \leftarrow m + (m - m_i) \cdot q \cdot q' \pmod{qp_i}$ and
        $q \leftarrow qp_i$.
C4. [End] Return $m$.
```

We are now going to extend this algorithm to an interpolation algorithm for polynomials. In order to do this we will rephrase this version of the Chinese remainder algorithm in terms of modern algebra. In general, the problem we are to solve is: Given that w satisfies

$$\begin{aligned} w &\equiv m_1 \pmod{p} \\ w &\equiv m_2 \pmod{q} \end{aligned}$$

where p and q are relatively prime, we are to find m_3 such that

$$w \equiv m_3 \pmod{pq}.$$

To generalize this algorithm we will say that p, q, m_1 m_2 and w are now all polynomials. The expression $w \equiv m_1 \pmod{p}$, just as in the integral case, means that $w - m_1$ is divisible by p. If p is a linear polynomial, $(X - a_1)$, then the representation of the residue classes are rational numbers. We can compute the residue of $f(X)$ modulo $(X - a_1)$ by division:

$$f(X) = (X - a_1)q(X) + b.$$

b must be of degree less than 1, so it must be a constant. Replacing X by a_1 we see that $b = f(a_1)$. So the residue of $f(X)$ modulo $(X - a_1)$ is $f(a_1)$.

More generally, let F be a field and $p(X)$ a polynomial over F of degree d. We write this as $p(X) \in F[X]$. The *canonical* representatives of the residue classes modulo $p(X)$ are the elements of $F[X]$ of degree less than d. Arithmetic modulo $p(X)$ is very similar to arithmetic modulo an integer. Addition is a bit simpler since the sum of the two polynomials of degree less than d will have degree less than d. Thus no normalization step is necessary. The product of the residue classes represented by $f(X)$ and $g(X)$ is the remainder of $f(X)g(X)$ when divided by $p(X)$. The same repeated squaring algorithm used for modular arithmetic can be applied here.

The elements that are zero are multiples of $p(X)$. Denote this set by I. These elements form an additive group. Furthermore, if a is in $F[X]$ and $b \in I$ then $ab \in I$. These two facts mean that I is an *ideal*. This particular ideal is denoted by $(p(X))$. This notation indicates that all the elements of I are multiples of $p(X)$. The residue class ring is denoted by $F[X]/(p(X))$. Similarly, the integers modulo p are denoted by $\mathbb{Z}/(p)$. (The ideal (p) is sometimes also written as $p\mathbb{Z}$ for clarity.)

Thus far we have placed no restriction on $p(X)$. If F is a ring instead of a polynomial then the multiplication algorithm will be valid if and only if $p(X)$ is monic.

If $p(X)$ is reducible then not all elements of $F[X]/(p(x))$ will be invertible, even if F is a field. To see this recall the algorithm used in section 3.1 for computing the inverse of an integer modulo m. That algorithm was based on the fact that we could solve

$$ap - bm = 1.$$

Thus $a \pmod m$ was the inverse of p. In the polynomial case the same algorithm holds. Assume we wish to compute the inverse of $f(X)$ modulo $p(X)$. We can again try to solve the equation

$$a(X)f(X) - b(X)p(X) = 1.$$

As before, this equation can have solutions if and only if $f(X)$ and $p(X)$ are relatively prime. Assume for now that they are not relatively prime. If $p(X)$ is irreducible then $f(X)$ is a multiple of $p(X)$ and thus $f(X) \equiv 0 \pmod{p(X)}$. Conversely, if every $f(X)$ which is not coprime to $p(X)$ is a multiple of $p(X)$ then $p(X)$ is irreducible.

Thus if each residue class modulo $p(X)$ is to have an inverse $p(X)$ must be irreducible. Furthermore, it is not hard to carry through the algorithm of section 3.1 for polynomials. In order for the final remainder to be 1, it will be necessary to use the Euclidean PRS. This means that F must be a field. If these two conditions are met then $F[X]/(p(X))$ will be a field. This equivalent to saying that $(p(X))$ is a maximal ideal.

We can now finally return to the Chinese remainder algorithm. It is clear that just changing the words "relatively prime integers" to "relatively prime polynomials" in algorithm C will make this algorithm valid for the polynomial case. Let $f(X)$ be the polynomial that is to be computed. In its most common usage, the Chinese remainder algorithm uses linear polynomials $(X - p_i)$ for the generators of the ideal and thus the residues are just the values of $f(X)$ at $X = p_i$. Furthermore, the computation $q^{-1} \pmod{p_i}$ used in step C3 is now equivalent to $q(p_i)^{-1}$ since F is a field. With these observations we are lead to the following algorithm.

Algorithm D. Given two sets of rational integers $\{p_1, \dots, p_k\}$ and $\{m_1, \dots, m_k\}$, returns a polynomial $f(x)$ such that $f(p_i) = m_i$ for $1 \le i \le k$.

```
D1. [Initialize] Set $f(x) \leftarrow m_1$, $q(x) \leftarrow (x - p_1)$.
D2. [Loop] For $i = 2, \dots, k$ do step D3.
D3. [Determine new $f$] Set $f(x) \leftarrow f(x) + q(p_i)^{-1}q(x)(m_i - f(p_i))$ and $q(x) \leftarrow (x - p_i)q(x)$.
D4. [End] Return $f(x)$.
```

It is important to note that even if the goal polynomial for algorithm D is very sparse the intermediate results can be completely dense. The following example should demonstrate both this and the use of the algorithm. All computations are performed in the field $Z/163Z$.

p_i	m_i	$f(x)$
-12	70	70
14	-75	$-62x - 22$
24	75	$61x^2 - 21x - 1$
33	72	$-28x^3 - 26x^2 + 79x + 62$
51	55	$-53x^4 + 2x^3 - 20x^2 - 23x - 2$
-1	0	$x^5 + 1$

Since $q(x)$ is a product of linear polynomials, it is almost certain to always be dense.

3.3. Detailed Description of the GCD Algorithm

As before we wish to compute the GCD, $G(X)$, of the primitive polynomials $F_1(X)$ and $F_2(X)$. Let B be a number which bounds the maximum of the absolute value of the coefficients of G. (This bound may be computed from a theorem of Gelfond [Gel60], but in many implementations [Bro71] the maximum of the absolute values of the coefficients of $F_1(X)$ and $F_2(X)$ is used instead.) Let $d = \deg(G)$. Pick k random prime rational integers $p_1, \dots, p_k$, such that $p_1 \cdots p_k > 2B$. Usually p_i are chosen to be less than a machine word in size for efficiency. By using the balanced representation we can get negative coefficients also. Now the coefficients of $F_1(X)$ and $F_2(X)$ are reduced modulo each of the p_i successively and the GCD is computed over Z/p_iZ. Denote these GCDs by $G_i(X)$. When computing polynomial remainder sequences over a field, the elements will always be monic. Therefore, the GCD which we compute modulo p_i will be monic. For simplicity let us assume that both $F_1(X)$ and $F_2(X)$ are monic. This restriction is removed later.

Since both $F_1(X)$ and $F_2(X)$ are assumed to be monic, their GCD must also be monic. Therefore when reduced modulo p_i, the GCD's degree will not be decreased. But the degree of $G_i(X)$ can be larger than the degree of the GCD. If d is differs from $\deg(G_i)$ then we say that p_i was an *unlucky prime*. If a p_i turns out to be unlucky it is discarded and a new one is chosen. (In practice a prime is considered to be unlucky if $\deg(G_i) > \deg(G_{i-1})$.)

We now have k polynomials, each of degree d which are images of G modulo p_i. We can now apply algorithm C to the vector of constant terms of G_i, and then to the linear terms, and so on. The interpolated values are the coefficients of G. A slightly more efficient arrangement of this algorithm is expressed in Algorithm M, which does not require that $F_1(x)$ and $F_2(x)$ be monic.

Algorithm M takes two primitive univariate polynomials F_1 and F_2 over the integers and a bound B on the size the coefficients of their GCD as inputs and returns their GCD.

```
M1. [Initialize] Set $c$ to the GCD of the leading coefficients of $F_1(X)$ and $F_2(X)$. Pick a prime
    $m$ which does not divide $c$ and set $G(X)$ to the GCD of $F_1(X)$ and $F_2(X)$ computed modulo
    $m$. Set $d \leftarrow \deg(G)$ If $d$ is zero, immediately return 1 as the GCD.
M2. [New prime] Pick a prime $p$ which does not divide $c$. Set $H(X)$ to the GCD of $F_1(X)$ and
    $F_2(X)$ computed modulo $p$.
    M3. [Impose leading coefficient] Set $H(X) \leftarrow cX^d + (H(X) - X^d) \cdot c^{-1}$ where all the
        coefficient arithmetic is done modulo $p$.
    M4. [Unlucky prime?] If $\deg(H) > d$ then $p$ was an unlucky prime. Go to M2. If $\deg(H) < d$
        then $m$ was an unlucky prime, start all over again. Go to M1.
    M5. [Loop over coefficients] Set $\hat{G} \leftarrow 0$ For $0 \le i \le k$ do step M6.
        M6. [Interpolate] Use algorithm C on the coefficients of $X^i$ in $G$ and $H$ with moduli
            $m$ and $p$. Let the interpolated result be $w$. Set $\hat{G} \leftarrow \hat{G} + wX^i$.
    M7. [Reset the world] Set $G \leftarrow \hat{G}$, $m \leftarrow pm$. If $m > 2B$ then return $G$ else go to M2.
```

This algorithm can easily be extended recursively to handle multivariate polynomials. A detailed description is contained in [Bro71]. Here we will present a rather cursory overview of the algorithm. The main purpose of this overview is to point out the source of the modular algorithm's exponential behavior. After this presentation we will demonstrate another, more inefficient version of this algorithm that exhibits the source of this behavior quite clearly.

Assume that we are again trying to compute the GCD of the two primitive, monic polynomials $F_1(X, X_2, \ldots, X_v)$ and $F_2(X, X_2, \ldots, X_v)$. We will assume that the coefficients of F_1 and F_2 lie in a field. (If the coefficients are rational integers, then they can be reduced modulo some large prime and the integer coefficients restored by a slight variation of algorithm M.) The GCD of F_1 and F_2 is the polynomial $G(X, X_2, \ldots, X_v)$. Assume we know that none of the X_i appears in G with degree greater than d. The multivariate algorithm begins by picking values for $X_2 = a_{20}, \ldots, X_v = a_{v0}$, and substitutes these values into F_1 and F_2. The GCD of the resulting polynomials is readily computed. The value of the constant term of the GCD must be the constant term of G evaluated at $X_2 = a_{20}, \ldots, X_v = a_{v0}$ and similarly for the other coefficients. The algorithm now picks d new values for X_2, $a_{21}, \ldots, a_{2d}$ and computes the GCD of F_1 and F_2 with $X_2 = a_{21}, \ldots, a_{2d}$ and $X_3 = a_{30}, \ldots, X_v = a_{v0}$. That is X_2 is the only variable whose value changes. The coefficients of the $d + 1$ univariate polynomials may be interpolated using algorithm D to compute the bivariate GCD of F_1 and F_2 at $X_3 = a_{30}, \ldots, X_v = a_{v0}$, $G(X, X_2, a_{30}, \ldots, a_{v0})$.

Now a new value is chosen for X_3, a_{31}; $d + 1$ values are chosen for X_2, and the interpolation procedure of the previous paragraph is repeated. This yields $G(X, X_2, a_{31}, \ldots, a_{v0})$. Using $d - 1$ bivariate polynomials, $G(X, X_2, \cdot, \ldots, a_{v0})$. Now the coefficients of these polynomials can be interpolated using algorithm D to compute $G(X, X_2, X_3, \ldots, a_{v0})$. The computation of this polynomial required $d + 1$ bivariate polynomials be computed. Each bivariate polynomials required $d + 1$ univariate GCDs so thus far we have used $(d + 1)^2$ univariate GCDs. To compute the four variable GCD we will need $d + 1$ trivariate GCDs and will thus require $(d + 1)^3$ univariate GCDs, and the whole problem will require $(d + 1)^{v-1}$ GCDs. This is illustrated by figure 1 for $v = 4$.

Each level of the tree has a branching factor of $d + 1$ so one can't help but have exponential behavior. In chapter III we will show how to slash the whole structure of the tree so that it does not have an exponential number of leaves.

Now we will present another way of looking at the GCD problem that yields some insight into the necessity for the exponential growth. Basically we will demonstrate how the modular

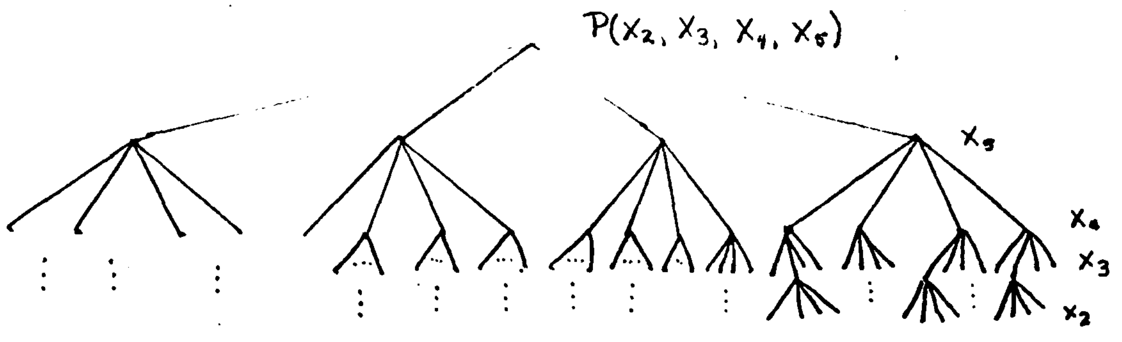

Figure 1.

algorithm can be shown to be determining a number of coefficients. Since there are an exponential number of these unknown coefficients this algorithm must be exponential. This version also leads more naturally to our new "sparse" modular algorithm.

Again $F_1(Z, Z_2, \ldots, Z_v)$ and $F_2(Z, Z_2, \ldots, Z_v)$ are primitive, monic polynomials and we want to compute their GCD, $G(Z, Z_2, \ldots, Z_v)$. In place of the bound B that we had for the size of the coefficients in the univariate case, we need a bound on the degree of $Z_2, \ldots, Z_v$ in $G(Z, Z_2, \ldots, Z_v)$. Assume that no Z_i appears with degree greater than d in G. Given this we can write

$$G(Z, Z_2, \ldots, Z_v) = \sum_{i_1=0}^{d} \cdots \sum_{i_v=0}^{d} c_{i_1 \cdots i_v} Z^{i_1} Z_2^{i_2} \cdots Z_v^{i_v}$$

and we wish determine the $c_{i_1 \cdots i_v}$.

A very simple way to determine the coefficients of G is to set up a system of linear equations for which the unknown coefficients will be a solution. There are $(d+1)^v$ of these unknown coefficients, $c_{i_1 \cdots i_v}$. Pick integers for $a_{12}, \ldots, a_{1v}$ and compute

$$G_1 = \gcd(F_1(Z, a_{12}, \ldots, a_{1v}), F_2(Z, a_{12}, \ldots, a_{1v})).$$

If we write

$$G_i(Z) = g_{i,0} Z^d + g_{i,1} Z^{d-1} + \cdots + g_{i,d}$$

then equating the coefficients of Z produces the following $d+1$ relations among the $c_{i_1 \cdots i_v}$:

$$g_{1,j} = \sum_{i_2=0}^{d} \cdots \sum_{i_v=0}^{d} c_{j, i_2 \cdots i_v} a_{0,2}^{i_2} \cdots a_{0,v}^{i_v} \qquad j = 0, \ldots, d \tag{4}$$

Repeating this process by computing $(d+1)^{v-1} - 1$ more univariate GCD's, $G_2, \ldots, G_{(d+1)^{v-1}}$ enough linear equations are produced to determine the unknown coefficients of $G(Z, Z_2, \ldots, Z_v)$.

This algorithm will use $O(c^v)$ steps to compute the GCD of two polynomials involving v variables. If the GCD is dense then this is the amount of time you would expect a GCD algorithm to take since there will be $(d+1)^v$ terms in the GCD, and thus $(d+1)^v$ coefficients that need to be determined. In the case of sparse polynomials most of the coefficients will be zero. The modular algorithm will still set up the large system of equations (4) and will still take exponential time to solve the system.

The major advantage of the modular algorithm is that it never introduces intermediate results that are larger than the answer. But its running time is also independent of the structure of the answer. This can prove to be very unfortunate in the many common uses of the algorithm where the expressions involved are very sparse.

Chapter III

Sparse Modular Algorithm

All modular algorithms have basically the same form. The value of a polynomial is computed at a number of points and these values are interpolated to produce the original polynomial. We will call this polynomial the *goal polynomial* of the algorithm. The goal polynomial is assumed to involve v variables. Each variable appears to no higher degree than d in the goal polynomial. The goal polynomial will be denoted by $P(X_1, \ldots, X_v)$.

There are $(d+1)^v$ independent coefficients in P. If "many" of them are zero then P is said to be sparse, otherwise it is dense. We will denote the number of non-zero coefficients by t. If P is completely dense it will have $t_d = (d+1)^v$ non-zero coefficients. For sparse polynomials $t \ll (d+1)^v$.

Any interpolation algorithm that computes P must determine $(d+1)^v$ coefficients. If less than t_d points are examined there will be an infinite number of possible polynomials that satisfy the appropriate degree bounds and that take on the correct values at the test points. To see this notice that each point that is examined yields a single linear constraint on the coefficients of P

$$g_0 = \sum_{i_1=0}^{d} \cdots \sum_{i_v=0}^{d} c_{i_1 \cdots i_v} a_1^{i_1} \cdots a_v^{i_v}$$

If there are fewer linear equations than coefficients to be determined, the system of equations will be underdetermined.

Just looking at $(d+1)^v$ points requires time exponential in the number of variables. Thus any modular interpolation algorithm that can guarantee that its results are correct must require an exponential amount of time to compute P.

This chapter presents a probabilistic algorithm that computes P in polynomial time. Since this algorithm is probabilistic, there is some chance that the polynomial it returns as P is not the desired one. However, the chance of being in error can be made arbitrarily small by increasing the size of the set from which the test points are chosen.

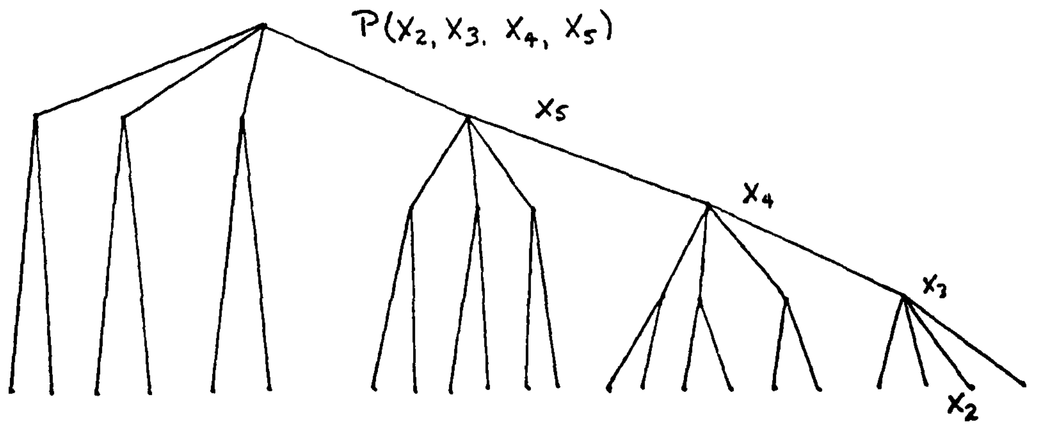

Figure 2.

1. Overview of the Sparse Modular Algorithm

The fundamental idea behind the sparse modular algorithm can be seen by comparing figure 1 of section II.3.3 with the figure of this section. As before, a sequence of points is chosen, $X_2 = a_{20}, \ldots, X_v = a_{v0}$ and the univariate GCD is computed. This is done for $d+1$ different values of X_2. The resulting univariate GCDs are interpolated using algorithm D of chapter II to give a bivariate polynomial. As before, a new value of X_3 is chosen and a new bivariate GCD is computed, but *this time, instead of using* $d+1$ univariate GCDs we use some of the structure determined by the first bivariate GCD. We assume that the coefficients of X_2^i that were non-zero in the first GCD will be the *only* non-zero coefficients. (In the figure we have assumed that there are only two non-zero coefficients in the answer.) Now a different, slightly less efficient interpolation algorithm is used, but fewer GCDs were necessary to compute the base points for the interpolation. As this proceeds, the recursive nature of the algorithm disappears along with the exponential behavior.

The modular algorithm presented in section II.4 solved one large system of equations to determine the coefficients of the goal polynomial. The sparse algorithm begins by choosing a starting point for the interpolation, $(x_{10}, \ldots, x_{v0})$. It then produces the sequence of polynomials,

$$\begin{aligned} P_1 &= P(X_1, x_{20}, \ldots, x_{v0}), \\ P_2 &= P(X_1, X_2, x_{30}, \ldots, x_{v0}), \\ &\vdots \qquad \vdots \\ P_v &= P(X_1, X_2, \ldots, X_v). \end{aligned}$$

The process of computing P_i from P_{i-1} is called a *stage* of the lifting process.

