Symbolic Computation with Algebraic Numbers and Functions

Trevor Janathan Smedley

Research Report CS-87-29
May 1987

Symbolic Computation with Algebraic Numbers and Functions

by

Trevor Janathan Smedley

A thesis presented to the University of Waterloo
in fulfillment of the requirements for the degree of
Master of Mathematics in Computer Science

Waterloo, Ontario, Canada.
January, 1987

Contents

1. Introduction

As symbolic computation systems have evolved, one of the major factors in their evolution has been the expansion of the domain of computation. Exact computation with integers and rationals has been available since the advent of symbolic computation, as well as computation with polynomials and rational functions over these domains. Recently there has been a lot of work on expanding the domain of computation of various symbolic computation systems to include algebraic numbers. Algebraic numbers arise in the solutions of equations and in integration problems, so it is quite natural to want to be able to handle them in a consistent manner.

This thesis describes the principles behind, and the algorithms used in an implementation of an algebraic number and algebraic function package in the Maple algebra system (3).

The algorithms are presented in a pseudo-language similar to Maple. The extension from algebraic numbers to algebraic functions is seen to be quite easy and increases the usefulness considerably.

Algebraic Numbers and Functions

An algebraic number is a root of a univariate polynomial over the rationals (or integers). Examples are $2^{1/2}$, $(3^{1/3} + 5^{3/7})^{2/3}$, and α where $\alpha^5 + \alpha^3 + \alpha^2 + 5 = 0$. Note that, in the last example, α cannot be expressed in terms of radicals, and that the symbol, α, can be any of the five roots of the polynomial, just as $2^{1/2}$ can be 1.4142... or −1.4142....

An algebraic function is a root of a multivariate polynomial over the rationals (or integers). Examples are $x^{1/2}$, $(2x^{1/3} + y^{3/5})^{5/7}$, or α where $(y + x)\alpha^5 + y\alpha^3 + \alpha^2 + y^2 + x = 0$. Note that, as in the example for algebraic numbers, α cannot be expressed in terms of radicals and stands for any one of the five roots of the equation.

Throughout this thesis, with the exception of Chapter Seven, no specific value is associated with any algebraic number or function.

Previous Work

Before the advent of computer algebra systems, any work done concerning computation in algebraic fields was purely theoretical. Because of the complexity of even simple operations, it was essentially impossible to apply any of the theory to solving concrete problems. In spite of this, algorithms were developed for various operations involving algebraic numbers: in particular, algebraic polynomial factorisation. Van der Waerden (15) and Kronecker (7) are important references for this.

Since computer algebra systems have come into use, much emphasis has been placed on developing efficient and easily implementable algorithms for computation with algebraic numbers. Loos (11) gives one of the first comprehensive presentations of algorithms for computation in algebraic number fields. He assumes, however, that the algebraic number fields are represented by simple algebraic extensions of the rationals, resorting to the use of primitive elements in order to do calculations in multiple extensions. In Chapter Two a

serious problem with this method is pointed out.

Abbott et al. (1) give a description of an algebraic number package which has been implemented in REDUCE. The implementation is based on the use of multiple algebraic extensions instead of primitive elements, and their tests indicate that this is, in general, an efficient method. It seems that they have a complete, functional implementation, although there is little mention of the possibility of extending their system to algebraic functions, as is described in this thesis. Also, they only give a description of the system, without any of the details of the implementation. In particular, no algorithms are given.

Kronecker (7) is probably the first reference on factorisation of polynomials over an algebraic extension field. Van der Waerden (15) also discusses this, and Trager (14) gives a complete algorithmic presentation. The algorithm presented by Trager has been implemented in a number of computer algebra systems, and has proven to be useful, although it is often quite slow due to the difficult integer polynomial factorisations which have to be done. Landau (8) presents a complexity analysis of Trager's algorithm, with the conclusion that it is a polynomial time algorithm. However, this analysis is based on the assumption that algebraic polynomial greatest common divisors can be computed in polynomial time, an assumption which is not substantiated anywhere in the literature. This assumption is based on the fact that integer polynomial GCDs can be calculated in polynomial time. However, since computing algebraic polynomial GCDs involves the computation of algebraic number inverses, an operation which is very expensive and normally results in large expression growth, it is not clear that this assumption is true. Certainly more work is needed on the analysis of algorithms using algebraic numbers. Smedley (13) presents coefficient growth bounds for the standard operations on univariate polynomials over a simple algebraic extension field, but he draws no conclusions about the tightness of his bound for the norm of the result of an algebraic polynomial GCD operation.

The problem of efficient factorisation of polynomials over an algebraic number field has attracted much attention. In particular, besides the algorithm presented by Trager (14), Wang (16) and Weinberger & Rothschild (17) present modular algorithms. This approach has difficulty, as mentioned in Abbott et al. (1), when the minimal polynomial factors modulo every prime — as in the case of $x^4 - 1$. Wang mentions that his algorithm resorts to Trager's algorithm in these cases.

Lenstra (9, 10) gives two versions of an algorithm based on short vectors in lattices. A comparison of Lenstra's and Trager's algorithms in Abbott et al. (1) is inconclusive as to which would perform better in general. It may be that a hybrid algorithm would be best for this problem; but in any case much more investigation is required.

Overview

Chapter Two includes most of the mathematical background needed for an understanding of the thesis. References are given where a more detailed discussion of some of the more complicated concepts can be found.

Chapter Three gives definitions for canonical forms for algebraic numbers and functions, and an algorithm to put an algebraic number or function in canonical form.

In Chapter Four the algorithms for the basic operations on elements of an algebraic field are presented.

The algorithms for the basic polynomial operations — quotient, remainder and greatest common denominator — are given in Chapter Five, and polynomial factorisation is discussed in Chapter Six. Both of these chapters discuss the case of univariate polynomials in detail, and indicate how this can be extended to the multivariate case.

Chapter Seven presents and application of computation in algebraic fields. An algorithm for simplification of expressions involving radicals is given which makes extensive use of the algorithms for computation in algebraic fields.

Chapter Eight has some concluding remarks and suggestions for future work.

2. Mathematical Background

This chapter gives most of the mathematical background necessary for an understanding of the thesis, with references given for more detail on certain subjects. The notation and definitions used throughout the thesis are presented here. A basic knowledge of algebra is assumed, including rings and fields.

Basic Definitions

The following definitions can be found in Hungerford (5), or van der Waerden (15).