P_1 is a univariate polynomial in X_1. The coefficient of X_1^k in P is a polynomial $f_k(X_2, \ldots, X_v)$. If P is sufficiently sparse there will be certain powers of X_1 that do not appear in P_1. (If $t < d+1$ then X_1 cannot appear in P to all $d+1$ powers.) Assume that the X_1^k term is one of those terms that is not present. There are two possible reasons why X_1^k does not appear in P_1. Either f_k is identically zero or $f_k(x_{20}, \ldots, x_{v0})$ is equal to zero. The probability that $f_k(x_{20}, \ldots, x_{v0})$ is zero when f_k is not identically zero is extremely small if the starting point $(x_{20}, \ldots, x_{v0})$ is chosen at random. Thus the probability that f_k is identically zero is quite large. The key idea in this algorithm is to assume that X_1^k does not appear in P; i.e., f_k is identically zero. Thus it is assumed that the coefficient of every monomial involving X_1^k is known, and that it is zero.

This information is used to construct P_2. Now the same reasoning can be applied to each monomial in X_1 and X_2 that does not appear in P_2. Since there are at most t terms in any of the P_i, almost all of the terms will be zero, and the number of coefficients that will need to be computed is small.

A few comments on the chances of f_k not being identically zero are in order. This probability depends only on the range from which $x_{20} \ldots, x_{v0}$ is chosen. For example, assume $v = 2$. Then f_k involves only one variable and f_k has only a finite number of zeroes, r. If we pick x_{20} randomly from a set with B elements the probability that we have picked a zero of f_k is, at most, r/B. This argument is generalized in chapter IV.

For simplicity we will walk through the algorithm when P is a polynomial in 3 variables, $P(X, Y, Z)$. As usual, we assume that P is a sparse polynomial with t terms ($t \ll (d+1)^v$, $v = 3$). (Whenever we say "pick x_i randomly" we will mean pick an integer x_i randomly from a set $\mathcal{S}$ that has at least B distinct elements.[1]) Pick y_0 and z_0 randomly. These two numbers must be chosen randomly and are the only numbers that need to be chosen at random for this example.

We now pick $x_0, \ldots, x_d$ and examine the values of P at the points (x_i, y_0, z_0). These may be interpolated by the Lagrange interpolation formula to give a univariate polynomial in X, namely $P(X, y_0, z_0)$. (Actually, the Lagrange interpolation formula is probably more rightly attributed to Waring [War79].) So far nothing probabilistic has entered the algorithm.

We now assume that if X^k (for some k) has a coefficient of zero in $P(X, y_0, z_0)$ then it will have a zero coefficient in $P(X, Y, Z)$. Pick a y_1, not necessarily at random. From $P(X, y_0, z_0)$ we know that a number of the coefficients of $P(X, y_1, z_0)$ are zero. In particular, since the coefficient of X^k is zero in $P(X, y_0, z_0)$, we will assume that the coefficient of X^k in $P(X, y_1, z_0)$ is also zero. Thus the only coefficients of $P(X, y_1, z_0)$ that need to be determined are the those that are suspected to be non-zero, i.e. the coefficients of monomials that appeared in $P(X, y_0, z_0)$. There can be no more than t of these unknown coefficients, since there are only t terms in the answer. These coefficients can be determined by solving a system of linear equations. Only the values of $P(x_0, y_1, z_0), \ldots, P(x_t, y_1, z_0)$ will be needed to set up this system of equations.

This procedure may be repeated until we have determined $d+1$ polynomials $P(X, y_0, z_0), \ldots, P(X, y_d, z_0)$. Pick a monomial in X that appears in each of these polynomials. For simplicity we will assume that it is the linear term. The coefficient of X in $P(X, Y, z_0)$ is a polynomial in Y of degree at most d. Call this polynomial $f(Y)$. From the $d+1$ polynomials we have computed we can determine the values of $f(Y)$ at $y_0, \ldots, y_d$. Again using the usual interpolation methods we can determine $f(Y)$ from this information. Repeating this for the quadratic and higher terms of $P(X, y_i, z_0)$ we can determine $P(X, Y, z_0)$.

Now that we have $P(X, Y, z_0)$, it is only natural to to compute $P(X, Y, z_1)$ for a new z_1, which does not need to be chosen at random. This can be done in a manner almost identical with that used earlier. We know that the monomials that appear in $P(X, Y, z_1)$ will have non-zero coefficients in $P(X, Y, Z)$. Just as in the stage where we introduced the polynomials in Y, we assume that none of the $X^i Y^j$ monomials that appear in $P(X, Y, Z)$ are absent from $P(X, Y, z_0)$. There are at most t of these monomials. Picking a z_1, we see that there are at most t unknown coefficients to be determined in $P(X, Y, z_1)$. All the others are believed to be zero. Picking t new pairs of values $(x_1, y_1), \ldots, (x_t, y_t)$ and computing $P(x_i, y_j, z_1)$, we can set up a system of linear equations in the unknown coefficients. Solving this system, we have $P(X, Y, z_1)$; repeating this procedure we will finally determine $P(X, Y, z_d)$. Now we can determine the polynomials in Z

[1] The set $\mathcal{S}$ is usually chosen to be the set of integers in the interval $[0, B-1]$, but can be any set of integers with B elements. Throughout this section lower case symbols will denote integers chosen at random while uppercase symbols will be reserved for variables.

that are coefficients of each X^iY^j monomial by the usual interpolation schemes.

There are two essentially different types of interpolation schemes going on in this algorithm. The first time we try to generate a polynomial in X, it is not known what its structure is and thus the interpolation *is performed as if the polynomial were dense*. That is we used $d+1$ points to determine a polynomial of degree d. This we call a *dense* interpolation. Now that we have this *skeleton* from which to work, we can produce additional polynomials in X by solving systems of equations. These steps are called *sparse interpolations*. These coefficients are then combined via a dense interpolation to give polynomials in Y. The algorithm proceeds in this manner. The first polynomial produced involving a particular variable is done via a dense interpolation. The structure determined by the dense interpolation is then used to produce a skeleton for the polynomial. This skeleton is used as the basis for a series of sparse interpolations that are done to set up the points for a new variable.

2. General Formulation of Sparse Modular Algorithm

In this section we will present the precise form of the sparse modular algorithm so it may be applied to a number of examples. We will present these algorithms using the same syntax that we used in presenting the Chinese remainder algorithm in chapter II. Making these algorithms precise in this manner will also aid in the analysis of the algorithm.

The first algorithm we consider is used when a dense lifting is required; it is an extension of the Chinese remainder algorithm to polynomial rings. We have a polynomial $f(x)$ and two sets of rational integers $\{p_1, \ldots, p_k\}$ and $\{m_1, \ldots, m_k\}$ such that $f(p_i) = m_i$. We want to determine $f(x)$. Let $\underline{p}_i = (x - p_i)$ denote the principal ideal of $Q[x]$ generated by $x - p_i$. Since $f(x)$ is an element of the ring $Q[x]$, the m_i are the images of $f(x)$ under the map

$$\varphi_i : Q[x] \to Q[x]/(x - p_i) = Q.$$

So one way of expressing the relationship between the f, p_i and m_i is

$$f(x) = m_i \pmod{\underline{p}_i} \qquad i = 1, \ldots, k.$$

Thus using precisely the same arguments as were presented in section II.3.1 we can justify the following algorithm.

The sparse modular algorithm needs a data structure to indicate which terms are known to be zero. Since there are fewer terms that are likely to have nonzero coefficients than terms with zero coefficients, we will keep track of the nonzero terms. A monomial of the form $X_1^{e_1} \cdots X_v^{e_v}$ will be represented by the v-tuple $(e_1, \ldots, e_v)$. A *skeletal polynomial*, S, is understood to be a set v-tuples such that each element of S represents nonzero term in the goal polynomial.

Whenever a skeletal polynomial is produced, we will want to determine the coefficients that are determined. This will be done by solving a system of linear equations. To simplify the notation a bit we will adopt the following convention. Assume a skeletal polynomial S contains t terms. We will assume that each skeletal polynomial has associated with it t symbols that will represent the coefficients of the monomials given by S. Denote these symbols by $s_1, \ldots, s_t$ where the subscript, i, is associated with the exponent vector $(e_{i1}, \ldots, e_{iv})$. Then we define

$$S(a_1, \ldots, a_v) = s_1 a_1^{e_{11}} \cdots a_v^{e_{1v}} + s_2 a_1^{e_{21}} \cdots a_v^{e_{2v}} + \cdots + s_t a_1^{e_{t1}} \cdots a_v^{e_{tv}}.$$

The sparse modular algorithm can be specified as follows.

Algorithm S takes a set of variables $\{ X_1, \ldots, X_v \}$, a degree bound d, a function $F(X_1, \ldots, X_v)$ and a starting point $(a_1, \ldots, a_v)$ as arguments. It is assumed that the values F returns are the values of some polynomial of at most v variables and of degree at most d in each variable. The starting point is assumed to be a good starting point. The algorithm returns a polynomial $P(X_1, \ldots, X_v)$, where each variable occurs with degree no more than d and $P(b_1, \ldots, b_v) = F(b_1, \ldots, b_v)$ for all integers b_i.

```
S1. [Initialize] Set $S \leftarrow \{ (0) \}$ and $p_0 \leftarrow a_0$.
    S2. [Loop over variables] For $i = 1$ through $v$ do S3 through S8.
    S3. [Iterate $d$ times] For $j = 1$ through $d$ do S4 through S7.
        S4. [Initial linear equations] Pick $r_j$, set $L$ to the empty list, set $t$ to the length of $S$.
        S5. [Iterate $t$ times] For $k = 1$ through $t$ do S6.
        S6. [Set up linear equations] Pick an $(i-1)$-tuple $\Lambda_k$, and add the the linear equation
            $S(\Lambda_k) = F(\Lambda_k, r_j, a_{j+1}, \ldots, a_v)$ to $L$.
        S7. [Solve] Solve the system of equations $L$ and merge the solution with $S$ to produce
            a polynomial $p_j(X_1, \ldots, X_{i-1})$.
    S8. [Introduce $X_i$] For each monomial in $S$ pass the corresponding coefficients from
        $p_0, \ldots, p_d$ and $a_i, r_1, \ldots, r_j$ to algorithm D. This will produce $t$ polynomials in $X_i$ that
        can be merged with $S$. Set $p_0$ to this new polynomial and $S$ to its skeletal polynomial.
S9. [Done] Return $p_0$.
```

There is one point at which caution should be exercised in implementing this procedure. The first time through the i loop the linear equations that are set up will be trivial since there is only one unknown. There is a chance that the linear equations that are developed will not be independent. If this happens then it is necessary to execute step S6 until sufficiently many independent equations are produced.

Since we must solve a system of linear equations in this algorithm, it is usually best to perform the entire computation over some finite field as we mentioned in the remarks about algorithm D.

3. An Example

To illustrate the sparse modular algorithm we will compute the linear term of the GCD, over the integers, of the two polynomials

$$\begin{aligned}
F_1(X,Y,Z) = X^5 &+ (Y^2 + YZ + 1)X^4 + (2Y^3 - 7YZ^2 - 2)X^3 \\
&+ (2Y^5 + 2Y^4Z - 7Y^3Z^2 + 2Y^3 - 7Y^2Z^3 - 7YZ^2 + 3)X^2 \\
&+ (-4Y^3 + 3Y^2 + 14YZ^2 + 3YZ + 3)X - 6, \\
F_2(X,Y,Z) = X^6 &+ (2Y^3 - 7YZ^2 + Y - 6Z^3)X^4 + 6X^3 \\
&+ (2Y^4 - 12Y^3Z^3 - 7Y^2Z^2 + 42Z^5Y)X^2 \\
&+ (6Y^3 - 21YZ^2 + 3Y - 18Z^3)X + 9.
\end{aligned}$$

The basic technique will be use the sparse modular algorithm to compute the GCD over several finite fields. These GCDs are then interpolated to determine the correct GCD over the integers. The first problem is to compute a bound for the size of the coefficients of the GCD. Following Brown [Bro71] we will assume that the magnitude of the coefficients of the GCD of two polynomials is bounded by *the product of the absolute values of the largest coefficient* in each. Thus, in our example we assume that no coefficient in the GCD will be larger than 14×21. Since we do not know the sign of the coefficients, they must be determined modulo some number larger than $2 \times 14 \times 21 = 588$.

Since the maximum degree of Y in F and G is 5 and 3 respectively, it can appear to degree no higher than 3 in the GCD. Similarly Z cannot appear to any degree higher than 4 in the GCD. Since both F and G are monic polynomials H must be monic also. We need not worry about adjusting the leading coefficient.

In a real implementation the primes would probably be chosen to just fit in a word. For most computers we would then need only one prime. To demonstrate how these algorithms work in general we will use two smaller primes, 19 and 31.

We will only try to compute the coefficient of the linear term in X of the GCD; the other coefficients are computed similarly and should be computed in parallel. Thus the goal polynomial will involve two variables, Y and Z. We will denote the GCD by $G(X, Y, Z)$ and the goal polynomial by $P(Y, Z)$. The starting point is chosen at random. The integers involved might as well be less than the modulus chosen. Starting with 19 as the modulus we chose $Y = 12$ and $Z = 15$ as the starting point.

In the notation of algorithm S, the goal polynomial is $P(X_1, X_2)$, $X_1 = Y$, $X_2 = Z$ and the starting point is $a_1 = 12$ and $a_2 = 15$. With Z fixed at 15 we compute the GCD of F and G with Y varying over several values. Actually, only four are necessary.

Y	Z	$G(X,\cdot,\cdot)$
12	15	$X^3 + 3X + 3$
0	15	$X^3 + 3$
11	15	$X^3 + 5X + 3$
13	15	$X^3 - 7X + 3$
2	15	$X^4 + 5^3 + X^2 + 8X - 4$
5	15	$X^3 - 6X + 3$

From the first polynomial it is clear that the degree of the GCD is not more than 3. So the next-to-last polynomial can be thrown out since (2, 15) must be an "unlucky evaluation point" [Bro71].

From this evidence it seems likely that the quadratic term in H is zero. If the modulus were somewhat larger, so that the interpolation points could be chosen from a larger set, we would have significantly more confidence in this result. As we shall see, this problem was chosen so *that none of these guesses which might go awry actually do. It* also seems that the constant term is independent of Y. At any rate, it is easy to interpolate: it must be 3.

The only interesting term is the linear one. It is determined by interpolating the linear coefficients of H using algorithm D.

p_i	m_i	$P(Y,\cdot)$
0	0	0
11	5	$0+\frac{1}{11}\cdot Y\cdot 5$
13	-7	$-9Y^2+Y$
5	-6	$2Y^3+2Y$

Algorithm D results in $G(X,Y,15) = X^3 + (2Y^3+2Y)X + 3$. Nothing terribly exciting or new has happened yet. So far we have not had a chance to use the sparse modular algorithm. We now need to compute $G(X,Y,\cdot)$ for several other values of Z. This is done using the sparse algorithm. We first pick a new value of Z, $Z=7$. We need only to pick 2 values for Y at this stage since there are just 2 coefficients of the linear term to be determined. We have assumed that the coefficients of the Y^2X and Y^0X terms are zero. Picking a couple of values for Y and computing the univariate GCDs,

Y	Z	$G(X,\cdot,\cdot)$
4	7	X^3-9X+3
13	7	X^3-8X+3

We know that the goal polynomial (coefficient of the linear term in X) is of the form $c_1Y^3+c_2Y$, so we get the linear equations

$$7c_1+4c_2=-9$$
$$-7c_1+13c_2=-8$$

These equations can be easily solved, giving $c_1=2$ and $c_2=-1$. Note that this time only two univariate GCD's were necessary, while four were needed for the previous interpolation. This procedure must be repeated for three more values of Z. Each time a pair of linear equations must be solved. The results are

# GCDs	Z	$G(X,Y,\cdot)$
4	15	$X^3+(2Y^3+2)X+3$
2	7	$X^3+(2Y^3-Y)X+3$
2	13	$X^3+(2Y^3-5Y)X+3$
2	3	$X^3+(2Y^3-6Y)X+3$
2	13	$X^3+(2Y^3+8Y)X+3$

In the first column we have given the total number of additional univariate GCDs required to solve determine the particular polynomial. Interpolating these coefficients, we see that

$$G(X,Y,Z) = X^3 + (2Y^3 - 7YZ^2)X + 3.$$

Now we change the modulus. Picking two values of Y and Z, with the modulus set at 31 we get the following univariate GCDs.

$$G(X,22,19) = X^3 - 12X + 3$$
$$G(X,17,21) = X^3 + 3X + 3$$

Since the linear term of H is believed to be of the form $c_3Y^3 + c_4YZ^2$ we can use these two results to set up the following two linear equations:

$$15c_3 + 6c_4 = -12$$

$$15c_3 - 5c_4 = 3$$

These two equations can be solved to give $c_3 = 2$ and $c_4 = -7$. Thus

$$G(X,Y,Z) = X^3 + (2Y^3 - 7YZ^2)X + 3.$$

Clearly this must be the GCD. The total number of univariate GCDs which was needed was only 14. Had we used the old modular algorithm we would have required 40 univariate GCDs.

4. Analysis

The purpose of this section is to show that the time to compute one candidate solution to a problem using the sparse modular algorithm is polynomial in the size of the candidate and ϵ, where ϵ is a bound on the probability that the candidate is erroneous.

Denote the goal polynomial by $P(X_1, \ldots, X_v)$ and the starting point by $\vec{a} = (a_1, \ldots, a_v)$. The a_i are chosen at random from a set of B possible values. When ϵ is chosen to be very small, B will be quite large. Thus the a_i may be quite large and we must be somewhat careful when including the cost due to integer arithmetic. When using classical algorithms, the cost of multiplication and division is quadratic in the length of the integers while the cost for addition and subtraction is linear. To crudely take this into account we will count the number of arithmetic operations with integers and multiply that by $\log^2 B$.

Recall that it is the dense interpolation (using the Chinese remainder algorithm) that introduces new variables as the goal polynomial is built up. The sequence of polynomials that is produced is

$$P(X_1, a_2, \ldots, a_v), P(X_1, X_2, a_3, \ldots, a_v), \ldots, P(X_1, X_2, \ldots, X_v). \qquad (1)$$

Next there is a series of sparse interpolations that form the basis for the next dense interpolation.

These sparse interpolations use a skeletal polynomial as a guide to the structure of the system of linear equations must be solved. If the true goal polynomial does not conform to the structure indicated by the skeletal polynomial then the next dense iteration, in addition to being wrong, will almost certainly be dense and the world will come crashing down afterwards.

Thus the entire algorithm depends upon the accuracy of the skeletal polynomials. Since the skeletal polynomials are extracted from the structure of the polynomials in the sequence (1), it is important to know if $P(X_1, a_2, \ldots, a_v)$ has the wrong number of terms. Clearly it cannot have too many terms. If it has too few terms then the coefficient of some X_1^k in $P(X_1, \ldots, X_v)$ is zero at $\vec{a}$. Let F_1 be the product of the nonzero coefficients of X_1^k in P for $k = 1$ through d. If $\vec{a}$ is not a zero of F_1 then the first skeletal polynomial will be computed correctly.

Similarly if the coefficient of some monomial in X_1 and X_2 is zero at $\vec{a}$ the second skeletal polynomial will be erroneous. Define F_2 to be the product of the coefficients of nonzero monomials in X_1 and X_2 in the goal polynomial and define $F_3, \ldots, F_{v-1}$ similarly. If the starting point is not a zero of any of the F_i, then none of the skeletal polynomials will be erroneous.

The *auxiliary polynomial* for P is defined to be $F = F_1F_2 \cdots F_{v-1}$. F is a polynomial in $X_2, \ldots, X_v$. The key assumption used in the sparse modular algorithm is that the initial evaluation point is not a zero of this polynomial. As we shall see, in the sparse version of the Hensel algorithm the determinant of the Jacobian will play the part of the auxiliary polynomial. A point that is not a zero of F is called a *good point*. Since all bad points satisfy $F = 0$ they form a variety of codimension 1. Qualitatively, almost all points in affine $(v - 1)$-space are good. The following lemma makes this precise.