Polynomial Rings

The ring of polynomials over the field, K, in the variables $x_1, \cdots, x_n$, is denoted by $K[x_1, \cdots, x_n]$.

Rational Function Fields

The field of rational functions over the field, K, in the variables $x_1, \cdots, x_n$, is denoted by $K(x_1, \cdots, x_n)$.

Extension Fields

If F and G are fields, and $F \subset G$ as a subfield, then G is called an extension field of F.

Algebraic

If G is an extension field of F, and $\alpha \in G$, then α is algebraic over F if there exists a polynomial $f(x) \in F[x]$ such that $f(\alpha) = 0$. Note that elements of F are also algebraic over F.

Minimal Polynomial

If α is algebraic over a field, F, then $f(x) \in F[x]$ is the minimal polynomial of α over F if $f(\alpha) = 0$; f is monic; and if, $\forall\ g(x) \in F[x]$ such that $g(\alpha) = 0$, f divides g. Under this definition the minimal polynomial is unique. See Hungerford (5) for a proof.

Note that in this definition the requirement that a minimal polynomial be monic is only there to ensure uniqueness. All of the theory is also valid with non-monic minimal polynomials. We will see, however, that it is also beneficial from an efficiency standpoint to have minimal polynomials monic.

Simple Algebraic Extension Field

If α is algebraic over a field, F, then $F(\alpha)$ is a simple algebraic extension of F. It is the smallest field containing both F and α.

Multiple Algebraic Extension Field

If $\alpha_1, \cdots, \alpha_n$ are algebraic over F, then $F(\alpha_1, \cdots, \alpha_n)$ is a multiple algebraic extension of F. It is the smallest field containing F and $\alpha_1, \cdots, \alpha_n$.

Algebraic Number

If α is algebraic over the rationals then it is an algebraic number. Note that rationals are also algebraic numbers.

Algebraic Function

If α is algebraic over $Q(x_1, \cdots, x_m)$, for some variables $x_1, \cdots, x_m$, then α is an algebraic function in the variables $x_1, \cdots, x_m$. Note that algebraic numbers and rationals are also algebraic functions.

Algebraic Number Field

A field extension of the rationals which contains only algebraic numbers is called an algebraic number field.

Algebraic Function Field

A field extension of $Q(x_1, \cdots, x_m)$ which contains only algebraic functions in the variables $x_1, \cdots, x_m$ is called an algebraic function field in the variables $x_1, \cdots, x_m$.

Algebraic Field

The term *Algebraic Field* is used to refer to either an algebraic number field or an algebraic function field.

Multiple Extensions vs. Primitive Elements

When it is desired to compute in a multiple algebraic extension field, $Q(\alpha_1, \cdots, \alpha_n)$, one has the choice of doing the computations in the multiple extension directly (as presented in this thesis and in Abbott et al. (1)) or computing a primitive element, $\gamma \in Q(\alpha_1, \cdots, \alpha_n)$, such that $Q(\gamma) = Q(\alpha_1, \cdots, \alpha_n)$ and then doing the computations in the simple algebraic extension $Q(\gamma)$. This is the approach advocated in Loos (11), and there has been considerable discussion as to which method is better. We present here an argument for the use of multiple extensions.

Suppose we want to compute in $Q(\sqrt{2}, \sqrt{3}, \sqrt{5}, \sqrt{7})$. Under the multiple extension approach, the field would be defined by $[x_1^2 - 2, x_2^2 - 3, x_3^2 - 5, x_4^2 - 7]$, and $\sqrt{2}$ would be represented by x_1, $\sqrt{3}$ by x_2, $\sqrt{5}$ by x_3, and $\sqrt{7}$ by x_4. Under the primitive element approach, a primitive element would be found such that $Q(\gamma) = Q(\sqrt{2}, \sqrt{3}, \sqrt{5}, \sqrt{7})$. One such element is:

$$\gamma \quad \textit{where} \quad 46225 - 5596840\gamma^2 + 13950764\gamma^4 - 7453176\gamma^6 + 1513334\gamma^8$$

$$-141912\gamma^{10} + 6476\gamma^{12} - 136\gamma^{14} + \gamma^{16} = 0$$

In this representation:

$$\sqrt{2} = \frac{1000302037\gamma}{63406080} - \frac{4763001509\gamma^3}{105676800} + \frac{1547095997\gamma^5}{63406080} - \frac{1572360191\gamma^7}{317030400}$$

$$+\frac{5894795\gamma^9}{12681216}-\frac{6720901\gamma^{11}}{317030400}+\frac{627\gamma^{13}}{1409024}-\frac{1037\gamma^{15}}{317030400}$$

$$\sqrt{3}=-\frac{48197780729\gamma}{2493972480}\frac{231448592713\gamma^3}{4156620800}-\frac{385742234501\gamma^5}{12469862400}+\frac{79643171179\gamma^7}{12469862400}$$

$$-\frac{7531157647\gamma^9}{12469862400}+\frac{345235241\gamma^{11}}{12469862400}-\frac{2422917\gamma^{13}}{4156620800}+\frac{53521\gamma^{15}}{12469862400}$$

$$\sqrt{5}=-\frac{37112819933\gamma}{1496383488}\frac{135356339947\gamma^3}{2493972480}-\frac{41565980041\gamma^5}{1496383488}+\frac{41545500733\gamma^7}{7481917440}$$

$$-\frac{774253511\gamma^9}{1496383488}+\frac{176168243\gamma^{11}}{7481917440}-\frac{82107\gamma^{13}}{166264832}+\frac{27151\gamma^{15}}{7481917440}$$

and

$$\sqrt{7}=\frac{12200206607\gamma}{415662080}-\frac{67424441651\gamma^3}{1039155200}+\frac{71310531461\gamma^5}{2078310400}-\frac{3626646037\gamma^7}{519577600}$$

$$+\frac{1364453637\gamma^9}{2078310400}-\frac{6241559\gamma^{11}}{207831040}+\frac{1312971\gamma^{13}}{2078310400}-\frac{151\gamma^{15}}{32473600}$$

Under the multiple extension method, if we have expressions involving the square roots and perform operations with them within the field, then the structure of the results in terms of $\sqrt{2}$, $\sqrt{3}$, $\sqrt{5}$, and $\sqrt{7}$ is immediately obvious. However, when using a primitive element, it is impossible, for example, to tell that an expression such as:

$$-\frac{46886083}{4349952}-\frac{127900421\gamma^2}{28999680}+\frac{2066144153\gamma^4}{217497600}-\frac{1127914157\gamma^6}{434995200}$$

$$+\frac{116903\gamma^8}{424800}-\frac{5815669\gamma^{10}}{434995200}+\frac{21071\gamma^{12}}{72499200}-\frac{947\gamma^{14}}{434995200}$$

is, in fact, $2\sqrt{2}\sqrt{3}+\sqrt{5}\sqrt{7}$. Under the multiple extension approach, this would be represented by $2x_1x_2+x_3x_4$.