Lemma 1. *Let $f \in \mathbf{Z}[X_1, \ldots, X_v]$ and the degree of f in X_i be bounded by D. Let $N_v(B)$ be the number of zeroes of f, $(x_1, \ldots, x_v)$ such that $x_i \in \mathcal{I}$ (a set with B elements $B \gg D$). Then $N_v(B) \le B^v - (B-D)^v$.*

Proof: There are at most D values of X_v at which f is identically zero. So for any of the D values of X_v and any value for the other X_i, f is zero. This comes to DB^{v-1}. For all other $B-D$ values of X_v we have a polynomial in $v-1$ variables. The polynomial can have no more than $N_{v-1}(B)$ zeroes. Therefore,

$$N_v(B) \le DB^{v-1} + (B-D)N_{v-1}(B).$$

Let $N_v = (B-D)^{v-1} n_v$ then

$$n_v - n_{v-1} \le \frac{DB^{v-1}}{(B-D)^{v-1}}$$

For $v = 1$, $N_1 \le D$, so we have $n_1 \le D$. Summing the previous equation

$$n_v \le n_1 + \sum_{i=2}^{v} \frac{DB^{i-1}}{(B-D)^{i-1}}$$

$$\le D \sum_{i=0}^{v-1} \frac{DB^i}{(B-D)^i} \le D\left(\frac{\left(\frac{B}{B-D}\right)^v - 1}{\frac{B}{B-D} - 1}\right)$$

$$\le (B-D)\left(\frac{B^v}{(B-D)^v} - 1\right)$$

By the definition of n_v we have

$$N_v = (B-D)^{v-1} n_v \le (B-D)^v\left(\frac{B^v}{(B-D)^v} - 1\right).$$

The lemma follows immediately. ∎

This bound is actually attained by the dense polynomial

$$f(x_1, \ldots, x_v) = \prod_{i=1}^{D}(x_1 - i) \cdots \prod_{i=1}^{D}(x_v - i).$$

One would expect a much tighter bound to hold for sparse polynomials. To get an idea of how much better than this bound a typical polynomial can be consider the following. Let

$$f(\vec{X}) = f(X_1, \ldots, X_v) = c_1 m_1 + \cdots + c_t m_t$$

be a polynomial as in the lemma. We have used m_i to represent a monomial in the X_j, so f has t terms. Assume the monomials m_i are fixed for now. If $f(\vec{a}_1)$ is zero then we have a constraint on the c_i

$$c_1 m_1(\vec{a}_1) + c_2 m_2(\vec{a}_1) + \cdots + c_t m_t(\vec{a}_1) = 0.$$

Without being clever in our choice of points, we can only force $t-1$ points to be zeroes of f. In fact it is known [Lan62] that a curve of genus greater than 0 has only a finite number of integral points. Thus for $v = 2$, in general, if $D > 2$ there is a constant which bounds $N_v(B)$. The proof of this statement is extremely difficult. For higher numbers of variables much less is known.

Assume now that the goal polynomial has v variables and that no variable appears to degree higher than d. Each of the F_i is the product of at most t terms and each term is of degree at most d. There are $v-1$ of these polynomials, so the maximum possible degree of F is $(v-1)td$. There are only $v-1$ variables in F. There are B^{v-1} points in the set $\mathcal{S} \times \cdots \times \mathcal{S}$. Applying the lemma to F, the probability that a point chosen at random will be a zero of F is

$$\begin{aligned}\frac{N_{v-1}(B)}{B^{v-1}} &= \frac{B^{v-1}-(B-D)^{v-1}}{B^{v-1}}\\ &= 1-\left(1-\frac{D}{B}\right)^{v-1}\\ &= \frac{vD}{B}-\frac{v(v-1)D^2}{2B^2}+\frac{(v-1)(v-2)D^3}{6B^3}+\cdots\\ &\leq \frac{dv(v-1)t}{B} \leq \frac{dv^2t}{B},\end{aligned}$$

ignoring the higher order terms. So a worst case bound for the probability that a point will be a zero of F, and thus a bad point and will lead to an erroneous candidate solution, is

$$\epsilon = \frac{dv^2t}{B}.$$

Taking logarithms we have

$$\log B = \log\left(\frac{dv^2t}{\epsilon}\right).$$

The following theorem is the fundamental result of this thesis. It shows that the time required to compute a candidate goal polynomial is polynomial in the d, v, the size of the candidate and the size of ϵ.

Theorem 1. *Assume d bounds the degree to which each variable in the goal polynomial, $P(X_1,\ldots,X_v)$, appears. Also assume $O(T)$ bounds the number of integer arithmetic operations required to compute $P(a_1,\ldots,a_v)$ given $a_1,\ldots,a_v$. Let ϵ be some number close to zero. Then it is possible to compute a candidate polynomial in time*

$$O((T+t^3)dvt\log^2(dv^2t\epsilon^{-1}))$$

where the probability that the candidate polynomial is different from P is ϵ and t is the number of terms in the candidate.

Proof: To prove this we will analyze the time required by the sparse modular algorithm. This will give a constructive proof of the theorem. The analysis is divided into two portions. First we will determine the number of points at which the value of the goal polynomial must be determined. Then the time required to determine the goal polynomial from its values is determined.

$P(X_1,a_2,\ldots,a_v)$ is computed by doing a dense interpolation of $d+1$ points. To compute $P(X_1,X_2,a_3,\ldots,a_v)$ requires d more univariate polynomials, each of which is computed by a sparse interpolation. So we need $(d+1)+dt$ values of P to compute $P(X_1,X_2,a_3,\ldots,a_v)$. In general we will need

$$(d+1)+dt+\cdots+dt = (d+1)+dt(v-1) \approx dvt$$

interpolation points. Each of these cost T integer operations. This gives the first term of the cost.

We now look at algorithm D, which performs the dense interpolation using the Chinese remainder algorithm. It was pointed out in the discussion of this algorithm that f and q will almost always be dense except possibly at the final step. We will assume a dense representation for these polynomials here. At the beginning of step D3, $f(x)$ is a polynomial of $i - 1$ terms and $q(x)$ is a polynomial of i terms. Using Horner's rule, computing $f(p_i)$ requires $i - 1$ additions and $i - 1$ multiplications, or $2i - 2$ total operations. Computing $q(p_i)$ will take $2i$ operations. The addition of the two polynomials will take i operations. Adding these up we have $2i - 2 + 2i + i + 3 = 5i + 1$ operations to compute $f(x)$. Updating of $q(x)$ requires $2i$ operations. All in all this comes to $7i + 1$ operations. Summing this for $i = 2, \ldots, k$, we get $\frac{7k^2+9k-16}{2}$ operations.

Algorithm S, which solves linear equations to obtain the sparse interpolation, is somewhat more complex and we will make a few crude assumptions in analyzing it. We will assume that $P(X_1, a_2, \ldots, a_v)$ and all the intermediate polynomials up through $P(X_1, \ldots, X_v)$ have t terms. That is, the number of terms does not decrease when some of the variables are replaced by integers.

There are t linear equations that must be solved. This will take about t^3 operations, so we will ignore all costs that are dominated by t^3. Each monomial contained in S is a product of $i - 1$ terms, and each term is raised to degree, at most, d. Evaluation of a monomial will thus cost $(i - 1)\log d$ operations. There are no more than t terms in S, so step S6 will take about $(i - 1)t \log d$ operations. Step S6 will be iterated t times to produce the each set of linear equations. Thus it will cost $(i - 1)t^2 \log d$ integer operations to produce the system of linear equations.

There will be t independent linear equations to be solved. Using straight forward algorithms this will take about $c_l t^3$ operations. Steps S5 through S7 will be executed d times for each variable so it will cost

$$(i - 1)dt^2 \log d + c_l dt^3 \approx c_l dt^3$$

operations to produce the polynomials $p_1, \ldots, p_t$.

There will be t terms in each of these polynomials, so algorithm D will be run t times. Since the time required by algorithm D is independent of t this cost of this step will be dominated by the t^3 term above. Since these steps are repeated for each variable we have about $O(dvt^3)$ integer operations for the lifting stage and

$$O\left(dvtT + dvt^3\right)$$

integer operations all in all.

The lemma indicates that the random integers must be chosen from a set of about $dv^2t\epsilon^{-1}$ elements. Arithmetic operations with integers this large require at most $\log^2(dv^2t\epsilon^{-1})$ actual operations. Thus the total time required by the sparse modular algorithm must be increased by this factor. This proves the theorem. ∎

Chapter IV

Some Applications

This chapter considers some of the more general applications of the sparse modular algorithm to multivariate computations in algebraic manipulation. One of the major problems that arises is intermediate expression swell. Intermediate expression swell can occur in several manners. In the course of a computation the determination of the answer may require the computations of extraneous variables, terms or factors. The final step of the computation is usually to remove the extraneous information. This is what happens in the computation of polynomial GCDs. The final term in a PRS generally has a large content that must be removed. Alternatively, computations may have a more subtle, yet still explosive behavior. For instance, the perturbation series expansions in quantum electrodynamics generally involve sums of very complex integrals. The integrals yield combinations of rational numbers, powers of π and values of the Rieman zeta function at odd positive integral values. Yet in certain computations [Ben77] these huge expressions simplify to simple rational integers.

For multivariate computations it is usually not difficult to carry through the computation with a single variable, replacing the other variables by randomly chosen numbers. After performing a number of these simple computations, their results can be interpolated to give the desired answer. The dense interpolation procedure could be used, but there is a problem. Usually, we do not have a very good bound on the degree of the various variables. Using the old interpolation scheme, the simplified problem must be solved about $(d+1)^v$ times. As we commented before, for large or moderate d and moderate v, this can be horrendously expensive. Since the basic computation may be quite complex and even dominate the cost of the interpolation it is especially important to minimize the number of points that need to be examined by the interpolation algorithm.

If the answer is not dense, our interpolation method can be used to great advantage since it needs only $v(d+1)t$ points from which to interpolate the result. Since our algorithm is probabilistic it is generally a good idea to verify the answer. For certain particular applications, like GCD of polynomials, verification is not difficult. For others, like the evaluation of determinants, there appears to be no good technique. Though this introduces some uncertainty into the computation, the uncertainty can be controlled by increasing the range from which the random points are chosen.

This chapter will show how these ideas may be applied to a few somewhat contrived examples. In section 1, an example using resultants is given. Section 2 presents a few simple ideas that show how rational numbers and rational functions may be interpolated. These ideas would be extremely useful when solving systems of equations. Section 3 uses this idea in computing a term in the inverse of a matrix.

1. A Resultant Calculation

The application of the sparse modular algorithm to many computations in algebraic manipulation, as outlined above, is a fairly straightforward application of algorithm S of the section *III.2*. *In particular the computation of determinants, the solution of linear equations* and the computation of polynomial resultants fall into this class. Here we will present a detailed example involving the computation of a piece of a resultant.

If a and b are relatively prime then one primitive element of the field $Q[\sqrt[r]{a}, \sqrt[r]{b}]$ is $\gamma_r = \sqrt[r]{a} + \sqrt[r]{b}$. The minimal polynomial for γ_r is known to be the resultant of $(y-x)^r - a$ and $y^r - b$ with respect to y. We will denote this polynomial by $f_r(x)$. For the first few r this polynomial is

$$\begin{aligned}
f_2(x) &= x^4 - (2a+2b)x^2 + a^2 - 2ab + b^2,\\
f_3(x) &= x^9 - (-3a+3b)x^6 + (3a^2 + 21ab + 3b^2)x^3 + (a-b)^3,\\
f_4(x) &= x^{16} - (4a+4b)x^{12} + (6a^2 - 124ab + 6b^2)x^8 + \cdots + (a-b)^4,\\
f_5(x) &= x^{25} - (-5a+5b)x^{20} + (10a^2 - 605ab + 10b^2)x^{15} + \cdots + (a-b)^5.
\end{aligned}$$

Apparently the first few terms of $f_r(x)$ are of the form

$$\begin{aligned}
f_r(x) = x^{r^2} &- r\left((-1)^r a + b\right)x^{r(r-1)}\\
&+ \left(\frac{r(r-1)}{2}a^2 - rq_r ab + \frac{r(r-1)}{2}b^2\right)x^{r(r-2)} + \cdots .
\end{aligned} \tag{1}$$

It would be very interesting if we could produce an explicit formula for $f_r(x)$ but that seems to be difficult. The first step in trying to determine an explicit formula is to determine the structure of a few of the terms of the minimal polynomial. The first two terms are pretty easy to guess and the structure of the third one is not too bad. In this section we will demonstrate how the coefficient of the $x^{r(r-2)}$ term can be determined fairly easily. If we have a table of these terms, it should not be difficult to determine the explicit form of the coefficient by empirical induction. Table 1 illustrates how costly this computation can be if $f_r(x)$ were computed using resultants. All times listed are in seconds. This computation was done with the aid of MACSYMA [MAC78].

It is clear that it would not be practical to continue to compute q_r in this manner. q_{20} will be determined to demonstrate how the sparse modular algorithm is used. We will compute $f_{20}(y)$ for several values of a and b and modulo several different primes. Then the coefficients of the x^{360} terms will be interpolated to compute q_r modulo the primes. These are combined via the Chinese remainder theorem to compute q_r.

First we need to estimate how big q_{20} will be. At each level in the table it appears that $q_r < 2^{2r-3}$. So we would expect q_{20} to be less than 2^{37}. Two primes near 2^{35} will more than suffice. We will use $p_1 = 34359738337$ and $p_2 = 34359738319$. The particular term of the resultant in which we are interested appears to have only 3 terms so we will need only three points for a and b. For simplicity we will choose $(a, b) = (1, 2), (2, 3), (3, 4)$

r	time	gctime	q_r
2	.090	0.	1
3	.167	0.	7
4	.287	0.	31
5	.665	0.	121
6	2.446	.919	456
7	5.397	1.876	1709
8	11.219	3.972	6427
9	23.816	9.709	24301
10	45.620	20.179	92368
11	182.768	113.993	352705

Table 1.

The resultant of $(y-x)^{20}-1$ and $y^{20}-2$ modulo p_1 is

$$\begin{aligned}x^{400} &- 60x^{380} - 8151507690x^{360} - 1440118365x^{340} + 11816121225x^{320}\\ &+ 15871655180x^{300} - 7583210257x^{280} + 11111546973x^{260} + 6825130515x^{240}\\ &+ 8301575652x^{220} + 11964418384x^{200} + 1965613129x^{180} - 909489473x^{160}\\ &- 16580105186x^{140} + 16495845941x^{120} + 13937306864x^{100}\\ &- 5608147531x^{80} + 4122651953x^{60} + 10757822857x^{40}\\ &+ 12644080612x^{20} + 1.\end{aligned}$$

Though this is large it is much smaller and can be computed significantly faster than the general resultant. Computing this resultant for $(a,b) = (2,3)$ and $(3,4)$ we determine that the x^{360} term has 9905214887 and -14549308753 as its coefficient in each case. This leads to the following system of linear equations

$$\begin{aligned}-8151507690 &= 1^2w_1 + 1\cdot 2w_2 + 2^2w_3\\ 9905214887 &= 2^2w_1 + 2\cdot 3w_2 + 3^2w_3\\ -14549308753 &= 3^2w_1 + 3\cdot 4w_2 + 4^2w_3\end{aligned}$$

The solution to this system is $(w_1, w_2, w_3) = (190, -4075754320, 190)$. Proceeding similarly with p_2 as the modulus we get the following system of equations

$$\begin{aligned}-8151509130 &= 1^2w_1 + 1\cdot 2w_2 + 2^2w_3\\ 9905210549 &= 2^2w_1 + 2\cdot 3w_2 + 3^2w_3\\ -14549317411 &= 3^2w_1 + 3\cdot 4w_2 + 4^2w_3\end{aligned}$$

which has $(190, -4075755040, 190)$ as its solution.

The 190's are expected; they correspond precisely with (1). To compute the middle coefficient we apply algorithm C to the system

$$\begin{aligned}w &\equiv -4075754320 \pmod{34359738337}\\ w &\equiv -4075755040 \pmod{34359738319}\end{aligned}$$

This gives -1378465287800 for the middle coefficient, and 68923264390 for q_r.

Each of the resultant computations required about 11.5 seconds. Extrapolating from smaller computations, had the resultants been computed over the integers, they would have required about 142 seconds each. Furthermore, when the systems of linear equations were solved, the coefficients involved did not grow, as they would have had they been rational integers.

In general this seems to be the best practice when in applying the sparse modular techniques. The problem is solved using the algorithm several times over finite fields and the Chinese remainder theorem is used to reconstruct the answer.

The computation of q_{20} required about 70 seconds of computation. Extrapolating from the preceding table it appears that the computation of the full resultant would take about 10 hours, and would probably not fit in most machines. Though the resultant gives more information, for this problem we were only interested in a small portion of the answer and the extra information is not useful at this stage.

Furthermore, by repeating this process all the terms could have been determined if desired, either one at a time or in parallel (in all senses of the word) without requiring significant quantities of storage.

2. Recovering Rational Numbers and Rational Functions

Thus far we have only been concerned with using the interpolation techniques to determine polynomials from their values. In this section we will consider some ideas that allow the sparse interpolation scheme to be used to determine rational numbers and rational functions from their values.

Let m be the ratio of two small integers x and y modulo p and assume the product $2xy$ is less than p. Then there is an integer q such that

$$my - pq = x.$$

Rewriting this we get

$$\frac{m}{p} - \frac{q}{y} = \frac{x}{py}.$$

Since x is assumed to be small q/y must be a convergent of the continued fraction expansion of m/p. From this it is easy to determine x. This all all depends on the following theorem which is proved in [Har68] (theorem 184).

Theorem 2. *If p/q is rational number approximation to ς which satisfies*

$$\left|\frac{p}{q} - \varsigma\right| < \frac{1}{2q^2}$$

the p/q is a convergent of the continued fraction of ς.

Algorithm T is given two numbers m and p and computes a pair of integers x and y such that $x \equiv my \pmod{p}$. This done by computing the continued fraction of m/p whose partial quotients are denoted by a_i and convergents by q_i/p_i.

T1. [Initialize] Set $q_{-1} \leftarrow 1$, $q_0 \leftarrow 0$, $y_{-1} \leftarrow 0$ and $y_0 \leftarrow 1$. Set $r \leftarrow m$ and $s \leftarrow p$. Finally set $i \leftarrow 1$.

T2. [Partial quotient] Set a_i to the integer quotient of r and s. Then set r to s and s to the remainder of r divided by s.

T3. [Next term of CF] Set $q_i \leftarrow a_i q_{i-1} + q_{i-2}$ and $y_i \leftarrow a_i y_{i-1} + y_{i-2}$.

T4. [Enough?] If $y_i^2 \leq 2p$ then set $i \leftarrow i + 1$ and go to step T2.

T5. [End] Return the fraction $(my_{i-1} - pq_{i-1})/q_{i-1}$.

This algorithm allows us to determine a rational number from its residue modulo a prime. A similar algorithm can be used to determine a rational function from its value modulo a ideal $(p(x))$. If $p(x) = x^n$, then its value is a truncated power series. In this case the rational function that this algorithm returns is known as a *Padé approximation*. A recent study of this idea is contained in [McE78].

If there are several variables in the rational function that we are trying to determine, then we can replace all but one of the variables by an integer, compute the power series expansion of the answer in terms of the other variable, and then interpolate the coefficients. This technique is quite satisfactory and is particularly appropriate to problems like solving differential equations where it is impossible to replace all the variables by constants.

There is another interesting idea that can be used if even computing with power series in one variable is impractical. The basic idea is to compute the rational function f/g by taking its values, $r_1, \ldots, r_k$, modulo $p_1, \ldots, p_k$ and converting them to fractions, $f_1/g_1, \ldots, f_k/g_k$. (It is assumed that $2f_i g_i < p_i$.) The numerator and denominator can then be separately interpolated. There is a slight complication. Though the rational numbers that are computed are the values of f/g at the point in question, f_i will not be the value of f unless the values of f and g are relatively prime. As far as I can see there is no good way to ensure this, and there also does not seem to be an easy way to analyze the probabilities involved. Empirically, however this seems to be a quite reasonable way to determine f and g. It is perhaps best used as a heuristic. If the polynomials that are produced turn out to be too dense, they are discarded and new points are chosen on the assumption that there was a GCD between the values of the numerator and denominator.

There is another problem, though it is easily overcome. Generally, the p_i are chosen to be just smaller than a machine word for efficiency considerations. The values of f and g can be quite large, especially in view of the possibility of the values of f and g having a nontrivial GCD, so $2f_i g_i < p_i$ may be false. This problem can be avoided by combining the several of the r_i via the Chinese remainder theorem before applying algorithm T. This will yield the value of f/g modulo $p_i p_j$.

3. Determinants

In this example we will demonstrate how the sparse algorithm could be used to compute an element in the inverse of a matrix. These techniques may also be applied to compute the solution of a system of linear equations. The matrix we will consider is actually quite small, but will serve to illustrate our methods. This example is from [Wan76].

$$\mathcal{M} = \begin{pmatrix} x^2 & y^2 & 0 & u^2 & 0 & z^2 \\ x^4 & y^4 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ x^5 & 0 & 0 & 0 & 0 & 0 \\ x^3 & y^3 & 0 & 0 & 0 & z^3 \\ x & y & 0 & u & v & z \end{pmatrix} \tag{7}$$

We will be trying to determine the (3, 1) component of inverse of this matrix.

Almost certainly, the elements of the inverse of $\mathcal{M}$ will be, non-polynomial, rational functions. We will use the techniques of the previous section solve this problem.