Thus we see that, when using primitive elements, much useful information is lost, and, in general, the results of computations are essentially useless. Because of this the primitive element approach was abandoned in favour of the use of multiple extensions. It is still uncertain which method is more efficient, but the multiple extension method is certainly more useful, and it is doubtful that there is an appreciable difference in the efficiency of the two methods. We have have implemented both methods, and no large differences in efficiency have been noticed.

Representation of Algebraic Fields

Polynomial Quotient Field

If K is a field and $f \in K[x]$, then $K[x]/(f)$ denotes the ring of polynomials, $K[x]$, modulo the polynomial f. If f is irreducible over K then $K[x]/(f)$ is a field.

Isomorphism Theorem

If K is a field; α is algebraic over K; and $f(x) \in K[x]$ is the minimal polynomial for α over K, then:

$$K(\alpha) \approx K[x]/(f)$$

and each element of $K(\alpha)$ can be uniquely represented in the form $a_0 + a_1x + \cdots + a_{n-1}x^{n-1}$ where $n = degree(f, x)$ and $a_i \in K$, $i=0, \ldots, n-1$.

For a proof of the above and a more detailed discussion of algebraic field theory see Hungerford (5).

Extended Isomorphism Theorem

The previous isomorphism theorem tells us that the elements of a simple algebraic extension field can be represented by univariate polynomials over the field modulo the minimal polynomial. However, we would like to do computations in multiple extensions, so we must extend the above theorem.

Suppose K is a field; α is algebraic over K; $G = K(\alpha)$; and β is algebraic over G. Then $G(\beta) = K(\alpha, \beta)$. Also suppose that $f_\alpha(x) \in K[x]$ is the minimal polynomial for α over K, and $f_\beta(y) \in G[y]$ is the minimal polynomial for β over G. Then, from the previous isomorphism theorem, we know that $G \approx K[x] / (f_\alpha)$ and $G(\beta) \approx G[y] / (f_\beta)$. Thus we have:

$$K(\alpha, \beta) = G(\beta) \approx K(\alpha)[y] / (f_\beta)$$

or equivalently:

$$K(\alpha, \beta) \approx (K[x] / (f_\alpha))[y] / (f_\beta)$$

We write this:

$$K[x, y] / [f_\alpha, f_\beta]$$

In general, if K is a field; $\alpha_1, \cdots, \alpha_n$ are algebraic over K; and $f_i(x_i) \in K(\alpha_1, \cdots, \alpha_{i-1})[x_i]$ is the minimal polynomial for α_i over $K(\alpha_1, \cdots, \alpha_{i-1})$, then:

$$K(\alpha_1, \cdots, \alpha_n) \approx K[x_1, \cdots, x_n] / [f_1, \cdots, f_n]$$

Note that the right hand side is just a shorthand notation and can be expanded as in the above example for a double extension.

We also know that each element of $K(\alpha_1, \cdots, \alpha_n)$ can be uniquely represented by $p(x_1, \cdots, x_n) \in K[x_1, \cdots, x_n]$ where $degree(p, x_i) < degree(f_i, x_i)$, $i=1, \cdots, n$.

Representation

From the preceding sections we see that the elements of a simple algebraic extension of a field can be represented by univariate polynomials over the field, and that extending this to multiple field extensions is a simple matter of applying the above concept recursively. Thus we can represent the elements of a multiple algebraic extension by multivariate polynomials over the base field.

In order to do computations in an algebraic field, we need information about the structure of the field. This information is carried in the minimal polynomials. Thus a simple algebraic extension field is given by the minimal polynomial, and a multiple extension is defined by a list of minimal polynomials, ordered so that f_i is the minimal polynomial for α_i over $K(\alpha_1, \cdots, \alpha_{i-1})$.

All of the algorithms presented in this thesis use the same representation for an algebraic extension field. There is a single parameter — usually called *Extension* — which defines the extension. It is a list of three items: the first is the number of algebraics in the extension; the second is the names of the variables used; and the third gives the minimal polynomials. Thus if the following assignments are made:

```
n := Extension[1];
alg := Extension[2];
f := Extension[3];
```

Then $f_i(alg_i)$ for i=1, $\cdots$,n are the minimal polynomials for the extension.

Example 1

$$Q(3^{1/3}, 5^{1/7}, (3^{1/3} + 5^{3/7})^{1/3})$$

is defined by

$$[x_1^3 - 3, x_2^7 - 5, x_3^3 - x_1 - x_2^3]$$

and

$$3^{2/3} - 5^{5/7} + (3^{1/3} + 5^{3/7})^{2/3}$$

is represented by

$$x_1^2 - x_2^5 + x_3^2$$

Example 2

$$Q(y^{1/3}, z^{1/5}, (2y^{1/3} + z^{3/5})^{1/7})$$

is defined by

$$[x_1^3 - y, x_2^5 - z, x_3^7 - 2x_1 - x_2^3]$$

and

$$y^{1/3} + z^{2/5} + (2y^{1/3} + z^{3/5})^{3/7}$$

is represented by

$$x_1 + x_2^2 + x_3^3$$

Efficiency Considerations

If α is algebraic over a field K, its minimal polynomial over K is unique; however, the actual field $K(\alpha)$ does not have a unique representation. For example, $Q(\sqrt{2}) = Q(5\sqrt{2})$. We can take advantage of this fact to try to improve the efficiency of algebraic computations by choosing a representation for the field (that is, a list of minimal polynomials) which leads to efficient computation.

A minimal polynomial for an algebraic number is a polynomial over the rationals. Since computation with integers is typically faster than with rationals, it would be desirable to have a minimal polynomial with integer coefficients. Thus, if we want to compute with α where the minimal polynomial is $f(x)$, which is monic in x and has non-integer coefficients, we do the following:

i) Choose $r \in Z$ such that $r \cdot f(x)$ has only integer coefficients.

ii) Let $g(x) = r^n f(x/r)$ where $n = degree(f, x)$.

Then $g(r\alpha) = 0$, g has only integer coefficients, and g is monic in x. Thus we use g as the minimal polynomial and do the computation in $Q(r\alpha)$ instead of $Q(\alpha)$.

Note that we could have used $r \cdot f(x)$ as the minimal polynomial even though it is not monic.† However, since the main computation done with a

† Recall that the requirement in the definition that the minimal polynomial be monic is only there to ensure that it is unique.

minimal polynomial is remaindering modulo this polynomial, it is extremely advantageous to ensure that minimal polynomials are monic.

If we are dealing with algebraic functions, the minimal polynomials are multivariate polynomials with multivariate rational function coefficients. As before it is desirable to have minimal polynomials which are monic integer polynomials; in this case we can perform a transformation similar to those done with algebraic numbers to improve the efficiency of the computations.

Synopsis

Algebraic numbers are represented by multivariate polynomials over the rationals. Algebraic functions are represented by multivariate polynomials whose coefficients are rational functions in the variables of the algebraic functions. An algebraic field is given by a list of multivariate polynomials over the integers, ordered so that f_i is the minimal polynomial for α_i over $K(\alpha_1, \cdots, \alpha_{i-1})$, where K is the base field — either the rationals, or a rational function field over the rationals.

Norms

Definitions

Let α be algebraic over a field K; $f(x) \in K[x]$ be the minimal polynomial for α over K; and $G \supset K(\alpha)$ be a field containing all the roots of f. Let $n = degree(f, x)$; $\alpha=\alpha_1, \cdots, \alpha_n \in G$ be all of the roots of f; and $\beta \in K(\alpha)$, which is represented by a polynomial $p(x) \in K[x]$. Then define

$$Norm_{K(\alpha)/K}(\beta) = \prod_{i=1}^{n} p(\alpha_i)$$

If $g(y_1, \cdots, y_k) \in K(\alpha)[y_1, \cdots, y_k]$ is represented by $g'(x, y_1, \cdots, y_k) \in K[x][y_1, \cdots, y_k]$, then also define,

$$Norm_{K(\alpha)/K}(g) = \prod_{i=1}^{n} g'(\alpha_i, y_1, \cdots, y_k)$$

If it is clear which extension field is implied, the subscript $K(\alpha)/K$ may be omitted from the notation.

Important Properties

$$Norm(A \cdot B) = Norm(A)\ Norm(B)$$

$$\beta \in K(\alpha) \Rightarrow Norm_{K(\alpha)/K}(\beta) \in K$$

$$g \in K(\alpha)[y_1, \cdots, y_k] \Rightarrow Norm_{K(\alpha)/K}(g) \in K[y_1, \cdots, y_k]$$

Under the definitions above,

$$Norm_{K(\alpha)/K}(\beta) = resultant_x(f(x), p(x))$$

$$Norm_{K(\alpha)/K}(g) = resultant_x(f(x), g'(x, y_1, \cdots, y_k))$$

where $resultant_x(f_1(x), f_2(x))$ is the resultant of f_1 and f_2 taken with respect to x.

Some Theorems

The following theorems are from Trager (14). They are also discussed in detail in Kronecker (7) and in van der Waerden (15).

Let α be algebraic over a field K, with minimal polynomial $f(x) \in K[x]$. Let $p(y) \in K(\alpha)[y]$ be represented by the polynomial $p'(x, y) \in K[x, y]$. Let $P(y) = Norm_{K(\alpha)/K}(p(y)) \in K[y]$.

1) If $p(y)$ is square-free, there exist only a finite number of $s \in K$ such that $Norm_{K(\alpha)/K}(p(y - s\alpha))$ is not square-free, and a bound on the number of $s \in K$ such that the norm is not square-free can be calculated.

2) If $P(y)$ is square-free, and $\prod_{i=1}^{k} P_i(y)$ is a complete factorisation of P over K, then:

$$\prod_{i=1}^{k} gcd(p(y), P_i(y))$$

is a complete factorisation of $p(y)$ over $K(\alpha)$, where the GCD is done in $K(\alpha)[y]$.

Use in Algebraic Polynomial Factorisation

Using the two theorems from the previous subsection, it is possible to develop an algorithm for factorisation of square-free polynomials over a simple algebraic extension of the rationals: first a transformation is found which makes the norm of the polynomial to be factored square-free; then this norm is factored over the base field (the rationals); the factors are lifted back to the algebraic extension field using algebraic polynomial greatest common divisors; and finally the transformation is undone to get the factors of the original polynomial.

This algorithm, along with an extension to deal with multiple algebraic extensions and algebraic functions, are discussed in detail in Chapter Six.

3. Canonical Forms

Before we can even discuss computation with elements of an algebraic field, we should have some concept of a canonical form for elements of the field. This allows us to recognise zero, elements of the base field, and makes it easy to determine equivalence of two field elements.

Algebraic Numbers

If we are dealing only with algebraic numbers, and not with algebraic functions, then the definition of canonical form is as follows:

If $\alpha_1, \cdots, \alpha_n$ are algebraic numbers; $\gamma \in Q(\alpha_1, \cdots, \alpha_n)$, which is represented in $Q[y_1, \cdots, y_n]$; and $f_i(y_1, \cdots, y_i)$ is the minimal polynomial for α_i over $Q(\alpha_1, \cdots, \alpha_{i-1})$; then γ is in canonical form if $degree(\gamma, y_i) < degree(f_i, y_i)$, $i=1, \cdots, n$, and the coefficients of the y_i in γ, rational numbers, are in lowest terms.

In Chapter Two we saw that every element of an algebraic number field has as a unique representation as above, so this is a canonical form.

Algebraic Functions

The definition for the canonical form in an algebraic function field is essentially the same as for algebraic numbers, except that the ground field is $Q(x_1, \cdots, x_m)$ instead of Q, where the x_i are the variables in the algebraic functions.

If $\alpha_1, \cdots, \alpha_n$ are algebraic functions in the variables $x_1, \cdots, x_m$; $K = Q(x_1, \cdots, x_m)$; $\gamma \in K(\alpha_1, \cdots, \alpha_n)$, which is represented in $K[y_1, \cdots, y_n]$; and $f_i(y_1, \cdots, y_i)$ is the minimal polynomial for α_i over $K(\alpha_1, \cdots, \alpha_{i-1})$; then γ is in canonical form if $degree(\gamma, y_i) < degree(f_i, y_i)$, $i=1, \cdots, n$, and the coefficients of the y_i in γ (elements of K) are in canonical form.