The degree bounds for the determinant of $\mathcal{M}$ are easy to compute. Since each of the variables appears in precisely one column in the determinant, no variable appears to higher degree than it appears in the matrix. If we compute the inverse of $\mathcal{M}$ modulo 34359738337, with random integers substituted for x, y, z, u and v, we discover that the (3, 1) coefficient is independent of x, y and z. The following table gives the values of this coefficient for various values of u and v and what algorithm T yields for f/g.

u	v	$\mathcal{M}_{31}$	f_i	g_i
1	1	−2	−2	1
2	3	14316557640	−5	12
3	4	6681060232	−7	36
−2	3	2863311528	−1	12
4	7	3374617158	−11	112
−2	−5	−15461882252	7	−20

The f_i/g_i can be easily combined by the results already presented to compute the exact value

$$\mathcal{M}_{31} = \frac{-u-v}{u^2 v}$$

.

Again each element of the inverse could be computed independently (and in parallel) if the value of the entire matrix were desired.

Chapter V

Ideal-adic Arithmetic

As we shall see, Hensel's lemma and the algorithms based on it are clever applications of Newton's iteration to problems over particular domains. In this chapter we present Newton's method in its familiar form for equations over the reals. We will show how this may be extended to coupled systems of equations and analyze its convergence properties.

Although it is possible to apply Newton's method directly to polynomial problems it has proven to be easier to apply when the coefficients of the polynomials lie in some p-adic domain. In section 2 we present a synopsis of the basic results about p-adic numbers and present some examples.

In section 3 we generalize this slightly, and consider $\underline{m}$-adic domains, where $\underline{m}$ is some ideal contained in a commutative noetherian ring. The initial presentation is kept quite abstract so that the results can be carried over to algebraic number rings and other rather complex structures. However, a number of examples are presented which should help to clarify the ideas involved.

Virtually nothing in this chapter is new and the mathematically sophisticated reader can safely skip it, referring back to it for notation as needed in the reading of later sections. Hopefully we have presented enough of the basic concepts to allow those with other backgrounds to follow the bulk of these results.

1. Newton's Iteration

Assume we are trying to find the zeroes of the function $f(z)$. If we have some idea were a solution of this equation is, then it is possible to use the Newton-Raphson iteration to refine our guess to a zero of $f(z)$. Often the guess need not be very close at all for this iteration to converge.

Assume x_k is a real number close to a zero of $f(x)$ which we call $\hat{x}$. We next try to find a better approximation x_{k+1}. Expanding $f(x_{r+1})$ as a power series in $(x_{k+1} - x_k)$ by Taylor's formula we have

$$f(x_{k+1}) = f(x_k) + (x_{k+1} - x_k)f'(x_k) + \cdots.$$

We assume that $f(x_{k+1})$ is close to zero so that we may replace the left hand side of this equation by zero. We will also ignore the higher order terms on the right hand side. All these assumptions leave us with a simple linear equation which can be solved for x_{k+1}.

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}.$$

This is the familiar form of the Newton-Raphson iteration. If the starting point x_0 is sufficiently well chosen then the sequence $\{ x_k \}$ will converge to a root of $f(z)$. This iteration converges quadratically to the roots of $f(x)$.

2. p-adic Numbers

The key problem in using Newton's method to solve a system of equations is determining a "good point" at which to begin the iteration. Once such a point has been found, the iteration will quickly converge to a solution of the system. The "goodness" of a point is directly related to its closeness to a zero of the system of equations. In the last section all arithmetic was performed with real numbers. Consequently, the closeness of two numbers was measured by determining the absolute value of their difference. To get a good initial approximation to a solution we must have some idea of the magnitude of the solutions. In physical problems this information can often be determined from the physical constraints on the problem.

For problems in algebraic manipulation it is often easy to solve the system of equations exactly modulo some prime. At worst we can exhaustively search the (finite) set of all possible solutions. For many problems it is possible to do much better than this. Thus we would like to modify the concept of distance somehow so that a solution to the system modulo a prime yields a "good" starting point.

This modification leads to the p-adic numbers, which we examine in this section. The basic ideas behind p-adic numbers were originally developed by Hensel [Hen18] and applied to problems in number theory. Over the years the p-adic outlook has permeated large portions of algebraic geometry and number theory and is now considered to be one of the cornerstones of modern mathematics.

What we are looking for is a new measure of distance that can be applied to integers. Since it is possible to compute solutions to problems modulo some prime, we would like to say that two numbers that have the same residue for a given modulus are close. Thus this new distance measure must indicate that 1 and 4 and 163 are close for a modulus of 3. The first test for proximity of two numbers is that their residue modulo 3 is the same.

It will be easier to continue if we restrict ourselves to proximity to zero. This is not much of a restriction since the distance between two integers p and q, should also be the distance between 0 and $p - q$.

So we now ask how to measure the distance between an integer p and 0. Assume that p is congruent to 0 modulo 3. Then p is divisible by 3 and we can ask how close $p/3$ is to zero. Clearly if $p/3$ were also congruent to zero then p would be "closer" to zero than if $p/3 \not\equiv 0 \pmod 3$. After all $0/3 \equiv 0 \pmod 3$.

If we write the number 163 in the base 3 system we have

$$1 + 0 \cdot 3 + 0 \cdot 3^2 + 0 \cdot 3^3 + 2 \cdot 3^4.$$

Clearly this number is quite close to 1, since the difference of 163 and 1 is divisible by a large power of 3.

We can now define the distance measure as follows. Let a be an integer, and assume 3^r divides a but 3^{r+1} does not. We can write a as $3^r p$. The the 3-*adic* valuation of a is

$$\|a\|_3 = \|3^r p\|_3 = 3^{-r}.$$

When $p = 0$ we will say that $r = \infty$.

We can extend this definition to rational numbers also. All rational numbers can be written in the form $3^r p/q$ where 3 does not divide p or q, and r can be positive, negative or zero. Then

$$\|3^r p/q\|_3 = 3^{-r}.$$

This definition can be extended in the obvious manner to other primes. In general these distance measures are called p-adic valuations.

In elementary analysis and topology, absolute values are used to define convergent sequences and the convergent sequences are used to "complete" the rational numbers to yield the reals. We will now use the p-adic valuation to complete the integers to give the *p-adic integers*, $\mathbf{Z}_p$.

The basic idea is quite simple. Let

$$A = \{a_0, a_1, a_2, \ldots, a_k, \ldots\}$$

be a sequence of rational integers such that

$$\|a_i - a_j\|_p = p^{-\min(i,j)}.$$

Thus $a_i - a_{i-1} = a_i p^i$. The we can view $\{a_0, a_1, \ldots, a_k\}$ to be a sequence of increasingly better approximations of a_k. The whole set is thus a sequence of increasingly good approximations to

$$\lim_{k \to \infty} a_k.$$

This limit is often represented by the infinite series

$$a_0 + a_1 \cdot p + a_2 \cdot p^2 + \cdots. \tag{8}$$

The a_i are chosen from the range $0 \le a_i < p$. This is called the *power series representation* for a p-adic integer.

It is clear that the positive integers have a unique representation of this form. The p-adic integers also include quantities for which (8) is an infinite series. Assume that $\alpha = \{a_0, a_1, \ldots\}$ and β are two p-adic integers such that $a_k \equiv \beta_k \pmod{p^k}$, for all k; that is, $\|a_k - \beta_k\|_p = p^{k+1}$. Then we say that α and β are equal. If this is the case then they must have the same power series representations. From now on we will only work with the power series representations of p-adic integers.

Negative integers also have p-adic representations. Consider the number -1. One sequence of good approximations by positive integers is $p^k - 1$. But this can be written in the following manner

$$p^k - 1 = (p-1)(p^{k-1} + p^{k-2} + \cdots + p + 1)$$
$$(p-1) + (p-1) \cdot p + (p-1) \cdot p^2 + \cdots + (p-1) \cdot p^{k-1}$$

Taking the limit as $k \to \infty$ we have

$$-1 = (p-1) + (p-1) \cdot p + (p-1) \cdot p^2 + \cdots.$$

It is easy to see that this is one less than zero:

$$\begin{aligned}-1+1 &= 1+(p-1)+(p-1)\cdot p+(p-1)\cdot p^2+\cdots\\ &= p+(p-1)\cdot p+\cdots\\ &= p^2+(p-1)\cdot p^2+\cdots\\ &= 0.\end{aligned}$$

Arithmetic with p-adic integers is performed in much the same manner as power series arithmetic. The only difference is that a normalization step must be performed to ensure the coefficients of p^k lie in the range $0 \le a_k < p$. For instance, to compute

$$q^2 = (1+3+3^2+2\cdot 3^4+\cdots)^2$$

in the 3-adic integers we begin by performing the power series multiplication.

$$\begin{aligned}q^2 &= 1+2\cdot 3+3\cdot 3^2+2\cdot 3^3+5\cdot 3^4+\cdots\\ &= 1+2\cdot 3+(1+2)\cdot 3^3+5\cdot 3^4+\cdots\\ &= 1+2\cdot 3+(1+5)\cdot 3^4+\cdots\\ &= 1+2\cdot 3+\cdots\end{aligned}$$

The p-adic integers clearly include all of $\mathbf{Z}$ since the positive integers and -1 are p-adic integers. There are other elements also. For instance,

$$-\frac{1}{2} = \frac{1}{1-3} = 1+3+3^2+3^3+\cdots.$$

So, $-1/2$ is an element of $\mathbf{Z}_3$.

Certain algebraic numbers also lie in $\mathbf{Z}_p$. We will now show that X^2-7 has a solution that lies in $\mathbf{Z}_3$. Let the symbol $\sqrt{7}$ represent a solution of X^2-7. If $\sqrt{7}$ does lie in $\mathbf{Z}_3$ then we must have

$$\sqrt{7} = a_0 + a_1\cdot 3 + a_2\cdot 3^2+\cdots.$$

Therefore, we must have $a_0 - 7 \equiv 0 \pmod 3$, so a_0 is either 1 or -1. We pick 1; using the other solution of X^2-7 using identical steps.

As with the computation of a_0, the computation of the other a_i proceeds by considering the equation

$$(a_0 + a_1\cdot 3+\cdots)^2 - 7 \equiv 0 \pmod{3^{i+1}}$$

and solving the resulting equation for a_i. Thus to determine a_1,

$$(1+a_1\cdot 3+\cdots)^2 - 7 \equiv 1+2a_1\cdot 3 - 7 \equiv 0 \pmod{3^2}$$

So $a_1 = 1$ and $\sqrt{7} = 1+3+\cdots$. Continuing

$$7 \equiv (1+3+a_2\cdot 3^2)^2 = 16+8a_2\cdot 3^2 \pmod{3^3}$$

so $a_2 = 1$. This may continued arbitrarily far:

$$\sqrt{7} = 1+3+3^2+2\cdot 3^4+\cdots.$$

Notice that the equations for a_1 and a_2 were both linear. This is generally the case.

3. $\underline{m}$-adic Arithmetic

This section requires a bit more mathematical sophistication then the earlier chapters and a bit deeper understanding of the concepts of abstract algebra. However, we use this sophistication only to obtain more general algorithms, the reader could merely interpret most everything we say using p-adic integers and would lose little in terms of content.

Let R be a commutative ring, $\underline{m}$ an ideal of R. Then there is a sequence of rings and canonical homomorphism as follows.

$$R/\underline{m} \xleftarrow{\theta_2} R/\underline{m}^2 \xleftarrow{\theta_3} \cdots \xleftarrow{\theta_i} R/\underline{m}^i \xleftarrow{\theta_{i+1}} \cdots \tag{3}$$

Consider the set $\mathfrak{R} = R/\underline{m} \times R/\underline{m}^2 \times \cdots$ and an element of $(a_i) \in \mathfrak{R}$ where $a_i \in R/\underline{m}^i$. An element of $\mathfrak{R}$ is said to be *coherent* if $\theta_{i+1}a_{i+1} = a_i$ for all i. The set of all coherent elements of $\mathfrak{R}$ is a ring. This ring is called the *inverse limit* of of the sequence and we write

$$R_{\underline{m}} = \lim_{\leftarrow} R/\underline{m}^i.$$

$R_{\underline{m}}$ is said to be the $\underline{m}$-adic completion of R. The last section dealt with the case $R = \mathbb{Z}$. The only ideals are the principal ones generated by a rational integer, m. The sequence of rings in this case is $\mathbb{Z}/(m)^i = \mathbb{Z}/(m^i)$.

For example, $\sqrt{7}$ is represented in $\mathbb{Q}_3$ the coherent sequence

$$\sqrt{7} = (1, 4, 13, 13, 175, \ldots) = a.$$

If we write this sequence as a sequence of differences we have the familiar form, $(1, 3, 9, 162, \ldots)$. Unfortunately, the subtractions we just performed have to be made precise. We subtracted an element of $R/(m^i)$ from an element of $R/(m^{i+1})$. Thus we need an embedding of $R/(m^i)$ into $R/(m^{i+1})$, i.e. the inverse of θ_{i+1}. There is no unique inverse. Before resolving this problem, we want to examine one additional example to bolster our intuition.

Let F be a field and consider the ring $R = F[x, y, z]$. The ideal $\underline{m} = (x, y, z)$ is maximal, so $R/\underline{m}$ is a field. Intuitively, the $\underline{m}$-adic completion of R should be the ring of power series in x, y and z, $F[[x, y, z]]$. This is in fact what happens. The first ring of (3) is F. The next ring is $R/\underline{m}$ which is the ring of power series in x, y and z truncated after first order. Since $(x, y, z)^2 = (x^2, y^2, z^2, xy, xz, yz)$, the next ring is the truncated power series ring of order 2. Clearly the inverse limit is the general power series ring. The map θ_i merely truncates the power series to order $i - 1$.

Again the inverse map is not unique, but since there is an natural embedding of the ring of power series of order i into the ring of order $i + 1$, θ_{i+1} has a "natural inverse." Notice that we can extend θ_i to a map from $R_{\underline{m}}$ onto $R/\underline{m}^{i-1}$.

We would like to write

$$a = a_0 + a_1 \cdot \underline{m} + a_2 \cdot \underline{m}^2 + \cdots.$$

So, in some sense, $a_i \cdot \underline{m}^i$ a the projection of a to the "subspace spanned by $\underline{m}^i$. There are only two conditions which need to be satisfied by $a_i \cdot \underline{m}^i$

(1) $a_i \cdot \underline{m}^i \in \underline{m}^i R_{\underline{m}}$

(2) $a - \sum_{i=0}^{n} a_i \cdot \underline{m}^i \in \underline{m}^{i+1}$

The first property ensures that, under the $\underline{m}$-adic topology, the $a_i \cdot \underline{m}^i$ get smaller and smaller, i.e. their sum will converge. By the second condition their sum will converge to a. This is all we need. The natural mappings, if they can be determined, are suitable.

Chapter VI

Hensel's Lemma

In its most common form, Hensel's lemma indicates when a factorization of a polynomial over a ring $R/\underline{m}$ can be lifted to a factorization over the ring $R_{\underline{m}}$. Lately, it has been used to lift factorizations modulo a prime to factorizations over the integers and then to multivariate factorizations. It is also used to lift GCD and other computations useful in algebraic manipulation.

In this chapter we will present a "new" framework for discussing computational formulations of Hensel's lemma. Actually, the formulation we use is fairly well known in mathematical circles. In the first section we present the new framework and show how to solve simple problems with it without making any particular appeal to Hensel's lemma. In the second section we prove Hensel's lemma, in its more familiar form, using the techniques of the first section. The final section presents the formulation that been used in algebraic manipulation circles and discusses its similarities and differences with our formulation.

1. A New Formulation of Hensel's Lemma

Our version of Hensel's lemma uses an $\underline{m}$-adic version of Newton's iteration to obtain the zeroes of a system of equations in an $\underline{m}$-adic field. The first part of this section converts a number of well known problems, and a few problems not so often considered, into systems of equations which must be solved. In the second part we show how to apply Newton's iteration to systems of equations over $\underline{m}$-adic fields. We concentrate on a linearly convergent algorithm since it is rather simple, but in the final section we also discuss the quadratic algorithm and point out its draw-backs.

1.1. Reducing Problems to Solving Equations

Both factoring polynomials and computing their GCD's are included among the problems which can be recast as systems of coupled, non-linear equations. By reducing these and other problems to systems of equations we will obviate the individual analyses which were previously required. The algorithms produced using this method benefit from all the advantages ascribed to the "Hensel lemma technique" and our improvements to it.

The first problem we consider does not appear to have been studied much but is interesting nonetheless. As we shall see it leads to a useful heuristic for solving quintic equations.

Assume $f(z)$ is a monic polynomial over Z. We want to know if $f(z)$ has a polynomial divisor of a given degree over Z. In its most common form this problem asks whether $f(z)$ has any linear factors and thus possesses integral zeroes. By determining if $f(z)$ has any quadratic factors we will be able to determine if $f(z)$ has any zeroes in a quadratic number field.

To illustrate this technique assume

$$f(z) = z^5 + c_1 z^4 + c_2 z^3 + c_3 z^2 + c_4 z + c_5$$

and that we are looking for a quadratic factor. (Notice that if we can find a quadratic factor then what will be left is a cubic polynomial. Thus this could form the basic of a simple algorithm for computing the roots of certain types of quintic polynomials.) Any such quadratic factor will be monic and thus of the form: $g(z) = z^2 + az + b$. Since $g(z)$ divides $f(z)$, the remainder must be zero. Dividing $f(z)$ by $g(z)$ we get a remainder of

$$[a^4 - c_1 a^3 + c_2 a^2 - c_3 a + b^2 - c_2 b - 3a^2 b + 2c_1 ab + c_4]z \\ + a^3 b - 2ab^2 - c_1 a^2 b + c_1 b^2 + c_2 ab - c_3 b + c_5.$$

For this to be zero the coefficients of both the linear and constant terms must be zero. We thus get the following two equations in the unknowns a and b.

$$\begin{aligned} a^4 - c_1 a^3 + c_2 a^2 - c_3 a + b^2 - c_2 b - 3a^2 b + 2c_1 ab + c_4 &= 0 \\ a^3 b - 2ab^2 - c_1 a^2 b + c_1 b^2 + c_2 ab - c_3 b + c_5 &= 0 \end{aligned} \tag{1}$$

Any solution of this system of equations will yield a quadratic polynomial which divides $f(z)$.

From the classical theory of equations, we know that all quintic equation, over the rationals, can be transformed into the Bring-Jerrard form $z^5 + c_4 z + c_5 = 0$. Assuming $c_1 = c_2 = c_3 = 0$ equations (1) simplify considerably. By elimination theory we find that a and b are zeroes of the following polynomials

$$a^{10} - 3c_4 a^6 + 11c_5 a^5 - 4c_4^2 a^2 - 4c_4 c_5 a + c_5^2,$$
$$b^{10} - c_4 b^8 - c_4^2 b^6 - 2c_5^2 b^5 + c_4^3 b^4 + c_4 c_5^2 b^3 + c_5^4.$$

Consider now the general case, $f(z)$ is a polynomial of arbitrary degree and we are looking for a polynomial, $g(z)$, which divides $f(z)$ and is of degree n. The remainder of $f(z)$ when divided by $g(z)$ is of degree $n - 1$. Equating each to zero, we get n equations. There were n unknown coefficients in $g(z)$ since we know $g(z)$ is monic.

If $f(z)$ is not monic we can still use this procedure. Using an additional indeterminate for the leading coefficient will not work since there doesn't seem to be an additional equation lying around. Instead we monicize the polynomial. Let

$$f(z) = c_0 z^n + c_1 z^{n-1} + \cdots + c_n$$

We could monicize $f(z)$ by dividing it by c_0 but this would introduce fractions that would be difficult to handle. Then

$$c_0^{n-1} f(z) = (c_0 z)^n + c_0 c_1 (c_0 z)^{n-1} + \cdots + c_n c_0^{n-1} = \bar{f}(c_0 z).$$

Since $\bar{f}(z)$ is monic we can use the above procedure with $\bar{f}$ and then adjust the leading coefficient.

On the other hand if we knew the leading coefficient of $g(z)$, or for that matter any coefficient of $g(z)$, then we would have enough equations to solve the system.

Assume we have a monic polynomial

$$f(x) = x^n + f_1 x^{n-1} + \cdots + f_n$$

which we are trying to factor into two polynomials g and h of degrees r and s respectively. Then we have

$$f(x) = (x^r + g_1 x^{r-1} + \cdots + g_r)(x^s + h_1 x^{s-1} + \cdots + h_s)$$

Multiplying out the polynomials and equating the coefficients of x gives the following system of equations.

$$\begin{aligned} g_1 + h_1 &= f_1 \\ g_2 + g_1 h_1 + h_2 &= f_2 \\ &\vdots \\ g_r h_{s-1} + g_{r-1} h_s &= f_{n-1} \\ g_r h_s &= f_n \end{aligned} \qquad (1)$$

The f_i are known so we have n equations in $r + s = n$ unknowns. Since there are the same number of unknowns as equations we have reason to believe that there is are at most a finite number of solutions to the system of equations.

Notice that had we not assumed that f was monic then there would have been an additional equation

$$g_0 h_0 = f_0.$$

Unfortunately, there are two additional unknowns. The system would not have had a finite number of solutions. However, if one of the g_i or h_i is known then the solution set would be finite. (This is a slight generalization of Hensel's Lemma.)

If it was known that $f(x)$ could be expressed as the product of more than two factors then one could use the technique sketched above recursively. However it is possible to set up the systems of equations for more than two factors without much difficulty. For instance, assume $f(x)$ is known to be the product of three factors $A(x)B(x)C(x)$ of degrees r_1, r_2 and r_3 respectively. We then know that

$$\begin{aligned} f(x) &= A(x)B(x)C(x) \\ x^n + \cdots + f_n &= (x^{r_1} + \cdots + a_{r_1})(x^{r_2} + \cdots + b_{r_2})(x^{r_3} + \cdots + c_{r_3}) \\ &= x^{r_1+r_2+r_3} + a_1 x^{r_2+r_3} + b_1 x^{r_1+r_3} + c_1 x^{r_1+r_2} + \cdots + a_{r_1} b_{r_2} c_{r_3} \end{aligned}$$

Again we equate the coefficients x^i on the right hand side of the equation with those on the left hand side. There are $n = r_1 + r_2 + r_3$ equations and an equal number of unknowns, a_i, b_j and c_k so we seem to have won again.

Recall that the square-free decomposition of a polynomial $f(z)$ is a factorization into the form

$$f(z) = P_1(z) P_2(z)^2 P_3(z)^3 \cdots P_n(z)^n$$

where each of the P_i is square-free and every pair, P_i and P_j, is relatively prime. The classical way to compute the square-free decomposition of a polynomial is to first compute the GCD of $f(z)$ and $f'(z)$:

$$\gcd(f(z), f'(z)) = P_2(z) P_3^2(z) \cdots P_n(z)^{n-1}.$$

This allows us to determine $P_1(z)$ and we can repeat the process with the GCD to compute P_2 and so on.

For multivariate polynomials, the first GCD is extremely costly. Trager and Wang [Tra79] have suggested using an evaluation homomorphism to reduce to a univariate polynomial, computing the square free decomposition using univariate GCDs and then using Hensel's lemma to get the square-free factorization. Using our formulation, all that is necessary to lift the univariate factorization is to set up a system of equations that must be solved. For $n = 2$, this is quite easy.

$$\begin{aligned} f(x) &= A(x)B(x)^2 \\ x^n + \cdots + f_n &= (x^{r_1} + \cdots + a_{r_1})(x^{r_1} + \cdots + b_{r_2})^2 \\ &= x^{r_1+2r_2} + (a_1 + 2b_1)x^{r_1+2r_2-1} + \cdots + a_{r_1}b_{r_2}^2 \end{aligned}$$

For larger n, similar results are easily obtained.

The GCD problem can be handled in a very similar manner. Assume we want to compute H which is the GCD of F and G. Again we will treat all polynomials as polynomials in x with coefficients which are to be determined. Letting $F = AH$ and $G = BH$ we get a system of equations which is similar to the system which was derived for a factorization (1). To obtain a solution to this system we will also have to obtain values for A and B.

It would be interesting to see if it is possible to compute H without computing the cofactors. It is possible to set up a system of equations which does this but in practice it will probably prove too expensive. Since H divides F, the remainder of F when divided by H must be zero. This gives one set of equations which the coefficients of H must satisfy. In fact it is possible to do the same with G and produce 2 systems of equations which H must satisfy. For instance with F and G monic, of degree 4 and H quadratic, we get the following system.

$$\begin{aligned} 2h_1h_2 - f_1h_2 - h_1^3 + f_1h_1^2 - f_2h_1 + f_3 &= 0 \\ h_2^2 - h_1^2h_2 + f_1h_1h_2 - f_2h_2 + f_4 &= 0 \\ 2h_1h_2 - g_1h_2 - h_1^3 + g_1h_1^2 - g_2h_1 + g_3 &= 0 \\ h_2^2 - h_1^2h_2 + g_1h_1h_2 - g_2h_2 + g_4 &= 0 \end{aligned}$$

Remember that only h_1 and h_2 are variables here. Everything else is determined *a priori*. At most two of these equations will be independent.

I have been unable to produce a similar set of equations for resultants. All the systems I have managed to produce have had fewer equations than unknowns. It seems unlikely that a system of equations which has only a finite number of solutions exists, but a proof seems difficult. From a practical point of view this does not really matter since the sparse modular lifting algorithm can be used to compute resultants. Nonetheless it is still curious that no such system seems to exist.

1.2. Solving the Systems of Equations

Now that we know how to set up systems of equations, we need to show that there actually is an easy way to solve the resulting systems of equations. We will use the $\underline{m}$-adic language developed in section VI.3 throughout this section.

We assume that R is a commutative ring with unit and that $\underline{m}$ is an ideal of R. For simplicity, we begin with the single variable version of Newton's iteration. Let $f(X)$ be a polynomial in X over R. Let x_i be a coherent sequence of values in R such that $f(x_k) \equiv 0 \pmod{\underline{m}^{k+1}}$. Clearly

x_k converges to an element of $R_{\underline{m}}$, r. If we are lucky, the limit will also lie in R. There is a canonical embedding of R in $R_{\underline{m}}$ so we might as well consider $f(X)$ to be a polynomial over $R_{\underline{m}}$

r can be written as a series

$$r = r_0 + r_1 \cdot \underline{m} + r_2 \cdot \underline{m} + \cdots,$$

where $r_k \cdot \underline{m}^k = (x_k - x_{k-1})$. Following Newton's example, we expand $f(x_k)$ as a power series in $(x_k - x_{k-1})$:

$$f(x_k) = f(_{k-1}) + f'(x_{k-1})(x_k - x_{k-1}) + \cdots.$$

Since $(x_k - x_{k-1}) = r_k \cdot \underline{m}^k \in \underline{m}^k$, $(x_k - x_{k-1})^2$ will be zero modulo $\underline{m}^{k+1}$. So we have

$$0 \equiv f(x_k) \equiv f(x_{k-1}) + f'(x_{k-1})(x_k - x_{k-1}) \pmod{\underline{m}^{k+1}} \tag{2}$$

This equation can be solved for the correction term $(x_k - x_{k-1})$. Unfortunately, using this iteration involves computing the inverse of $f'(x_{k-1})$ at each stage in the iteration. There are two choices, if we compute the the inverse at each step, and make use of all the information available, the iteration converges quadratically. On the other hand, if we need an iteration which is only linearly convergent it is possible set the iteration up so that only one inversion is performed.

Since $(x_k - x_{k-1})$ is an element of $\underline{m}^k R_{\underline{m}}$ we need only compute $f'(x_{k-1})$ modulo $\underline{m}$. Again using Taylor's theorem, this time to expand $f'(x_{k-1})$ at x_0, we get

$$f'(x_{k-1}) = f'(x_0) + f''(x_0)(x_{k-1} - x_0) + \cdots = f'(x_0) \tag{3}$$

all modulo $\underline{m}$. Combining (2) and (3) we have

$$0 \equiv f(x_{k-1}) + f'(x_0)(x_k - x_{k-1}) \qquad \pmod{\underline{m}^{k+1}}$$

$$x_k - x_{k-1} \equiv -f'(x_0)^{-1} f(x_{k-1}) \qquad \pmod{\underline{m}^{k+1}} \tag{4}$$

Since k steps are required to compute an approximation to r whose first k terms agree with r we say that this iteration is linearly convergent. It is possible to produce a quadratically convergent iteration by a very similar method. We do this in section 1.3.

We give a simple example to illustrate the linear iteration. Assume we wish to compute the square root of an integer. The most common method used is to use Newton's iteration with integers and round everything off to integers. The algorithm we present computes the square root of n using a linearly convergent iteration. For the starting point we use the square root of n computed modulo 3. We begin by dividing out all powers of 3. If an odd number of powers of 3 divided n then n has no square root. Now n is congruent to either 1 or 2 modulo 3. If 2 then n does not have an integral square root otherwise the initial "guess" is 1.

For f we use $f(r)^2 - n = 0$. Equation (4) now reads

$$x_k - x_{k-1} = \frac{f(x_{k-1})}{2x_0} \qquad \pmod{\underline{m}^{k+1}}$$

$\underline{m} = (3)$ and x_0 is always 1. Both the left and right hand sides are divisible by 3^k so this really is a congruence modulo 3. Since $2^2 = 1 \pmod 3$ the correction term is

$$3^k\left(2\left(\frac{f(x_{k-1})}{3^k}\right) \pmod 3\right).$$

We now derive the iteration for solving systems of equations. The procedure is very similar, but it is easy to become mired in indices if some sort of vector notations is not used. On the other hand, vector notations can easily lead to wrong answers if the basis behind them is not thoroughly understood.

First, Taylor's theorem in several variables: Let $f(z_1, \ldots, z_n)$ be a function in the variables $z_1, \ldots, z_n$. Taylor's theorem gives the power series expansion at $(x_1, \ldots, x_n)$

$$\begin{aligned} f(z_1, \ldots, z_n) = f(x_1, \ldots, x_n) + \sum_{i=1}^{n} \frac{\partial f}{\partial z_i}(x_1, \ldots, x_n)(z_i - x_i) \\ + \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \frac{\partial^2 f}{\partial z_i \partial z_j}(x_1, \ldots, x_n)(z_i - x_i)(z_j - x_j) + \cdots \end{aligned} \tag{5}$$

The first summation in equation (5) can be viewed as the dot product of two vectors

$$(\vec{z} - \vec{x}) = \langle z_1 - x_1, z_2 - x_2, \ldots, z_n - x_n \rangle$$
$$\frac{\partial f}{\partial \vec{z}}(\vec{x}) = \langle \frac{\partial f}{\partial z_1}(\vec{x}), \frac{\partial f}{\partial z_2}(\vec{x}), \ldots, \frac{\partial f}{\partial z_n}(\vec{x}) \rangle$$

where we have used $\vec{x}$ as an abbreviation for $(x_1, x_2, \ldots, x_n)$. If we treat the first of these vectors as a column vector and the second as a row vector the dot product is merely matrix multiplication.

Using this notation, the first two terms of Taylor's formula can be expressed as

$$f(\vec{z}) = f(\vec{x}) + \frac{\partial f}{\partial \vec{z}}(\vec{x}) \cdot (\vec{z} - \vec{x}) + \cdots.$$

(It is a good thing we do not need more terms.)

If we have several f's then each may be expanded as a power series like (5). If $(f_1, \ldots, f_m)$ is treated as a column vector we see that

$$\frac{\partial \vec{f}}{\partial \vec{z}} = \begin{pmatrix} \frac{\partial f_1}{\partial z_1} & \frac{\partial f_1}{\partial z_2} & \cdots & \frac{\partial f_1}{\partial z_n} \\ \vdots & & & \vdots \\ \frac{\partial f_n}{\partial z_1} & \frac{\partial f_n}{\partial z_2} & \cdots & \frac{\partial f_n}{\partial z_n} \end{pmatrix}.$$

This matrix is known as the *Jacobian* of $\vec{f}$, which we write as J. We then have

$$\vec{f}(\vec{z}) = \vec{f}(\vec{x}) + J(\vec{x}) \cdot (\vec{z} - \vec{x}) + \cdots$$

Now assume that $\vec{x}_k$ is a sequence of column vectors which converge to a solution of $\vec{f}(\vec{z})$. Further assume that $f(\vec{x}_k) \equiv 0 \pmod{\underline{m}^{k+1}}$. So the sequence of $\vec{x}_k$ is a coherent sequence which converges $\underline{m}$-adically to a zero of the system of equations. Then using (12) we have

$$0 \equiv \vec{f}(\vec{x}_k) \equiv \vec{f}(\vec{x}_{k-1}) + J(\vec{x}_{k-1}) \cdot (\vec{x}_k - \vec{x}_{k-1}) \qquad \pmod{\underline{m}^{k+1}}.$$

Again we are only interested in a linearly convergent iteration at this point. By the reasoning used in the univariate case we may replace $J(\vec{x}_{k-1})$ by $J(\vec{x}_0)$ so

$$\vec{x}_k - \vec{x}_{k-1} = -J^{-1}(\vec{x}_0) \cdot \vec{f}(\vec{x}_{k-1}).$$

This iteration is called the Hensel lifting algorithm. Notice that it is only applicable when the Jacobian is invertible. We state this as a theorem.

Theorem 3. *Let $\{ f_i(z_1, \ldots, z_n) \}_{i=1}^{n}$ be a system of a system of equations over $R_{\underline{m}}$, where $\underline{m}$ is an ideal of R. Let J be the Jacobian of $\{ f_i \}$. If $f_i(\hat{x}_0) \equiv 0 \pmod{\underline{m}}$ and if the determinant of $J(\hat{x}_0)$ is invertible over $R_{\underline{m}}/\underline{m}$ then there exist elements of $R_{\underline{m}}$, $\hat{x}$ such that $f_i(\hat{x}) = 0$, and they may be determined effectively.*

The initial solution of the system of equations, $\hat{x}_0$ is called the starting point of the iteration. If $J(\hat{x}_0)$ is invertible then $\hat{x}_0$ is said to be a *good starting point*. If the Jacobian matrix is not invertible, either because the $\hat{x}_0$ was a bad starting point or because the Jacobian itself was not square, then the system of linear equations which is to be solved at each step will not have a unique solution. There are cases where this can be useful, for instance as a sieve for possible solutions to diophantine equations over the integers. This idea has also been advanced by Lewis [Lew69] and Lauer [Lau78].

The quadratic form of the iteration uses $J(x_{k-1})$ in place of $J(x_0)$. Thus

$$x_{2^k} - x_{2^{k-1}} = -J(x_{2^{k-1}}) \cdot \vec{f}(x_{2^{k-1}}).$$

Unfortunately, this iteration requires that the inverse of a new Jacobian be computed at each stage of the iteration. Since $R/\underline{m}$ is field this is easy for the first step. Afterwards, rational expressions may appear and complicate the situation. Also note that $J(x_{2^k})$ is invertible in $R_{\underline{m}}/\underline{m}^{2^k}$ if and only if $J(x_0)$ is invertible.

2. Proof of Hensel's Lemma.

Hensel's lemma is usually formulated by investigators in algebraic manipulation in terms of the factoring problem. Last section has shown that when the Jacobian of a system of equations is invertible, it is possible to lift a solution of a system of equations modulo $\underline{m}$ to a solution in the $\underline{m}$-adic completion. The version of Hensel's lemma which is most commonly seen these days deals specifically with the problem of factoring polynomials. This version indicates that factorizations modulo $\underline{m}$ can be lifted to factorizations over the completion if the factors are relatively prime. More precisely,

Theorem 4. *(Hensel's Lemma) Let $f(X)$ be a monic polynomial over R, $\underline{m}$ an ideal of R. If there exist elements of $R/\underline{m}[X]$, $g(X)$ and $h(X)$, such that $f(X) - g(X)h(X)$ is an element of $\underline{m}R[X]$ and $g(X)$ and $h(X)$ are relatively prime, then $\hat{g}(X)$ and $\hat{h}(X) \in R_{\underline{m}}[X]$ exist such that $f(X) = \hat{g}(X)\hat{h}(X)$ and $\hat{g}(X) - g(X)$ and $\hat{h}(X) - h(X)$ are in $\underline{m}R[X]$.*

To understand this denote the coefficients of $f(X)$, $\hat{g}(X)$ and $\hat{h}(X)$ by f_i, g_j and h_k respectively. The f_i are known and the g_j and h_k are to be determined. Writing out the system of equations derived in section 1.1 we have

$$\begin{aligned} g_1 + h_1 &= f_1 \\ g_2 + g_1h_1 + h_2 &= f_2 \\ &\vdots \\ g_rh_{s-1} + g_{r-1}h_s &= f_{n-1} \\ g_rh_s &= f_n \end{aligned}$$

For the starting point we can use the coefficients of $g(X)$ and $h(X)$. So by theorem 2, it is possible to determine $\hat{g}(X)$ and $\hat{h}(X)$ if the Jacobian of this system of equations is invertible.

The Jacobian happens to be

$$\begin{pmatrix} 1 & 0 & 0 & 0 \ldots & 1 & 0 & 0 & \ldots \\ h_1 & 1 & 0 & 0 \ldots & g_1 & 1 & 0 & \ldots \\ h_2 & h_1 & 1 & 0 \ldots & g_2 & g_1 & 1 & \ldots \\ \vdots & & \vdots & \vdots & \vdots & \vdots & & \vdots \\ 0 & \ldots & & h_s & 0 & 0 & \ldots & g_r \end{pmatrix}.$$

The determinant of this matrix is also called the *resultant* of $g(X)$ and $h(X)$. It is known to be zero if and only if g and h have a nontrivial common factor. If the determinant is non-zero then we will be free to use the linear Hensel iteration as far as we please.

To see how this formulation actually works, consider factoring the polynomial

$$f(Z) = Z^6 + 67Z^5 + 41Z^4 + 2781Z^3 + 737Z^2 + 943Z + 253$$

over the rational integers. For this sort of problem, we choose $\mathfrak{m}$ to be an ideal generated by some rational prime. The initial point is determined by using the Berlekamp's algorithm [Ber70] for factoring polynomials over finite fields.

To get an idea what its factorization will be over the integers, $f(Z)$ is factored modulo the primes between 2 and 199. We discover that modulo 2, 3, 5, 17, 67, 71, 127 and 179, f is the product of two factors of degree 3. With other moduli f splits into more pieces. Careful examination of the 2 cubic factors shows that each has one term whose coefficient is zero. In fact this is true for all irreducible cubic factors for any modulus. Thus we conjecture that

$$f(Z) = (Z^3 + aZ^2 + b)(Z^3 + cZ + d).$$

While we would not expect an implementation of a general purpose factoring algorithm to apply this trick, we will use it for two reasons. First, it simplifies the equations which are developed somewhat, and second, this example will also indicate how appropriate additional knowledge about the problem to be solved can be utilized. Multiplying the candidate factorization of $f(Z)$ out we are led to the following system of equations

$$\begin{aligned} a &= 67 \\ c &= 41 \\ b + ac + d &= 2781 \\ ad &= 737 \\ bc &= 943 \\ bd &= 253 \end{aligned} \tag{6}$$

Since we knew (or at least hoped) that two of the terms in the factorization were zero, there are two more equations than unknowns. In fact, the equations are now trivial to solve. To make things a bit more interesting we will ignore the first two equations. We use 5 as the modulus, i.e., $\mathfrak{m} = (5)$. Berlekamp's algorithm gives $f(Z) = (Z^4 + 2Z - 2)(Z^2 + Z + 1)$, so for initial approximations we have $a_0 = 2$, $b_0 = -2$, $c_0 = 1$ and $d_0 = 1$. The inverse of the Jacobian of the system of equations is

$$J^{-1} = \begin{pmatrix} c_0 & 1 & a_0 & 1 \\ d_0 & 0 & 0 & a_0 \\ 0 & c_0 & b_0 & 0 \\ 0 & d_0 & 0 & b_0 \end{pmatrix}^{-1} = \begin{pmatrix} -1 & 0 & -1 & 2 \\ 1 & -1 & 1 & 2 \\ -2 & 2 & 1 & 1 \\ -2 & 2 & -2 & -1 \end{pmatrix}.$$

To compute the next stage, we first determine the correction terms which were denoted by $\tilde{f}(\bar{x}_{k-1})$ in the previous section. To do this, the equations in (6) are evaluated at the starting point modulo 25. We then get a column vector which is multiplied by $-J^{-1}$

$$\begin{pmatrix} a_1 \\ b_1 \\ c_1 \\ d_1 \end{pmatrix} - \begin{pmatrix} a_0 \\ b_0 \\ c_0 \\ d_0 \end{pmatrix} = -J^{-1} \cdot \begin{pmatrix} -5 \\ -10 \\ 5 \\ -5 \end{pmatrix} = \begin{pmatrix} -10 \\ 0 \\ -10 \\ 10 \end{pmatrix}.$$

Adding this to the term starting point, we have $a_1 = -8$, $b_1 = -7$, $c_1 = -9$ and $d_1 = 11$. Repeating this process,

$$\begin{pmatrix} a_2 \\ b_2 \\ c_2 \\ d_2 \end{pmatrix} - \begin{pmatrix} a_1 \\ b_1 \\ c_1 \\ d_1 \end{pmatrix} = -J^{-1} \cdot \begin{pmatrix} 50 \\ 50 \\ -50 \\ -25 \end{pmatrix} = \begin{pmatrix} -50 \\ 25 \\ 50 \\ 0 \end{pmatrix}.$$

So $a_2 = -58$, $b_2 = 23$, $c_2 = 41$ and $d_2 = 11$. Repeating this iteration one last time we are led to the final factorization

$$f(X) = (X^3 + 67X^2 + 23)(X^3 + 41X + 11).$$

3. Zassenhaus' Formulation of Hensel's Lemma

The "old" version of Hensel's lemma was first proposed by Zassenhaus [Zas69]. Wang and Rothschild [Wan75] and Musser [Mus75] utilized Zassenhaus' ideas in their factoring algorithm. Using the ideas of Moses and Yun [Mos75], Yun [Yun73, Yun76] investigated the general applicability of Hensel's lemma to problems in algebraic manipulation.