As in the case of algebraic numbers, we saw in Chapter Two that this is a canonical form.

Canonical Form Algorithm

Given an arbitrary rational function $\gamma \in Q(x_1, \cdots, x_m)(\alpha_1, \cdots, \alpha_n)$, and the minimal polynomials defining an algebraic field, $[f_1, \cdots, f_n]$, we want to be able to put γ in canonical form in the algebraic field. The following is an algorithm to do this:

```
alg_canonical(g, Extension)
   g := normal(g);
   numer := numerator(g);
   denom := denominator(g);
   numer := reduce_degree(numer, Extension);
   denom := reduce_degree(denom, Extension);
   denom_inv := alg_inverse(denom, Extension);
   result := reduce_degree(numer*denom_inv, Extension);
return(result);
```

First the expression is put in normal form as a rational function in the x_i and α_i. Then the algorithm simply splits the input into its numerator and denominator, which are polynomials; puts them in canonical form in the extension with the algorithm *reduce_degree*; computes the inverse of the denominator; and then returns the product of the numerator and the inverse of the denominator in canonical form. An algorithm for computing the inverse is presented in Chapter Four.

The following algorithm, *reduce_degree*, puts an expression which is a polynomial in the algebraic elements in canonical form. In this case all that is required is reduction of the expression modulo the minimal polynomials, and putting the coefficients in canonical form in the base field. This algorithm is used instead of *alg_canonical* whenever it is known that the expression to be put in canonical form is a polynomial in the variables representing the algebraics. That is, when it is known that the denominator of the expression contains no algebraics.

```
reduce_degree(p, Extension)
   n := Extension[1];
   alg := Extension[2];
   f := Extension[3];
   result := p;
   for i from n by -1 to 1 do
      result := remainder(result, f_i, alg_i)
   enddo;
   result := canonical_coeffs(result, alg);
return(result);
```

The function *canonical_coeffs* puts the coefficients of *result*, taken as a polynomial in the alg_i, in canonical form.

The algorithm iteratively reduces the degree of the expression with respect to each of the minimal polynomials by taking remainders. Although the expressions are multivariate polynomials, the remainders are true remainders, and not just pseudo-remainders, because the minimal polynomials are monic with respect to the variable of division. Also, since f_i does not involve alg_j for any $j > i$, division by f_i does not increase the degree of the expression in alg_j for any $j > i$. Thus at the end of the algorithm we have the following two properties:

$$degree(result, alg_i) < degree(f_i, alg_i),\ i=1, \cdots, n$$

$$p = result + \sum_{i=1}^{n} q_i \cdot f_i \qquad \text{as multivariate polynomials}$$

for some q_i. We also know that the coefficients of result with respect to the alg_i are in canonical form. From this we see that *result* is in canonical form, and $result = p$ in the algebraic field, since $f_i = 0$, $i=1, \cdots, n$ in the field.

Efficiency Considerations

In Chapter Four it is noted that the calculation of inverses is very slow, and often leads to large expression swell. Because of this it is often desirable to use an algorithm which does not give a canonical form, but does guarantee the recognition of zero. The following algorithm, *alg_normal*, basically puts the numerator and denominator in canonical form, but does not take the inverse of the denominator. This results in a normal form, which guarantees zero recognition.

```
alg_normal(g, Extension)
   g := normal(g);
   numer := numerator(g);
   denom := denominator(g);
   numer := reduce_degree(numer, Extension);
   denom := reduce_degree(denom, Extension);
   result := normal(numer/denom);
return(result);
```

The procedure *normal* in the above algorithm is a multivariate rational function normalisation routine. It removes the greatest common divisor of the numerator and denominator (as multivariate polynomials).

Use in Other Algorithms

Throughout this thesis, whenever the algorithm *alg_canonical* is mentioned, it is implied that the algorithm of those mentioned in this section which is most suited to the desired purpose is used. Also, whenever the algorithm is applied to a polynomial over an algebraic field, it is implied that it is to be applied to each of the coefficients individually.

4. Basic Operations

The basic arithmetic operations needed for computations in an algebraic field are addition, subtraction, multiplication and division. Using the canonical form algorithm of the previous chapter, we can assure that the operands are in canonical form before the computation is done, so we only need worry about doing the operations on algebraics which are in canonical form.

Addition and Subtraction

Addition and subtraction are done by performing the operation with the multivariate polynomial representations and putting the coefficients in the result in canonical form. In the case of algebraic numbers, the coefficients are just rational numbers, and they are put in lowest terms. However, in an algebraic function field the coefficients are rational functions, and putting them in canonical from involves taking polynomial greatest common divisors. Since adding or subtracting polynomials over a field cannot increase the degree of the polynomials there is no need to reduce the degree.

Multiplication

Multiplication of two algebraics is essentially the same as addition and subtraction. The operation is performed on the multivariate polynomial representations and then the result is put in canonical form. Since multiplication can increase the degree, it may be necessary to reduce the result modulo the minimal polynomials. Thus multiplication is achieved by multiplying the polynomial representations and then using the algorithm, *reduce_degree*, of Chapter Three to put the result in canonical form.

Inverses

Division of two algebraic field elements is achieved by computing the inverse of the divisor, and then multiplying. Thus it is necessary that we be able to compute inverses. The algorithm implemented in Maple is as follows:

```
alg_inverse(a, Extension)
   n := Extension[1];
   alg := Extension[2];
   f := Extension[3];
   inverse := 1;
   denom := a;
   for i from n by -1 to 1 do
      s := gcdex(denom, f_i, alg_i)[2];
      numer := numerator(s);
      denom := reduce_degree(denominator(s), Extension);
      inverse := reduce_degree(inverse*numer, Extension);
   enddo;
   inverse := canonical_coeffs(inverse/denom, alg);
return(inverse);
```

The inverse algorithm can also be implemented recursively, and this presentation makes the method clearer.

```
recursive_alg_inverse(a, Extension)
   n := Extension[1];
   if n=0 then
      inverse := 1/a;
   else
      alg := Extension[2];
      f := Extension[3];
      new_Extension[1] := n-1;
      new_Extension[2] := alg_i, i=1..n-1;
      new_Extension[3] := f_i, i=1..n-1;
      s := gcdex(a, f_n, alg_n)[2];
      numer := numerator(s);
      denom := reduce_degree(denominator(s), new_Extension);
      denom_inverse := recursive_alg_inverse(denom, new_Extension);
      inverse := reduce_degree(numer*denom_inverse, Extension);
   endif;
return(inverse);
```