As formulated by Yun [Yun76], Zassenhaus' version of Hensel's lemma differs from ours in that the number of equations produced is smaller than the number of variables. For instance, in the factoring problem, the Zassenhaus approach determines G and H by solving the equation

$$f(X) - GH = 0 \tag{7}$$

It is only when the solution of this equation is restricted to the ring $\mathbf{Z}_p[X]$ that (7) has a unique solution. By using a p-adic technique the non-linear diophantine equation (7) can be reduced to a series of easy to solve, linear equations. Then by piecing together the solutions of these linear equations, G and H can be determined. The procedure used is not very complex. Here we will outline the main points and demonstrate the connection with our formalism.

Let $f(X)$ be a univariate polynomial over the integers, and assume that it has two irreducible factors, G and H. We know that G_0 and H_0 are factors modulo p and we want to lift them to G and H. Writing G and H p-adically

$$\begin{aligned} f(X) &= (G_0 + G_1 p + \cdots)(H_0 + H_1 p + \cdots) \\ &= G_0 H_0 + (H_1 G_0 + G_1 H_0) + \cdots \end{aligned}$$

We know that $f(X) - G_0 H_0$ is a multiple of p, so dividing by p we have

$$H_1 G_0 + G_1 H_0 = (f(X) - G_0 H_0)/p \qquad (\text{mod } p) \tag{8}$$

This is a linear diophantine equation in G_1 and H_1. Its solution is obtained by first solving $A_iG_0 + B_iH_0 = X^i \pmod{p}$ for various i by the Euclidean algorithm. Since the left hand side of (8) is a polynomial in X, G_1 and H_1 can be determined by adding up the appropriate B_i and A_i respectively.

Having computed G_1 and H_1, the other terms may be computed similarly. For a linear iteration, the process continues as follows:

$$f(X) = (G_0 + G_1p + G_2p^2 + \cdots)(H_0 + H_1p + H_2p^2 + \cdots)$$

Considering this equation modulo p^3

$$(H_2G_0 + G_2H_0) = \bigl(f(Z) - (G_0 + G_1p)(H_0 + H_1p)\bigr)/p^2 \qquad \pmod{p}$$

The right hand side of this equation is again a polyonomial and the left hand side is essentially the same as (8). So using the A_i and B_i obtained in the previous step we can compute G_2 and H_2.

As an illustration consider the polynomial which we factored in the last section

$$f(Z) = Z^6 + 67Z^5 + 41Z^4 + 2781Z^3 + 737Z^2 + 943Z + 253$$

As before we know that its factorization modulo 5 is

$$= (Z^3 + 2Z^2 - 2)(Z^3 + Z + 1).$$

So we now want to solve

$$\begin{aligned} H_1G_0 + G_1H_0) &= f(Z) - (Z^3 + 2Z^2 - 2)(Z^3 + Z + 1) \\ &= -10Z^5 - 10Z^4 + 5Z^3 + 10Z^2 - 5Z + 5 \end{aligned} \tag{9}$$

modulo 5^2.

Since $G_0 = Z^3 + 2Z^2 - 2$ and $H_0 = Z^3 + Z + 1$ the diophantine equations we need to solve are

$$A_iG_0 + B_iH_0 = A_i(Z^3 + 2Z^2 - 2) + B_i(Z^3 + Z + 1) = Z^i \tag{10}$$

where we require $\deg B_i < \deg G_0 = \deg G$ and $\deg A_i < \deg H_0 = \deg H$. This condition on *the degrees of A_i and B_i we call the degree constraint.* As we shall see, there is only one solution to (10) which also satisfies the degree constraint.

A_0 and B_0 can be determined by the usual application of the Euclidean algorithm to linear diophantine equations [Knu69]. We begin by computing the continued fraction of H_0/G_0. Then the convergents of the continued fraction are computed; the next to last one being the one which we want. Following the usual schema for continued fraction computations, we have:

		1	$2Z$	$2Z$	$-2Z+1$
0	1	1	$2Z+1$	$-Z^2+2Z+1$	$2Z^3+2Z+2$
1	0	1	$2Z$	$-Z^2+1$	$2Z^3-Z^2+1$

The first line consists of the "partial quotients" of the continued fraction (the quotients in the PRS) and the next two lines are the convergents of the continued fraction. So we have $A_0 = 2Z^2 + Z - 2$ and $B_0 = -2Z^2 + 2Z$ after removing a unit and multiplying B_0 by -1.

The other A_i and B_i can be computed from A_0 and B_0 by a simple device. Notice that while $Z^i A_0$ and $Z^i B_0$ probably do not satisfy the degree constraints they do satisfy equation (10). Pick Q and A_i such that $Z^i A_0 = H_0 Q + A_i$ and $\deg A_i < \deg H_0$. Now define $B_i = Z^i B_i + QG_0$. A_i and B_i will also satisfy (10) and now A_i also satisfies the degree constraint. It is not hard to see that B_i also satisfies the degree constraint. Let $\tilde{A}_i$ and $\tilde{B}_i$ be solutions of (10) which are of the appropriate degrees. Then

$$(A_i - \tilde{A}_i)G_0 + (B_i - \tilde{B}_i)H_0 = 0 \qquad (\text{mod } 5)$$

Remember that G_0 and H_0 are relatively prime. The degree of A_i and $\tilde{A}_i$ is less than $\deg H_0$; thus the degree of $B_i - \tilde{B}_i$ must be less than $\deg G_0$. Since $\tilde{B}_i$'s degree is small, so is B_i. Since G_0 and H_0 relatively prime $A_i = \tilde{A}_i$ and $B_i = \tilde{B}_i$. Thus not only do A_i and B_i have the correct degrees they form the unique solution of (10) for a given i.

Repeating this procedure we easily get the following table for the basis of the solutions of linear diophantine equations of the form $AG_0 + BH_0 = C$.

$$\begin{array}{ll}
A_0 = 2Z^2 + Z - 2 & B_0 = -2Z^2 + 2 \\
A_1 = Z^2 + Z - 2 & B_1 = -Z^2 + 2Z + 1 \\
A_2 = Z^2 + 2Z - 1 & B_2 = -Z^2 + Z - 2 \\
A_3 = 2Z^2 - 2Z - 1 & B_3 = -2Z^2 - 2Z - 2 \\
A_4 = -2Z^2 + 2Z - 2 & B_4 = 2Z^2 - 2Z + 1 \\
A_5 = 2Z^2 + 2 & B_5 = -Z^2 + Z - 1
\end{array}$$

Dividing equation (9) by p gives

$$H_1 G_0 + G_1 H_0 = -2Z^5 - 2Z^4 + Z^3 + 2Z^2 - Z + 1 \qquad (\text{mod } 5)$$

To compute H_1 we use the linear basis for the solutions which were just determined.

$$\begin{aligned} H_1 &= -2A_5 + -2A_4 + A_3 + 2A_2 - A_1 + A_0 \\ &= -2Z + 2 \end{aligned}$$

One factor of $f(Z)$, computed to second order, is $H_0 + 5H_1 = Z^3 - 9Z + 11$ as we determined earlier using our formulation in the previous section.

Thus far, the liftings considered have been at a principal ideal. This restriction can be lifted without much difficulty, but then this old formulation becomes more complex. The complications involved also obscure some of the relationships with our formulation. These modifications are contained in a paper of Yun [Yun76] and the references contained therein. Since we feel that our formulation is easier to understand, more powerful and easier to implement than this version of Hensel's lemma we will not discuss these improvements.

The Zassenhaus version of Hensel's lemma, though somewhat more complex, than our version still produces the same answer--the correct factorization though each of the lifting stages. In the following paragraphs we will demonstrate that this is not a fortuitous accident but is due to the fact that both formulations are performing the same computation but in a slightly different manner.

The key to using Hensel's lemma to lift a factorization $F = G_0 H_0$ (mod $\mathfrak{m}$) to the $\mathfrak{m}$-adic completion is solving the diophantine equations

$$A_i G_0 + B_i H_0 = Z^i \qquad (\text{mod } \mathfrak{m}) \tag{11}$$

Once we have obtained the A_i and B_i, it is merely a matter of computing the error introduced when a factorization (mod $\underline{m}^{k+1}$). This error is then used to produce the appropriate linear combination of the A_i and B_i which is G_{k+1} and H_{k+1}.

Our formulation is quite similar. We begin by inverting the Jacobian matrix. Through the Newton-Raphson formula we combine the error terms to compute G_{k+1} and H_{k+1}. Structurally, these two algorithms are quite similar. Actually they are identical. We shall see that the solutions of (11) form the inverse of the Jacobian matrix.

To see this let,

$$G_0 = Z^r + g_1 Z^{r-1} + g_2 Z^{r-2} + \cdots + g_r$$
$$H_0 = Z^s + h_1 Z^{s-1} + h_2 Z^{s-2} + \cdots + h_s$$

so the Jacobian matrix is

$$J = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 1 & 0 & 0 & \cdots & 0 \\ h_1 & 1 & 0 & \cdots & 0 & g_1 & 1 & 0 & \cdots & 0 \\ h_2 & h_1 & 1 & \cdots & 0 & g_2 & g_1 & 1 & \cdots & 0 \\ \vdots & & \vdots & & & \vdots & & \vdots & & \vdots \\ 0 & \cdots & 0 & h_s & h_{s-1} & 0 & \cdots & 0 & g_r & g_{r-1} \\ 0 & \cdots & 0 & 0 & h_s & 0 & \cdots & 0 & 0 & g_r \end{pmatrix}$$

Consider what happens when this matrix is multiplied by a column vector

$$J \cdot \begin{pmatrix} b_0 \\ \vdots \\ b_{s-1} \\ a_0 \\ \vdots \\ a_{r-1} \end{pmatrix} = \begin{pmatrix} a_0 + b_0 \\ a_0 g_1 + a_1 + b_1 + b_0 h_1 \\ \vdots \\ a_{r-1} g_{r-1} + a_{r-2} g_r + b_{s-2} h_s + b_{s-1} h_{s-1} \\ a_{r-1} g_r + b_{s-1} h_s \end{pmatrix}$$

The $(r + s - 1) - i$ row of this column vector is clearly the coefficient of Z^i in $AG_0 + BH_0$ where

$$A = a_0 Z^{s-1} + a_1 Z^{s-2} + \cdots + a_{s-1},$$
$$B = b_0 Z^{r-1} + b_1 Z^{r-2} + \cdots + b_{r-1}.$$

Thus the computation of the inverse of the Jacobian of the system of equations is a clever way of obtaining all the solution of (11). Or, as we'd prefer to view it, solving (11) is a clever way of inverting a very special type of Jacobian matrix.

Chapter VII

The Sparse Hensel Algorithm

The last chapter presented our formulation of Hensel's lemma as well another formulation popularized by Zassenhaus. The univariate polynomial factorization example that was used to illustrate the algorithms did not point out their major failing since this inadequacy only appears in multivariate problems. The purpose of this chapter is to demonstrate that the same techniques that were used in our version of the modular algorithm can be applied to our formulation of the Hensel. Significantly, we are led to a class of probabilistic p-adic lifting algorithms that run in polynomial time.

To illustrate the difficulties inherent in multivariate problems we will again consider the polynomial factorization problem. Recall that the key idea is to convert a problem of factoring a polynomial in the polynomial ring $R[X, X_2, \ldots, X_v]$ to one of finding a solution (in $R[X_2, \ldots, X_v]$) of a system of equations. To apply Newton's method we attempt to find solutions which lie in a completion of $R[X_2, \ldots, X_v]$. Usually, the particular completion considered is $\underline{\mathfrak{m}} = (X_2, \ldots, X_v)$.

Consider the problem of factoring the square free polynomial

$$\begin{aligned}F(X, X_2, X_3) = {} & X^4 + (X_2X_3^4 + X_2X_3 + X_2^3 + 2)X^3 \\ & + (X_2^2X_3^5 + 2X_2X_3^4 - 3X_2^5X_3 + X_2^4X_3 + 3X_2^3)X^2 \\ & + (-3X_2^6X_3^5 + X_2X_3^4 - 3X_2^8X_3 + X_2^4X_3 - X_2X_3 + 3X_2^3 - 2)X \\ & + (-3X_2^8X_3 + 3X_2^5X_3 + X_2^3 - 1).\end{aligned}$$

By picking a few values for X_2 and X_3, we can quickly convince ourselves that unless F is irreducible, it factors into 2 quadratic polynomials.

$$\begin{aligned}F(X, 0, 0) &= (X - 1)(X + 1)^3 \\ F(X, -1, 1) &= (X^2 - 2X - 2)(X^2 + X + 4) \\ F(X, 3, 5) &= (X^2 + 17X - 3644)(X^2 + 1902X + 26) \\ F(X, -5, 7) &= (X^2 - 12130X - 126)(X^2 - 33X + 65626)\end{aligned}$$

Notice that even though $F(X, 0, 0)$ splits into linears, there is only one way to combine the linears into a factorization into two quadratics. Letting the factorization be $F(X) = (X^2 +$

$AX + B)(X^2 + CX + D)$ we get the following system of equations

$$\begin{aligned} A + C &= (X_2X_3^4 + X_2X_3 + X_2^3 + 2) \\ AC + B + D &= (X_2^2X_3^5 + 2X_2X_3^4 - 3X_2^5X_3 + X_2^4X_3 + 3X_2^3) \\ AD + BC &= (-3X_2^6X_3^5 + X_2X_3^4 - 3X_2^8X_3 + X_2^4X_3 - X_2X_3 + 3X_2^3 - 2) \\ BD &= (-3X_2^8X_3 + 3X_2^5X_3 + X_2^3 - 1) \end{aligned} \tag{1}$$

We are looking for elements A, B, C and D of $R[X_2, X_3]$ that satisfy this system of equations.

Using Newton's method, the solution to (1) will be generated as a multivariate power series in X_2 and X_3. This solution is developed one term at a time, first the constant term, then the linear terms, and so on. To get the initial approximation, we would like to set X_2 and X_3 to zero and solve the resulting simpler system. This system would usually be solved by factoring $F(X, 0, 0)$. In this case (1) reduces to the simple system

$$\begin{aligned} A + C &= 2, \\ AC + B + D &= 0 \\ AD + BC &= -2 \\ BD &= -1 \end{aligned}$$

From the univariate factorization we see that this system has two solutions, either $(A, B, C, D) = (0, -1, 2, 1)$ or $(2, 1, 0, -1)$. This corresponds to the commutativity of the factors. We will choose the $A = 0$ solution. The other solution will give a similar result, but with A and C, and B and D interchanged. Newton's iteration for systems of equations takes the form

$$\vec{x}_k = \vec{x}_{k-1} - J^{-1}(\vec{x}_0) \cdot \vec{f}(\vec{x}_{k-1}),$$

where J is the Jacobian matrix of the system. Let the column vector

$$\vec{x}_0 = \begin{pmatrix} 0 \\ -1 \\ 2 \\ 1 \end{pmatrix}$$

denote the initial approximation, and the column vector

$$\vec{f}(A, B, C, D) = \begin{pmatrix} A + C - f_1 \\ AC + B + D - f_2 \\ AD + BC - f_3 \\ BD - f_4 \end{pmatrix}$$

represent the system of equations which is to be solved. We have used f_i as an abbreviation for the coefficient of X^{1-i} in $F(X, X_2, X_3)$.

For the system of equations in (1) the Jacobian matrix is

$$J = \begin{pmatrix} 1 & 0 & 1 & 0 \\ C & 1 & A & 1 \\ D & C & B & A \\ 0 & D & 0 & B \end{pmatrix}.$$

When evaluated at $X_2 = 0$ and $X_3 = 0$, the Jacobian is

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 1 \\ 1 & 2 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix}.$$

The next step is to invert this matrix. Unfortunately, it is singular. Thus we will not be able to lift the factorization at the ideal $(X_2 - 0, X_3 - 0)$ and must try some other ideal.

If we try $(X_2 + 1, X_3 - 1)$ then we get

$$\begin{aligned} F(X, -1, 1) &= X^4 - X^3 - 10X - 8 \\ &= (X^2 - 2X - 2)(X^2 + X + 4) \end{aligned}$$

The corresponding Jacobian matrix is

$$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & -2 & 1 \\ 4 & 1 & -2 & -2 \\ 0 & 4 & 0 & -2 \end{pmatrix},$$

which is invertible. The problem with using this ideal is the factorization will be developed in a multivariate power series in $X_2 + 1$ and $X_3 - 1$. Since the actual factorization is

$$\begin{aligned} F(X, X_2, X_3) = &(X^2 + (X_2X_3 + 2)X - 3X_2^5X_3 + 1) \\ &(X^2 + (X_2X_3^4 + X_2^3)X + X_2^3 - 1), \end{aligned}$$

these factors will have 16 and 17 terms respectively, when written in terms of $X_2 + 1$ and $X_3 - 1$. If there are more variables, the number of terms produced will increase exponentially.

The solution for the Hensel algorithm, just as for the modular algorithm, is to perform the lifting one variable at a time. We replace the X_i by $Y_i + a_i$. After performing the lifting for Y_i, the answer will be dense. But after converting it back to a polynomial in X_i the answer can be no more dense than the final answer. Section 1 presents the details of the sparse Hensel algorithm using the formalism developed in the last two chapters. Section 2 points out a number of "tricks" which can be used to great advantage in an implementation of the sparse Hensel algorithm.

1. Introduction of Sparse Techniques

Following this procedure, we will solve (1) for a power series in X_2. Since X_3 will be 1 throughout this portion of the computation, we can simplify (1) to

$$\begin{aligned} A + C &= (2X_2 + 3) \\ AC + B + D &= (-3X_2^5 + X_2^4 + 3X_2^3 + X_2^2 + 2X_2) \\ AD + BC &= (-3X_2^8 - 3X_2^6 + X_2^4 + 3X_2^3 - 2) \\ BD &= (-3X_2^8 + 3X_2^5 + X_2^3 - 1) \end{aligned}$$

Since we will be working with power series in $X_2 + 1$ it will simplify some things to write these equations in terms of $Y_2 = X_2 + 1$

$$\begin{aligned} A + C &= Y_2^3 - 3Y_2^2 + 5Y_2 - 1 \\ AC + B + D &= -3Y_2^5 + 16Y_2^4 - 31Y_2^3 + 28Y_2^2 - 10Y_2 \\ AD + BC &= -3Y_2^8 + 24Y_2^7 - 87Y_2^6 + 186Y_2^5 - 254Y_2^4 \\ &\quad + 227Y_2^3 - 132Y_2^2 + 47Y_2 - 10 \\ BD &= -3Y_2^8 + 24Y_2^7 - 84Y_2^6 + 171Y_2^5 - 225Y_2^4 \\ &\quad + 199Y_2^3 - 117Y_2^2 + 42Y_2 - 8 \end{aligned} \tag{2}$$

To simplify the computations, we will again work with a rather small modulus, this time 163. The inverse of the Jacobian is then

$$-J^{-1} = \begin{pmatrix} 72 & 18 & 18 & -9 \\ 36 & 36 & -18 & 36 \\ -73 & -18 & -18 & 9 \\ 72 & 72 & -36 & -9 \end{pmatrix}.$$

Subtracting the left hand side of (2) from the right, and using the initial approximations $A = -2$, $B = -2$, $C = 1$ and $D = 4$, we see the first term of the error is

$$\begin{pmatrix} -5Y_2 \\ 10Y_2 \\ -47Y_2 \\ -42Y_2 \end{pmatrix}.$$

Notice that in selecting the error terms we ignored all but the lowest order ones since only they will affect the correction. When multiplied by $-J^{-1}$, we have the first correction term

$$\begin{pmatrix} 72 & 18 & 18 & -9 \\ 36 & 36 & -18 & 36 \\ -73 & -18 & -18 & 9 \\ 72 & 72 & -36 & -9 \end{pmatrix} \cdot \begin{pmatrix} -5Y_2 \\ 10Y_2 \\ -47Y_2 \\ -42Y_2 \end{pmatrix} = \begin{pmatrix} 4Y_2 \\ 3Y_2 \\ Y_2 \\ -15Y_2 \end{pmatrix}.$$

To compute the next correction term, we replace A, B, C and D by $-2 + 4Y_2$, $-2 + 3Y_2$, $1 + Y_2$ and $4 - 15Y_2$ respectively in (2). The only nonzero terms should be of degree 2 or greater in Y_2. Continuing this iteration we discover that

$$\begin{aligned} A &= -2 + 4Y_2 - 3Y_2^2 + Y_2^3 \\ B &= -2 + 3Y_2 - 3Y_2^2 + Y_2^3 \\ C &= 1 + Y_2 \\ D &= 4 - 15Y_2 + 30Y_2^2 - 30Y_2^3 + 15Y_2^4 - 3Y_2^5 \end{aligned}$$

Reexpressing these solutions in terms of X_2 we have

$$\begin{aligned} A &= X_2^3 + X_2 \\ B &= X_2^3 - 1 \\ C &= X_2 + 2 \\ D &= -3X_2^5 + 1 \end{aligned}$$

So our factorization thus far is

$$F(X, X_2, 1) = (X^2 + (X_2^3 + X_2)X + X_2^3 - 1)((X^2 + (X_2 + 2)X - 3X_2^5 + 1)$$

Now we need to construct a system of equations for the coefficients of X_2 so A, B, C and D can be lifted to polynomials in X_3. We now know that $A(X_3 = 1) = X_2^3 + X_2$. We now make our standard probabilistic assumption, that $A = \alpha X_2^3 + \beta X_2$ where α and β are polynomials in X_3. Continuing we have

$$\begin{aligned} A &= \alpha X_2^3 + \beta X_2 \\ B &= \gamma X_2^3 + \delta \\ C &= \epsilon X_2 + \varsigma \\ D &= \eta X_2^5 + \theta \end{aligned}$$

Consider the first equation of (1) after having made these replacements.