The function $gcdex(u, v, x)$ returns a list of length three, where the first element is the greatest common divisor of u and v, and the other two are the s and t satisfying $s \cdot u + t \cdot v = gcd(u, v)$, such that $degree(s, x) < degree(v, x)$ and $degree(t, x) < degree(u, x)$. Thus the s in the algorithm is a polynomial in alg_n, where the coefficients are rational functions in alg_j, j=1, $\cdots$,n−1, such that $degree(s, alg_n) < degree(f_n, alg_n)$ and $s \cdot a \bmod f_n = 1$. If this s is taken as a rational function, the denominator involves only alg_j, j=1, $\cdots$,n−1, and not alg_n, so the algorithm calls itself recursively to calculate the inverse of this denominator in the smaller extension. When it is finally called with a trivial extension, we know that the argument is an element of the base field, either

the rationals, or a rational function field over the rationals. Then $1/a$ is returned as the inverse. The result returned is a polynomial which is the product of the inverses of the denominators, put in canonical form. It satisfies the following property:

$$inverse \cdot a = 1 + \sum_{i=1}^{n} q_i \cdot f_i$$

for some q_i, as multivariate polynomials. Since $f_i = 0$, $i=1, \cdots, n$ in the extension, $inverse \cdot a = 1$, so *inverse* is the inverse of a in the algebraic field.

The iterative version does essentially the same thing, except that it is inherently more efficient because it avoids the overhead involved in recursive function calls.

Problems With Inverses

From the analysis by Smedley (13), we see that even in the case of a simple algebraic extension of the rationals, the coefficient growth when computing inverses can be quite unacceptable (exponential in the degree of the minimal polynomial). Empirical observations show that the bound given is, in fact, tight. When computing in multiple extensions, and in algebraic function fields, the problem is even worse. Because of this it is often advantageous not to compute inverses, and just use the symbolic expression $1/\gamma$ where we need the inverse of γ. When doing this we get expressions of the form:

$$\frac{numer}{denom}$$

where *numer* and *denom* are elements of the algebraic field. As long as we keep both *numer* and *denom* in canonical form, we are assured that we can detect zero, although we do not have a unique representation for each element of the field. This is a normal form.

Some of the higher level algorithms use this approach, computing inverses only when all other calculations have been completed. In many cases this speeds up the computations considerably.

5. Basic Polynomial Operations

Once we can do the basic operations in an algebraic field, the natural thing to want to do next are the basic operations on univariate polynomials: addition, multiplication, quotient, remainder and greatest common divisor.

Addition and Multiplication

The only operations required to do addition or multiplication of polynomials over an algebraic field are addition and multiplication of the field elements. It would be possible to implement the algorithms the same as those for polynomials over any field, with the arithmetic done in the algebraic field. However, it is usually more efficient to from the sum or product completely, as multivariate polynomials, before putting any of the algebraic field elements in canonical form. This usually reduces the number of calls to the canonical form algorithm required, resulting in a more efficient algorithm.

Quotient-Remainder

Given two polynomials $u, v \in K[y]$, where K is a field, the quotient, q, and the remainder, r, of u divided by v are defined by:

$$u = q \cdot v + r \quad \textit{where } degree(r, y) < degree(v, y)$$

The most natural way to implement a quotient-remainder algorithm over an algebraic field would be to take the standard algorithm for division of polynomials over the rationals from Knuth (6), and implement it with the coefficient arithmetic done in the algebraic field. This, however, involves repeated use of the canonical form algorithm given in Chapter Three, and is very slow. It is possible to delay the reductions modulo the minimal polynomials until the very end, speeding things up considerably, since there is little difference in the size of the expressions involved in the two methods, and the canonical form algorithm can be quite expensive. Thus we have the following algorithm:

```
alg_division(u, v, y, Extension)
   lc_inverse := alg_inverse(lcoeff(v, y), Extension);
   v_monic := alg_canonical(v*lc_inverse, Extension);
   remainder := pseudo_remainder(u, v_monic, y);
   quotient := pseudo_quotient(u, v_monic, y);
   remainder := alg_canonical(remainder, Extension);
   quotient := alg_canonical(quotient*lc_inverse, Extension);
return([quotient, remainder]);
```

Since *v_monic* is monic in the variable of division, the pseudo-remainder and pseudo-quotient in the algorithm are, in fact, a real remainder and quotient. Thus, before the final two statements of the algorithm, we have:

$$u = quotient \cdot v \cdot lc\_inverse + remainder$$

as multivariate polynomials, with $degree(remainder, y) < degree(v, y)$. Putting the remainder in canonical form cannot increase its degree in y, so this degree relationship still holds. Also, since the expressions for the quotient and remainder represent the same polynomials whether they are in canonical form or not, the equation above also remains true in the algebraic field after they are put in canonical form. Thus the results returned are the true quotient and remainder, in canonical form.

Greatest Common Divisor

The algorithm implemented for calculating the greatest common divisor of two polynomials over an algebraic field is a version of the standard algorithm for computing greatest common divisors in an Euclidean domain given in Knuth (6).

```
alg_gcd(u, v, y, Extension)
   if degree(u, y) < degree(v, y) then
      swap(u, v)
   endif;
   r := alg_canonical(pseudo_remainder(u, v, y), Extension);
   while r <> 0 do
      u := v;
      v := r;
      r := alg_canonical(pseudo_remainder(u, v, y), Extension);
   enddo;
return(alg_monic(v, y, Extension));
```

The algorithm is the standard Euclidean algorithm, except that pseudo-remainders are used instead of true remainders to avoid calculation of algebraic inverses. Thus, the v computed at the end of the *while* loop is a constant in the algebraic field times the true GCD. By making v monic we get the desired result. This involves only one algebraic inverse calculation, which occurs when making v monic. Hence it is much more efficient than calculating true remainders at every stage, which involves $O(n)$ inverse calculations, where n is a bound on the degree of the polynomials. The savings are even larger since the inverse calculations involved in calculating true remainders at every stage tend to result in large intermediate expressions.

In certain cases it may not be necessary to make the GCD monic; for example, when doing a square-free test, all that is required is the degree of the GCD. In these cases a great saving can be realised by returning v without making it monic.