$$A + C = \alpha X_2^3 + \beta X_2 + \epsilon X_2 + \varsigma = (X_2 X_3^4 + X_2 X_3 + X_2^3 + 2)$$

Since the Greek letter variables do not involve X_2 we are free to equate the coefficients of the powers of X_2. This gives the following equations

$$\begin{aligned} \alpha &= 1 \\ \beta + \epsilon &= X_3^4 + X_3 \\ \varsigma &= 2. \end{aligned}$$

Doing this with the other equations, we have

$$\begin{aligned} \eta &= -3X_3 & \alpha\epsilon &= X_3 \\ \alpha\varsigma + \gamma &= 3 & \beta\epsilon &= X_3^5 \\ \alpha\eta &= -3X_3 & \eta\beta &= -3X_3^5 \\ \gamma\epsilon &= X_3 & \alpha\theta + \gamma\varsigma &= 3 \\ \beta\theta + \delta\epsilon &= X_3^4 - X_3 & \delta\varsigma &= -2 \\ \gamma\eta &= -3X_3 & \gamma\theta &= 1 \\ \delta\theta &= -1 & \alpha &= 1 \\ \beta + \epsilon &= X_3^4 + X_3 & \varsigma &= 2 \end{aligned} \tag{3}$$

These equations can be solved by the same technique that we used before, or, as we point out in section 3, they can be solved by inspection. In the rest of this section we will try to make more precise the technique that was just used in factoring F.

As this example shows, the Hensel algorithm is somewhat more complex than the modular algorithm. Unfortunately, this algorithm also has a reputation among system implementors as being one of the more difficult algorithms to implement. Though this may be true of the old formalism, our new formalism is quite simple and to a very large degree reduces the duplication of code which seems to have been necessary in the older implementations we have examined. The program that solves systems of algebraic equations using p-adic techniques is just a page or two of code. Interfacing this code to any particular problem, such as computing GCD's, factoring polynomials and computing square free decompositions is merely a matter of setting up the appropriate system of equations.

The first algorithm we present is the univariate version of Newton's iteration for systems of equations. It is fairly straightforward and should present no problems.

Algorithm U implements the univariate version of the sparse Hensel algorithm. It assumes that $g_1, \ldots, g_n$ are unknowns, X is a variable and $f_1, \ldots, f_m = \vec{f}$ is a system of polynomials in $X, g_1, \ldots, g_n$, $m \geq n$. It is also given a point $X = a$ and an initial solution $\bar{g}_1, \ldots, \bar{g}_n$ such that $f_i(\bar{g}_1, \ldots, \bar{g}_n) = 0 \pmod{X - a}$. Finally it is given an integer k. This algorithm returns polynomials $G_1, \ldots, G_n$ such that $f_i(G_1, \ldots, G_n) \equiv 0 \pmod{(X - a)^{k+1}}$.

```
U1. [Jacobian] Compute the Jacobian of the system $\vec{f}$ at the point $X = a$.
U2. [Make it square] Select $n$ linearly independent rows of the Jacobian and call the resulting
    matrix $J$. Renumber the $f_i$ so $f_i$ corresponds to the $i$th row of $J$.
U3. [Invert] Invert the Jacobian to produce $J^{-1}$.
U4. [Shift system to origin] Replace $X$ by $Y + a$ in the $f_i$. (Now $f_i(\bar{g}_1, \ldots, \bar{g}_n) \equiv 0 \pmod{Y}$.)
    Set $G_1 \leftarrow \bar{g}_1, \ldots, G_n \leftarrow \bar{g}_n$.
U5. [Iterate over number of desired terms] For $j = 1$ through $k$ do steps U6 and U7.
    U6. [Iterate over number of equations] For $i = 1$ through $k$ set $E_i$ to the $Y^i$-term of
        $f_i(G_1, \ldots, G_n)$.
    U7. [Newton's iteration!] For $i = 1$ through $n$ set $\vec{G} \leftarrow \vec{G} + J^{-1} \cdot \vec{E}$.
U8. [Shift back to old variables] Replace $Y$ by $X - a$ in $G_1, \ldots, G_n$.
U9. [Done] Return $G_1, \ldots, G_n$.
```

The multivariate portion of this algorithm is the part that takes advantage of the sparsity in the system. It uses algorithm U to compute the the solution in one variable of the system of equations. It then uses this solution to create a new system of equations where the new unknowns are the coefficients of the polynomials which were just determined by algorithm U. The variables $S_1, \ldots, S_n$ are the solutions to the original system of equations, as they are improved with the introduction of each new variable.

Algorithm M does a multivariate Hensel lifting. It takes as input the unknowns $g_1, \ldots, g_n$, variables $X_1, \ldots, X_v$ and a system of polynomials $f_1, \ldots, f_m$ in the g_i and X_j, $m \geq n$. It is also given a list of evaluation points $a_1, \ldots, a_v$ and initial values for the variables $\bar{g}_1, \ldots, \bar{g}_n$ such that $f_i(\bar{g}_1, \ldots, \bar{g}_n) \equiv 0 \pmod{(X_1 - a_1, \ldots, X_v - a_v)}$.

```
M1. [Initialize] Initialize the variables $S_1, \ldots, S_n$ to the symbols $g_1, \ldots, g_n$ respectively.
M2. [Loop through variables] For $i = 1$ through $v$ do M3 up to M6.
    M3. [Remove most of the variables] Replace $X_{i+1}, \ldots, X_v$ in the $f_i$ by $a_{i+1}, \ldots, a_v$ to get
        polynomials $F_1, \ldots F_m$
    M4. [Solve univariate problem] Use algorithm U on $\vec{g}_i, X_i, \vec{f}_i, a_i$ and $\vec{g}_i$ to get solutions to
        $\vec{G}_i$.
    M5. [Update the original solution] Replace the symbols $g_i$ in $S_1, \ldots, S_n$ by the values of
        the $\vec{G}_i$.
    M6. [New equations] For each $f_i$ replace the $g_i$ by $\vec{G}_i$. The result is a polynomial in $X_i$.
        The coefficients are the new $f_i$.
M7. [Done] Return the $S_i$ as the solution.
```

2. Some Ideas for Implementations

In this section we will consider a couple of useful heuristics that can often dramatically speed up the running time of the sparse Hensel algorithm. However there are problems for which these heuristics do not help, so they do not affect the analysis of the running time of the algorithm. These heuristics were first used by Paul Wang [Wan78].

Recall the system of equations (2) that was encountered in the example at the beginning of this chapter. Ignoring several of the equations, and reordering them somewhat we have

$$\begin{array}{lll} \eta = -3X_3 & \alpha\eta = -3X_3 & \gamma\eta = -3X_3 \\ \eta\beta = -3X_3^5 & \alpha\epsilon = X_3 & \gamma\theta = 1 \\ & \delta\theta = -1 & \end{array}$$

Reading these equations horizontally, the values for the other variables just drop out.

In general this heuristic proceeds as follows: (1) Isolate all linear equations with one unknown. (2) In the system of equations to be solved, replace all occurrences of the unknowns just determined by their values. (3) Repeat until there are no linear equations with one unknown.

This procedure is very cheap; the only point that could be expensive is the substitutions in step (2). Since this procedure can be performed before the variable being lifted is shifted away from zero, step (2) can be no more expensive than verifying the sparse form of the answer.

In section 3 we will analyze the sparse Hensel algorithm. Here we only discuss the probability that this heuristic will actually be beneficial. It is rather difficult to determine precisely what the probability that this procedure will be successful is, but we can make some rough estimates in the factorization problem with the following observation. Assume that we wish to factor F, a polynomial in v variables. Assume F has 2 monic irreducible factors

$$F = (M_0 + m_1 + \cdots + m_r)(M_1 + m_{r+1} + \cdots + m_t)$$

where M_i are both monic monomials and m_i are monomials whose coefficients are to be determined. If the exponent vectors of all of the m_i are distinct we will be able to determine their coefficients by the procedure just outlined.

Assume that the degree of each variable in the monomials, m_i, is bounded by D. Then there are $N = (D+1)^v$ possible exponent vectors. There are N^t ways t monomials could be chosen (we are ignoring the fact that some are permutations of the others) and $N!/(N-t)!$ ways t different monomials could be chosen (again disregarding the permutations). Therefore the the probability that the system of equations is "presolved" is $P(t,v,D)$

$$\begin{aligned} P(t,v,D) &\geq \frac{\frac{N!}{(N-t)!}}{N^t} = \frac{N!}{(N-t)!N^t} \\ &\approx \sqrt{\frac{N}{N-t}} \cdot \frac{(\frac{N}{e})^t}{(\frac{N-t}{e})N^t} \end{aligned}$$

by Stirling's approximation. Simplifying this somewhat

$$\approx \sqrt{\frac{N}{N-t}} \cdot e^{-t}\left[1 + \frac{t}{N-t}\right]^{N-t} \tag{4}$$

Notice that as $N-t$ goes to infinity, holding t fixed, the quantity in square brackets tends towards e^t. So for large N, $P(t,v,D)$ is quite close to 1. Computing the Taylor series expansion of (4) at $N = \infty$ we find

$$P(t,v,D) \leq 1 - \frac{1}{2}\frac{t^2-t}{N} + \ldots.$$

So the chances that this technique will *not* be useful is somewhat greater than

$$\frac{1}{2}\frac{t^2-t}{(D+1)^v}.$$

As the number of variables increases this becomes very small. In fact for

$$v > \frac{\log t^2 - t}{\log(D+1)} \approx 2\frac{\log t}{\log(D+1)}$$

we would expect the equations to always be presolved.

Wang uses a technique that is essentially equivalent to this in his implementation of the EEZ algorithm. To illustrate how dramatically it can affect the running time of the Hensel algorithm have timed four different algorithms on the first set of examples given in the appendix. These examples were run on the MACSYMA [MAC77] algebraic manipulation system at Massachusetts Institute of Technology which runs on a Digital Equipment Corporation KL-10 processor. The first column is the EZ GCD algorithm given by Moses and Yun [Mos73]. The second column, labeled EEZ2 gives timings using Wang's new EEZ GCD algorithm [Wan79], but without using the heuristic described in this section. The second column, labeled EEZ, is Wang's algorithm *using* the heuristic. The final column is the sparse modular algorithm described in this thesis. All the times given are in seconds.

The appendix lists three sets of polynomials, f_i, g_i and d_i. The four algorithms were used to compute the GCD of $f_i d_i$ and $g_i d_i$ after these two products were multiplied out.

v	EZ	EEZ2	EEZ	Sparse Mod
1	.036	.040	.058	.040
2	.277	.389	.416	.160
3	.431	.785	.537	.381
4	1.288	1.224	.704	.842
5	3.128	7.331	1.410	1.825
6	*	7.428	1.966	3.364
7	*	10.282	1.628	4.190
8	*	*	2.446	4.534
9	*	*	2.346	4.006
10	*	*	2.832	8.202

The GCD computed here had 4 terms and the degree of each variable was 3 or less. The cofactors had the same parameters. Considered as a factoring problem, we would have $t = 10$, $D = 3$. So we would expect the timings to level off at about

$$v = \frac{2\log t}{\log(D+1)} = \frac{2\log 10}{\log 4} = 3.32$$

In fact the timings level off between 5 and 6. That's not too far off.

In the final section we give some timings for problems for which all algorithms but the ones described in this thesis fail.

3. Analysis of the Sparse Hensel Algorithm

We will assume we are using the sparse Hensel algorithm to lift a solution $\vec{x}_0 = x_0^{(1)} \cdots x_0^{(n)}$ of the system of equations $\vec{f}_1(\vec{z}_1) = 0$ modulo $\underline{m} = (X_1 - a_1, \ldots, X_v - a_v)$ to a solution $\vec{x}_d$ modulo $\underline{m}^{d+1}$. We will let t denote the maximum of the number of terms in any $x_k^{(i)}$ for $i = 1$ through n. In keeping with our concern for sparse polynomials we assume that $t \ll (d+1)^v$.

The solutions modulo $\underline{m}^{d+1}$ are determined by lifting the solutions we have modulo $\underline{m}$ to solutions modulo $\underline{m}_1 = (X_1 - a_1)^{d+1}\underline{m}$, which are then lifted to solutions modulo $m_2 = (X_2 - a_2)^{d+1}\underline{m}_1$ and so on. Since we pass from a solution modulo $\underline{m}_i$ to one modulo $\underline{m}_{i+1}$ we will introduce a new system of equations and unknowns, $\vec{f}_i(\vec{z}_i)$. The number of unknowns will never exceed $n \cdot t$ although the number of equations may. Though these additional equations may be used to significant advantage in practice, as was pointed out in section VIII.2, we will ignore them in the following analysis.

Following the same principles used in the analysis of the modular algorithm, we assume that the number of terms in the solution of $\vec{f}_i(\vec{z}_i) = 0$ modulo $\underline{m}_i$ is t. We will also only count integer arithmetic operations now and multiply by the appropriate factor at the end to account for ϵ. $\vec{f}_i(\vec{z}_i)$ will involve no more than nt unknowns $z_i^{(1)}, \ldots, z_i^{(nt)}$. This is important as it gives the size of the Jacobian which needs to be inverted at each stage. As we are only considering classical algorithms, we can assume that inverting the Jacobian will require about $(n \cdot t)^3$ operations. Performing an update of the unknowns involves multiplying a matrix by a vector, this requires about $2(n \cdot t)^2$ operations, but it must be performed d times. We get the following formula for the time required, excluding the computation of the error terms.

$$O\Big(v\big((n \cdot t)^3 + 2d(n \cdot t)^2\big)\Big)$$

which is dominated by $v(nt)^3$.

The error terms must be computed d times for each variable or dv times in all. The computation of an error term is merely a special case of verifying the answer. Without knowing the particular problem under consideration we can't say much more. Denote by T the time required to verify the answer. We can use dvT as an upper bound on the amount of time required to compute the error terms.

Thus the total number of integer operations required is

$$dvT + v(n \cdot t)^3.$$

The total size of the answer t_{tot} is $n \cdot t$, so we can write this as $dvT + vt_{tot}^3$. Including the factor due ϵ computed in section 4.2 we have

$$O\Big(\big(dvT + vt_{tot}^3\big)\Big)\log^2(dv^2 t\epsilon^{-1}).$$

4. Timings

In this section we will present the timings for a two of problems using the Sparse Modular, Brown's and Collins' Modular, the EZGCD algorithms and Wang's new EEZ GCD algorithm.

The first example was chosen to show the sparse modular GCD algorithm at its best. The polynomials for this test are the second set of polynomials listed in the appendix. Again, there are three polynomials listed for each row, f_i, g_i and d_i. This time notice that the structure of each of the polynomials is identical, they only differ by coefficients. The heuristic mentioned in section 2 does not help in this case. The following table gives the computation times, in seconds, for the EZGCD algorithm, the Modular algorithm, the Reduced algorithm, Wang's new EEZ algorithm and finally the Sparse Modular algorithm.

v	EZ	Modular	Reduced	EEZ	Sparse Mod
2	.614	.481	.710	.108	.312
3	2.938	6.092	1.876	2.908	1.074
4	14.935	64.963	*	5.906	1.413
5	*	282.373	*	9.075	2.394
6	*	*	*	60.417	4.153
7	*	*	*	*	5.145
8	*	*	*	*	4.953
9	*	*	*	*	8.699
10	*	*	*	*	8.811

The asterisks indicate that MACSYMA ran out of storage. As expected the modular algorithm ran in exponential time. Both the EZ and the Reduced algorithms ran out of storage. This was to be expected. In the case of the reduced algorithm the size of the PRS finally caught up with it. This example was carefully designed so that all the GCD's were bad zero problems for the EZ algorithm. Thus when the polynomials were shifted the lifting process exploded. Since the heuristic discussed in section 2 was not applicable in this case, the EEZ algorithm also exhibited exponential behavior.

The second example is from [Mos73]. This one was specially designed to be optimal for the EZ GCD algorithm. It is about as bad as possible for the sparse modular algorithm. The two polynomials whose gcd is taken are

$$\left(\sum_{i=1}^{v} x_i^v + 1\right)\left(\sum_{i=1}^{v} x_i^{v-1} + 2\right) \quad \text{and} \quad \left(\sum_{i=1}^{v} x_i^v + 1\right)\left(\sum_{i=1}^{v} x_i^{v-1} - 2\right)$$

Since the EZ algorithm produces all the terms of the answer of the same total degree at once, very little computation is done until the final step, at which time the entire answer is determined. Here we only compare the EZ, the EEZ and the sparse modular algorithms. The others take too long or exceed storage capacity very quickly. Since the sparse modular and EEZ algorithms determine the answer one variable at a time, you would expect at least linear growth as the number of variables increases. Since the degree and number of terms also increases we see significantly more than linear growth for the sparse modular algorithm. It is not at all clear why the EEZ algorithm ran in time so close to that required by the sparse modular algorithm, especially since they are so radically different.

v	EZ	EEZ	Sparse Mod
2	.116	.705	.496
3	.175	1.554	1.375
4	.236	3.368	2.791
5	.341	6.075	5.037
6	.460	9.803	8.299
7	.602	14.903	13.112
8	.760	21.165	19.704
9	.944	28.835	28.709
10	1.142	38.307	40.511

Chapter VIII

Conclusions

The basic idea advanced in this thesis is quite simple. By breaking a problem up into layers, it is possible to make use of the structure of the solution on one layer as a guide to the construction of the solution on the next layer. For the polynomial problems we have considered, and for multivariate polynomial problems in general, this is rather easy because we have a natural layering with which to work. The introduction of a new variable constitutes passing to a higher level. The evaluation homomorphism (replacing X_i by a_i) is a means of going to a lower layer. Throughout this thesis we have used the observation, due to Paul Wang, that the structure of polynomials does not change significantly for sparse polynomials under the evaluation homomorphism.

In the first portion of this thesis the familiar modular algorithm was modified to take advantage of the possible sparseness in the answer. This was accomplished by assuming that if the image of a group of monomials under the evaluation homomorphism is zero then the coefficients of these monomials are exactly zero. This modification has the significant advantage that it turns an algorithm that formerly required exponential time in the number of variables into one that required only probabilistic polynomial time in the size of the answer.

In the second section, we resurrected an old formulation of Hensel's lemma, namely Newton's iteration, and showed how it could be adapted to utilize the sparseness of the intermediate results. This version of Hensel's algorithm also only requires probabilistic polynomial time in the size of the answer. In addition to the increase in efficiency, we feel our formulation of Hensel's lemma is much easier to justify, pedagogically clearer and more widely applicable than other formulations.

There are a number of problems left open by this thesis that are well worth investigating in the future. First, it would be very interesting to see a conventional expected time analysis of the algorithms presented here. Towards that end much tighter bounds than those developed in Theorem 1 of chapter IV must be computed. In particular good bounds for very sparse goal polynomials would be exceedingly interesting.

Second, there are still a vast number of applications of the two basic algorithms presented here that we have not yet investigated. Particularly intriguing are problems of computing resultants and partial fractions expansions. Interfacing the Hensel algorithm to a system for solving systems of algebraic equations to obtain polynomial or rational function solutions would be very useful for a number of problems. This sort of system could also be profitably extended to compute the solution of algebraic equations in a prespecified algebraic field.

Finally, the basic principal of solving problems in layers and making use of the structure of the answer at one level to aid in "lifting" the solution to the next higher level seems to be applicable to many other problems besides those in algebraic manipulation. Algorithms for certain restricted sorts of graphs come to mind, as well as applications in information storage and retrieval.