Extension to Multivariate Polynomials

The preceding algorithms were given for univariate polynomials. Extensions to multivariate polynomials are parallel to the extensions of the algorithms for univariate rational polynomials to multivariate ones.

Addition and Multiplication

These algorithms are essentially the same for both univariate and multivariate polynomials: form the sum or product and then put the coefficients in canonical form.

Quotient-Remainder

Since a true quotient-remainder operation cannot be performed in $K[x_1, \cdots, x_n]$, where K is a field, a pseudo-division algorithm is needed. It could be implemented essentially the same as for the univariate case, with only minor changes needed to make it a pseudo-division. In particular, we cannot calculate the inverse of the leading coefficient, as it is, in general, a polynomial, and not an element of the base field.

Greatest Common Divisor

As in the case of division, only minor changes would be required to adapt this algorithm from univariate polynomials to multivariate. The algorithm would be the same as for multivariate polynomials over the rationals, with any inverse calculations postponed for efficiency reasons.

6. Polynomial Factorisation

After the basic polynomial operations, one would like to be able to factor polynomials over an algebraic field. This is, however, somewhat more complicated, as we cannot simply adapt the algorithms for polynomials over the integers or rationals as we could with the basic operations. We can use the standard technique of transforming the problem to one we already know how to do (in this case integer polynomial factorisation) and then lifting the results to the required domain.

Square-Free Factorisation

The algorithm which is presented for factoring polynomials over an algebraic field assumes that the polynomial to be factored is square-free. Thus we require an algorithm for square-free factorisation of polynomials over an algebraic field. The simplest way to implement such an algorithm is to modify one of the standard algorithms for square-free factorisation of polynomials over the rationals. For example, one of the algorithms from Yun (18) can be adapted so that the polynomial operations are done using the algorithms from the previous chapter for polynomials over an algebraic field.

Algorithm

The following algorithm, which factors a square-free univariate polynomial over an algebraic extension, is based on the theorems involving norms presented in the section on mathematical background. It works for both simple and multiple algebraic extension fields, with the only differences being in the routine *map_to_Q*. This routine is discussed later. Also, when factoring over an algebraic function field, after the polynomial is mapped to the base field, the polynomial to be factored has coefficients which are rational functions in the variables of the algebraic functions. In order to factor this polynomial we first multiply it by the least common multiple of the denominators of the coefficients, resulting in a polynomial over the rationals. This polynomial can be factored with any standard algorithm. At the end we divide by this factor to get the required factorisation.

```
alg_factor(p, x, Extension)
   [P, transformation] := map_to_Q(p, x, Extension);
   L := factor(P)  #over the rationals
   if no_of_factors(L) = 1 then
      return(p)
   else
      p_trans := alg_canonical(subs(transformation, p), Extension);
      for i from 1 to no_of_factors(L) do
         f[i] := alg_gcd(p_trans, L[i], x, Extension);
         f[i] := undo(transformation, f[i]);
         f[i] := alg_canonical(f[i], Extension);
      enddo;
   endif;
return(f);
```

The algorithm is quite simple. First, a transformation is found which makes the norm of the polynomial to be factored square-free, and the norm, P, is calculated; then this norm is factored over the base field (the rationals or a rational function field over the rationals); the factors, $L[i]$, are lifted back to the algebraic extension field using algebraic polynomial greatest common divisors; and finally the transformation is undone to get the factors, $f[i]$, of the original polynomial. Note that, whenever a substitution is done with an algebraic quantity, the result must be put in canonical form in the extension field, since the result of the substitution may no longer be in canonical form.

Simple Extension Fields

In the case of a simple algebraic extension, the routine *map_to_Q* is exactly the routine *sqfr_norm* presented in Trager (14).

```
map_to_Q(p, x, Extension)
   #We know that Extension[1]=1 since this is a simple extension
   alg := Extension[2];
   f := Extension[3];
   s := -1;
   repeat
      s := s + 1;
      trans := x - s*alg;
      g := alg_canonical(subs(x=trans, p), Extension);
      r := resultant(g, f1, alg1);
   until gcd(r, diff(r, x)) = 1;
return([r, x = trans]);
```

Multiple Extension Fields

The most obvious way to implement the *map_to_Q* algorithm for multiple algebraic extension fields would be to use the method for simple extensions iteratively. That is, if we want to map $g(y) \in Q(\alpha_1, \cdots, \alpha_n)[y]$ to a polynomial over the rationals, we let $K = Q(\alpha_1, \cdots, \alpha_{n-1})$ and use the same method as for simple extensions to map $g(y) \in K(\alpha_n)[y]$ to $K[y]$. Then we repeat this for each of the other algebraics in the multiple extension. The algorithm would be as follows:

```
map_to_Q(p, x, Extension)
   n := Extension[1];
   alg := Extension[2];
   f := Extension[3];
   r := f;
   transformation := x;
   for i from n by -1 to 1 do
      s_extension[1] := i;
      s_extension[2] := alg_j, j=1..i;
      s_extension[3] := f_j, j=1..i;
      [r, ntrans] := square_free_norm(r, x, s_extension);
      transformation := subs(x=ntrans, transformation);
   enddo;
return([r, x = transformation]);
```

Where *square_free_norm* is;

```
square_free_norm(p, x, Extension)
   n := Extension[1];
   alg := Extension[2];
   f := Extension[3];
   s := -1;
   repeat
      s := s + 1;
      trans := x - s*alg;
      g := alg_canonical(subs(x=trans, p), Extension);
      r := resultant(g, f_n, alg_n);
   until alg_gcd(r, diff(r, x), x, Extension) = 1;
return([r, trans]);
```

Because the square-free test involves polynomials over an algebraic extension, it must do an algebraic polynomial greatest common divisor. The polynomials are, in general, dense, of high degree, and have large coefficients. The repeated computation of these greatest common divisors leads to an extremely slow algorithm.