Appendix I

Polynomials Used in the Timing Tests

This appendix lists the polynomials that were used to test the various GCD algorithms in sections VII.2 and VII.4. The d_i polynomials are the GCDs which are computed, the f_i and g_i the cofactors. The polynomials that were fed to the various GCD routines were $d_i f_i$ and $d_i g_i$.

$$\begin{aligned}
d_1 &= x_1^2 + x_1 + 3\\
f_1 &= 2x_1^2 + 2x_1 + 1\\
g_1 &= x_1^2 + 2x_1 + 2
\end{aligned}$$

$$\begin{aligned}
d_2 &= 2x_1^2x_2^2 + x_1x_2 + 2x_1\\
f_2 &= x_2^2 + 2x_1^2x_2 + x_1^2 + 1\\
g_2 &= x_1^2x_2^2 + x_1^2x_2 + x_1x_2 + x_1^2 + x_1
\end{aligned}$$

$$\begin{aligned}
d_3 &= x_2^2x_3^2 + x_2^2x_3 + 2x_1^2x_2x_3 + x_1x_3\\
f_3 &= x_3^2 + x_2^2x_3 + x_1^2x_2x_3 + x_1x_3 + x_1^2x_2^2\\
g_3 &= x_2x_3 + 2x_1x_3 + x_3 + x_1
\end{aligned}$$

$$\begin{aligned}
d_4 &= x_1^2x_4^2 + x_2^2x_3x_4 + x_1^2x_2x_4 + x_2x_4 + x_1^2x_2x_3\\
f_4 &= x_1x_2x_3^2x_4^2 + x_1x_3^2x_4^2 + x_1x_4^2 + x_4^2 + x_1x_3x_4\\
g_4 &= x_1x_3^2x_4^2 + x_3^2x_4^2 + x_4^2 + x_1x_2^2x_3x_4 + x_1x_2^2
\end{aligned}$$

$$\begin{aligned}
d_5 &= x_1^3x_2^2x_3^2x_4x_5^2 + x_1x_2^2x_5^2 + x_1^3x_3x_4^2x_5 + x_1^3x_2x_3^2x_4x_5 + x_1^2x_2x_3^2x_4^2\\
f_5 &= x_1x_2^2x_5^2 + x_1x_2x_3^2x_4x_5 + x_1x_2x_3^2x_4^2 + x_1x_2^2x_4^2 + 1\\
g_5 &= x_1x_3^2x_4x_5^2 + x_2x_5^2 + x_1x_2x_4x_5 + x_2x_5 + x_1x_2x_3x_4^2
\end{aligned}$$

$$\begin{aligned}
d_6 &= x_1x_2x_4^2x_5^2x_6^2 + x_1x_2^2x_3^2x_4x_5^2x_6^2 + x_1^2x_3x_6^2 + x_1^2x_2x_3^2x_4x_5^2x_6 + x_1^2x_3x_5x_6\\
f_6 &= x_1^2x_2x_4x_5^2x_6^2 + x_1x_3x_5^2x_6^2 + x_1x_2^2x_6^2 + x_1^2x_2^2x_3^2x_5x_6 + x_1x_3^2x_4x_5\\
g_6 &= x_2^2x_3^2x_4x_5^2x_6 + x_1x_4^2x_5x_6 + x_2^2x_3^2x_4x_5x_6 + x_1x_2^2x_3x_4^2x_6 + x_1^2x_3x_5^2
\end{aligned}$$

$$\begin{aligned}
d_7 &= x_1x_2^2x_4^2x_6^2x_7^2 + x_1^2x_3x_4x_6^2x_7^2 + x_3^2x_4^2x_7^2 + x_1^2x_2x_4^2x_6 + x_3x_4x_5^2\\
f_7 &= x_1^2x_2x_4^2x_5x_6^2x_7^2 + x_1x_2x_4x_6x_7 + x_3x_4^2x_5^2x_7 + x_1x_4^2x_5^2x_7 + x_1^2x_2x_3x_4^2x_5x_6\\
g_7 &= x_1x_3x_5x_6^2x_7^2 + x_2^2x_3^2x_4^2x_5x_6x_7^2 + x_4x_6x_7^2 + x_1^2x_2x_3x_5x_6x_7 + x_1^2x_3^2x_4x_5^2
\end{aligned}$$

$$
\begin{aligned}
d_8 &= x_2^2x_4x_5x_6x_7x_8^2 + x_1^2x_2x_3^2x_4^2x_6^2x_7^2x_8 + x_1^2x_3x_4^2x_6^2x_7^2 + x_1^2x_2^2x_3^2x_4x_5^2x_6x_7^2 + x_2^2x_4x_8\\
f_8 &= x_1^2x_2^2x_3x_4^2x_5x_6^2x_8^2 + x_2x_5x_6^2x_8^2 + x_1^2x_2^2x_3^2x_4^2x_6^2x_7^2x_8 + x_1^2x_3^2x_4x_5^2x_7^2x_8 + x_1x_2^2x_3^2x_5^2x_7\\
g_8 &= x_1x_4^2x_5x_6x_7x_8^2 + x_1x_2^2x_4^2x_5^2x_6^2x_8 + x_1^2x_2x_3x_4^2x_6^2x_8 + x_1^2x_2^2x_3^2x_4x_5^2x_8 + x_1x_2x_4^2x_5^2
\end{aligned}
$$

$$
\begin{aligned}
d_9 &= x_1^2x_3^3x_4x_6x_8x_9^2 + x_1x_2x_3x_4^2x_5^2x_8x_9 + x_2x_3x_4x_5^2x_8x_9 + x_1x_3^3x_4^2x_5^2x_6^2x_7x_8^2 + x_2x_3x_4x_5^2x_6x_7x_8^2\\
f_9 &= x_1^2x_2^2x_3x_7^2x_8x_9 + x_2^2x_9 + x_1^2x_3x_4^2x_5^2x_6x_7^2 + x_4^2x_5^2x_7^2 + x_3x_4^2x_6x_7\\
g_9 &= x_1^2x_2x_4x_5x_6x_7^2x_8^2x_9^2 + x_1^2x_2x_3x_5x_6^2x_7^2x_8x_9^2 + x_1^2x_3x_4x_6x_7^2x_8x_9 + x_1^2x_2^2x_6x_8^2 + x_2^2x_4x_5x_6^2x_7
\end{aligned}
$$

$$
\begin{aligned}
d_{10} = {} & \qquad x_1x_2^2x_4^2x_8x_9^2x_{10}^2 + x_2^2x_4x_5^2x_6x_7x_9x_{10}^2 + x_1^2x_2x_3x_5^2x_7^2x_9^2 + x_1x_3^2x_4^2x_7^2x_9^2 + x_1^2x_3x_4x_7^2x_8^2\\
f_{10} = {} & x_1x_2x_3^2x_4x_6x_7x_8x_9^2x_{10}^2 + x_2^2x_3^2x_4^2x_6^2x_9x_{10}^2 + x_1x_2^2x_3^2x_4x_5x_6x_7x_8^2x_9^2x_{10}\\
& + x_1^2x_2x_4^2x_5^2x_8^2x_9^2x_{10} + x_3x_4^2x_5x_6x_7^2x_9x_{10}\\
g_{10} = {} & x_1x_2^2x_3^2x_5^2x_6^2x_7x_8x_9^2x_{10}^2 + x_3x_8x_9^2x_{10}^2 + x_1x_2^2x_3x_4x_5^2x_6^2x_8^2x_9x_{10}\\
& + x_1x_3x_6x_7x_8x_{10} + x_4^2x_5^2x_6^2x_7x_9^2
\end{aligned}
$$

The following set of polynomials was used for the examples of section VI.4. Notice that the structure of each of the polynomials in each triple is the same.

$$f_2 = 18x_1^2x_2^3 + 2x_1^3x_2 + 18x_2 + 10x_1^3 + 18x_1$$
$$g_2 = 16x_1^2x_2^3 + 11x_1^3x_2 + 14x_2 + 4x_1^3 + 12x_1$$
$$d_2 = 15x_1^2x_2^3 + 14x_1^3x_2 + 11x_2 + 19x_1^3 + 7x_1$$

$$f_3 = 16x_1x_2^3x_3^4 + 14x_2^4x_3^3 + 3x_1^3x_3^2 + 18x_1^3x_3 + x_1^2x_2$$
$$g_3 = 19x_1x_2^3x_3^4 + 2x_2^4x_3^3 + 12x_1^3x_3^2 + x_1^3x_3 + 10x_1^2x_2$$
$$d_3 = 7x_1x_2^3x_3^4 + 13x_2^4x_3^3 + 3x_1^3x_3^2 + 4x_1^3x_3 + 14x_1^2x_2$$

$$f_4 = 5x_1^3x_2x_3^3x_4^2 + 3x_2^4x_3x_4^2 + 16x_1^2x_2^2x_4^2 + 19x_2^2x_3^2 + 4x_3 + 19x_2^3$$
$$g_4 = 19x_1^3x_2x_3^3x_4^2 + 9x_2^4x_3x_4^2 + 18x_1^2x_2^2x_4^2 + 17x_2^2x_3^2$$
$$d_4 = 19x_1^5x_2x_3^3x_4^4 + 18x_1^2x_2^4x_3x_4^4 + 18x_1^4x_2^2x_4^4 + 11x_1^2x_3x_4^2 + 3x_1^2x_2^3x_4^2$$

$$f_5 = 17x_2x_3^3x_4x_5^3 + 10x_1^3x_3^3x_4^3x_5^2 + x_3^2x_4x_5 + 2x_2x_4^4 + 10x_1^3x_3^2$$
$$g_5 = 17x_2x_3^3x_4x_5^3 + 3x_1^3x_3^3x_4^3x_5^2 + 18x_3^2x_4x_5 + 10x_1^3x_3^2$$
$$d_5 = 2x_2x_3^2x_4x_5^3 + 3x_1^3x_3^3x_4^3x_5^2 + 12x_3^2x_4x_5 + 10x_2x_4^4 + 18x_1^3x_3^2$$

$$f_6 = 7x_1^2x_2^3x_3^2x_4^4x_5^2x_6^2 + 2x_1^3x_2^4x_3^2x_4^3x_5x_6^2 + 9x_1^3x_2^4x_6 + 19x_3^4x_4^4x_5^3 + 3x_1x_2^2x_4^3x_5^2$$
$$g_6 = 9x_1^2x_2^3x_3^2x_4^4x_5^2x_6^2 + 15x_1^3x_2^4x_3^2x_4^3x_5x_6^2 + 4x_1^3x_2^4x_6 + x_3^4x_4^4x_5^3 + 18x_1x_2^2x_4^3x_5^2$$
$$d_6 = 13x_1^2x_2^3x_3^2x_4^4x_5^2x_6^2 + 12x_1^3x_2^4x_3^2x_4^3x_5x_6^2 + 14x_1^3x_2^4x_6 + 17x_3^4x_4^4x_5^3 + 8x_1x_2^2x_4^3x_5^2$$

$$f_7 = 10x_2^4x_3^4x_4x_6x_7^4 + x_1^3x_2^3x_3^4x_4^2x_7^3 + 14x_1^2x_2x_4^3x_5^4x_6^2x_7^2 + 13x_1^3x_6^3 + 7x_1^3x_2^3x_4^3x_5^3x_6^2$$
$$g_7 = 4x_2^4x_3^4x_4x_6x_7^4 + 17x_1^3x_2^3x_3^4x_4^2x_7^3 + 2x_1^2x_2x_4^3x_5^4x_6^2x_7^2 + 5x_1^3x_6^3 + 13x_1^3x_2^3x_4^3x_5^3x_6^2$$
$$d_7 = 7x_2^4x_3^4x_4x_6x_7^4 + 17x_1^3x_2^3x_3^4x_4^2x_7^3 + 19x_1^2x_2x_4^3x_5^4x_6^2x_7^2 + 2x_1^3x_6^3 + 5x_1^3x_2^3x_4^3x_5^3x_6^2$$

$$f_8 = 14x_1x_2x_4^3x_5x_8^2 + 18x_1^2x_2^4x_3^3x_4^2x_5^3x_6^2x_7^2x_8 + 10x_2^2x_3^2x_5^3x_6^3x_7^2 + 12x_1x_2^2x_3x_4x_5^2x_6 + 7x_1^2x_3^2x_4^3x_8$$
$$g_8 = 11x_1x_2x_4^3x_5x_8^2 + x_1^2x_2^4x_3^3x_4^2x_5^3x_6^2x_7^2x_8 + 12x_2^2x_3^2x_5^3x_6^3x_7^2 + 17x_1x_2^2x_3x_4x_5^2x_6 + x_1^2x_3^2x_4^3x_6$$
$$d_8 = 16x_1x_2x_4^3x_5x_8^2 + 15x_1^2x_2^4x_3^3x_4^2x_5^3x_6^2x_7^2x_8 + 6x_2^2x_3^2x_5^3x_6^3x_7^2 + 14x_1x_2^2x_3x_4x_5^2x_6 + 8x_1^2x_3^2x_4^3x_6$$

$$\begin{aligned} f_9 = {} & 2x_2^3x_3^3x_5^2x_6^2x_8x_9^3 + x_1^3x_2^2x_3^3x_5^2x_7^2x_9^3 + x_1^4x_3^4x_4^2x_5^4x_6^4x_7x_8^3x_9 + 11x_1^4x_2x_3^4x_4^4x_5^4x_7x_8x_9 \\ & + 16x_2^3x_4^3x_6^3x_7^2x_8 \end{aligned}$$
$$\begin{aligned} g_9 = {} & x_2^3x_3^3x_5^2x_6^2x_8x_9^3 + 10x_1^3x_2^2x_3^3x_5^2x_7^2x_9^3 + 13x_1^4x_3^4x_4^2x_5^4x_6^4x_7x_8^3x_9 + 15x_1^4x_2x_3^4x_4^4x_5^4x_7x_8x_9 \\ & + 17x_2^3x_4^3x_6^3x_7^2x_8 \end{aligned}$$
$$\begin{aligned} d_9 = {} & 9x_2^3x_3^3x_5^2x_6^2x_8^3x_9^3 + 9x_1^3x_2^2x_3^3x_5^2x_7^2x_8^2x_9^3 + x_1^4x_3^4x_4^2x_5^4x_6^4x_7x_8^5x_9 + 10x_1^4x_2x_3^4x_4^4x_5^4x_7x_8^3x_9 \\ & + 12x_2^3x_4^3x_6^3x_7^2x_8^3 \end{aligned}$$

$$\begin{aligned} f_{10} = {} & 18x_1^2x_3x_4x_6^3x_7^2x_8x_{10}^4 + 15x_1^3x_2x_5^2x_6^2x_7x_8^3x_9^4x_{10}^3 + x_2^2x_3^4x_4^2x_7^4x_8^3x_9x_{10}^3 + 11x_1^3x_2^2x_6^3x_{10}^2 \\ & + 13x_1x_3x_5^2x_6^2x_7^4x_8^4 \end{aligned}$$
$$g_{10} = 13x_1^2x_3x_4x_6^3x_7^2x_8x_{10}^4 + 17x_2^2x_3^4x_4^2x_7^4x_8^3x_9x_{10}^3 + 6x_1^3x_2^2x_6^3x_{10}^2 + 2x_1x_3x_5^2x_6^2x_7^4x_8^4$$
$$\begin{aligned} d_{10} = {} & 9x_1^2x_3x_4x_6^3x_7^2x_8x_{10}^4 + 17x_1^3x_2x_5^2x_6^2x_7x_8^3x_9^4x_{10}^3 + 17x_2^2x_3^4x_4^2x_7^4x_8^3x_9x_{10}^3 + 3x_1^3x_2^2x_6^3x10^2 \\ & + 10x_1x_3x_5^2x_6^2x_7^4x_8^4 \end{aligned}$$

References

1. C. M. Bender, R. W. Keener and R. E. Zippel, "New Approach to the Calculation of $F_1(\alpha)$ in Massless Quantum Electrodynamics," *Physical Review D* 15, 6 (1977), 1572–1579.
2. E. R. Berlekamp, "Factoring Polynomials over Large Finite Fields," *Math. of Comp.* 24, 111 (1970), 713–735.
3. W. S. Brown, "On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors," *J. ACM* 18, 4 (1971), 478–504.
4. W. S. Brown, "The Subresultant PRS Algorithm," *ACM Trans. on Math. Software* 4, 3 (1978), 237–249.
5. G. E. Collins, "Subresultants and Reduced Polynomial Remainder Sequences," *J. ACM* 14, 1 (1967), 128–142.
6. G. E. Collins, "The Calculation of Multivariate Polynomial Resultants," *J. ACM* 18, 4 (1971), 515–532.
7. M. R. Garey and D. S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness,* W. H. Freeman and Co., San Francisco, (1979).
8. A. O. Gelfond, *Transcendental and Algebraic Numbers,* Dover Pub., Inc., New York, N. Y., (1960).
9. R. L. Graham, "Bounds for certain Multiprocessing Anomalies," *Bell Syst. Tech. J.* 45, (1966), 1563–1581.
10. G. H. Hardy and E. M. Wright, *An Introduction to the Theory of Numbers,* Oxford University Press, London, England, (1968).
11. A. C. Hearn, "Non-Modular Computation of Polynomial GCDs Using Trial Division," *Symbolic & Algebraic Computation (E. W. Ng, Ed.),* Springer-Verlag, Heidelberg, (1979), 227–239.
12. K. Hensel, "Eine neue Theorie der algebraischen Zahlen," *Math. Zeitschr.* 2, (1918), 433–452.
13. R. M. Karp, "The Probabilistic Analysis of some Combinatorial Search Algorithms," *Algorithm and Complexity—New Directions and Recent Results, (J. F. Traub, Ed.),* Acad. Press, New York, (1976), 1–19.
14. D. E. Knuth, *The Art of Computer Programming,* Vol. II, Addison-Wesley Publishing Company, Reading, Mass., (1969).
15. S. Lang, *Diophantine Geometry,* Interscience Publishers, New York, N. Y., (1965).
16. M. Lauer, *Generalized p-adic Constructions,* Ph. D.thesis, Univ. Karlsruhe, (1978).

17. D. J. Lewis, "Diophantine Equations: p-adic Methods," *Studies in Number Theory, (W. S. Leveque, Ed.),* Math. Assoc. of Amer., (1969).
18. MATHLAB Group, *MACSYMA Reference Manual—version 9,* Laboratory for Computer Science, Massachusetts Institute of Technology, (1977).
19. R. J. McEliece and J. B. Shearer, "A Property of Euclid's Algorithm and an Application to Padê Approximation," *SIAM J. Appl. Math.* 34, 4 (1978), 611-615.
20. J. Moses and D. Y. Y. Yun, "The EZGCD algorithm," *Proceedings of ACM Nat. Conf.* (1973), 159-166.
21. D. R. Musser, "Multivariate Polynomial Factoring," *J. ACM* 22, 2 (1975), 291-308.
22. C. H. Papadimitriou, "On the Symbolic Evaluation of Determinants," (in preparation).
23. M. O. Rabin, "Probabilistic Algorithms," *Algorithm and Complexity--New Directions and Recent Results, (J. F. Traub, Ed.),* Acad. Press, New York, (1976), 1-19.
24. J. T. Schwartz, "Probabilistic Algorithms for Verification of Polynomial Identities," *Symbolic & Algebraic Computation (E. W. Ng, Ed.),* Springer-Verlag, Heidelberg, (1979), 200-215.
25. A. Shamir and R. E. Zippel, "On the Security of the Merkle-Hellman Cryptographic Scheme," *IEEE Trans. on Information* , to appear.
26. R. Solovay and V. Strassen, "A fast Monte Carlo Test for Primality," *SIAM J. of Comp.* 6, 1 (1977), 84-85.
27. B. M. Trager and P. S.-H. Wang, "On Square-free Decomposition," *SIAM Rev. of Comp.* , to appear.
28. P. S.-H. Wang and T. Minamikawa, "Taking Advantage of Zero Entries in the Exact Inverse of Sparse Matrices," *Proc. of SYMSAC'76, ACM* (1976), 346-350.
29. P. S.-H. Wang and L. P. Rothschild, "Factoring Multivariate Polynomials over the Integers," *Math. Comp.* 29, (1975), 935-950.
30. P. S.-H. Wang, "An Improved Multivariate Polynomial Factoring Algorithm," *Math. Comp.* 32, (1978), 1215-1231.
31. P. S.-H. Wang, "An Improved Multivariate GCD Algorithm," , in preparation.
32. E. Waring, "Problems Concerning Interpolations," *Phil. Trans. of the Royal Society of London* 69, (1779), 59-67.
33. B. W. Weide, *Statistical Methods in Algorithm Design and Analysis,* Ph. D. thesis, Dept. of Computer Science, Carnegie-Mellon University, (1978).
34. D. Y. Y. Yun, *The Hensel Lemma in Algebraic Manipulation,* Ph. D. thesis, Dept. of Mathematics, Massachusetts Institute of Technology, (1974).
35. D. Y. Y. Yun, "Algebraic Algorithms using p-adic Constructions," *Proc. of SYMSAC'76, ACM* (1976), 248-259.
36. H. Zassenhaus, "On Hensel Factorization I," *J. Number Theory* 1, (1969), 291-311.

DATE

ILME