The following algorithm avoids calculation of algebraic polynomial greatest common divisors completely. It picks a complete transformation; does all of the resultant calculations necessary to get a polynomial over the rationals; and then does the square-free test. If this test fails, a new transformation is chosen, and the process is repeated until a transformation is found which yields a square-free result. This almost always leads to more resultant calculations, but the savings realised by avoiding the algebraic polynomial greatest common divisor calculations usually outweighs this increased cost.

```
map_to_Q(p, x, Extension)
   transformation := x;
   repeat
      transformation := next_trans(transformation, x, Extension);
      g := alg_canonical(subs(x=transformation, p), Extension);
      r := multi_norm(g, Extension);
   until gcd(r, diff(r, x)) = 1;
return([r, x = transformation]);
```

The subroutine *multi_norm* computes all the resultants necessary to map the polynomials to the base field.

```
multi_norm(p, Extension)
   n := Extension[1];
   alg := Extension[2];
   f := Extension[3];
   r := p;
   for i from n by -1 to 1 do
      r := resultant(r, f_i, alg_i);
   enddo;
return(r);
```

In the above algorithm, *map_to_Q*, the function *next_trans* selects a transformation to try next. That is, it picks the s_i used in the substitution $x = x - s_1\alpha_1 - \cdots - s_n\alpha_n$. Since we can compute a bound for each of the s_i, there are only a finite number of transformations which must be tried. The algorithm is designed to go through all of the combinations, without repetitions, ensuring that a transformation which works is found. In most cases only a small number of transformations must be tried before finding one which works.

Efficiency Problems

Unfortunately, the mapping to integer polynomials used in the algebraic polynomial factorisation algorithm involves extremely large coefficient and degree growth, and it is often the case that the resulting problem is unmanageable. For example, if we want to factor $x^4 - 2$ over $Q(\sqrt{2}, \sqrt{-1})$ then we must factor the following polynomial over the integers:

$$12769+8968x^2-6668x^4-5112x^6+3078x^8-264x^{10}+52x^{12}-8x^{14}+x^{16}$$

This is a relatively simple example, and making the problem even slightly more difficult makes the integer polynomial factorisation problem much more difficult.

Wang (16) and Weinberger & Rothschild (17) give algorithms based on modular techniques for factoring polynomials over an algebraic field. However, they have problems when the minimal polynomial is reducible modulo every prime, as in the case of $x^4 + 1$, and must use a different algorithm in these cases.

Lenstra (9, 10) gives another algorithm based on lattices. The observations by Abbott et al. (1), comparing this algorithm with Trager's, give no definite conclusions as to which algorithm would be more efficient in general, and since the resultant based algorithm is well known and easy to implement, it was chosen to be included in the Maple algebraic field package, as was the case in Abbott et al. (1). It is possible that some sort of a hybrid algorithm would be best for this problem, but more investigation is required before any conclusions can be reached.

Extension to Multivariate Polynomials

The algorithm given above is for factorisation of univariate polynomials over an algebraic field, but extending it to multivariate polynomials should be fairly straightforward. Although this has not yet been implemented in Maple, it would appear that extending the previous algorithm for use with multivariate polynomials would involve only two relatively minor changes: choosing a variable to use in the transformation used to get a square-free norm; and using a multivariate version of the *alg_gcd* algorithm.

7. Simplification of Radical Expressions

One of the most obvious applications of the preceding algorithms for computation with elements of an algebraic field is simplification of expressions involving radicals. A radical is something of the form $expr^r$ where r is a rational number which is not an integer. As long as *expr* involves only algebraic expressions — that is, no logarithms, trigonometric functions, exponentials or transcendental constants such as π or e — then $expr^r$ is an algebraic quantity and the algorithms of this thesis are applicable for the computations.

We present a method for simplifying radicals which are algebraic numbers — i.e. do not contain variables. We also discuss how this can be extended to handle algebraic functions.

The Algorithm

The following is an algorithm which puts an expression involving radicals, nested to any depth, in a normal or canonical form.

```
radical_simplify(expr)
   rads_left := get_list_of_radicals(expr);
   Extension[1] := 0;
   Extension[2] := alg;
   Extension[3] := f;
   i := 0;
   substitutions := {};
   while rads_left <> [] do
      rad := head(rads_left);
      rads_left := tail(rads_left);
      if not_in_extension(rad, Extension, substitutions) then
         i := i + 1;
         Extension[1] = i;
         f_i := minimal_polynomial(rad, Extension, alg_i, substitutions);
         substitutions := substitutions union {alg_i=rad};
         expr := subs(rad=alg_i, expr);
         rads_left := subs(rad=alg_i, rads_left);
      else
         s := express_in_extension(rad, Extension, substitutions);
         expr := subs(rad=s, expr);
         rads_left := subs(rad=s, rads_left);
      endif;
   enddo;
   expr := alg_normal(expr, Extension);
   # use alg_canonical for a canonical form
   expr := full_subs(substitutions, expr);
return(expr);
```

The above algorithm works as follows: It gets all of the radicals in the expression in order of their dependencies (i.e. $5^{1/2}$ would come before $(5^{1/2} + 1)^{1/3}$); then it goes through the list, checking if each radical is already in the extension; if it is not, then it is added to the extension by calculating its minimal polynomial; if it is then its representation in the extension is calculated, and a substitution is made; when this is done, the resulting expression is simplified in the extension, and then a substitution is made to put the expression back in terms of the original radicals.

Note that the presentation given here is not an exact description of how the algorithm is implemented. In the implementation transformations are performed, as described in Chapter Two, to ensure that all the minimal polynomials are monic and have only integer coefficients. These transformations are not necessary to the correct operation of the algorithm, but are done to increase efficiency. Thus these details have been omitted from this description in order to make the algorithm easier to understand.

The subroutine *full_subs* performs full substitution of a set of substitutions in the expression. That is, the substitutions are performed repeatedly until there are no further changes in the expression: *full_subs*($\{x=y, y=z\}$, $x + y^2$) would yield $z + z^2$.

Subroutine get_list_of_radicals(expr)

This routine returns a list of the radicals contained in *expr*, ordered by their dependencies. That is, if two radicals, $expr_1^{r_1}$ and $expr_2^{r_2}$, appear in *expr*, and $expr_1$ involves $expr_2^{r_2}$, then $expr_2^{r_2}$ comes before $expr_1^{r_1}$ in the returned list.

The list returned has no repetitions, and some intelligent processing is done. For example, both $2^{1/3}$ and $2^{2/3}$ will not appear in the list, as $2^{2/3}$ is simply $(2^{1/3})^2$ and thus occurs in any algebraic field containing $2^{1/3}$. This processing is not necessary, but does increase the efficiency of the *radical_simplify* algorithm.

The exact implementation of the algorithm is not given here as it is totally dependent on the way in which expressions are represented in Maple, and would be implemented differently in another language for symbolic computation.

Subroutine not_in_extension(expr, Extension, substitutions)

This subroutine returns *true* if *expr* is not in the algebraic extension *Extension*, and *false* if it is. The algorithm is as follows:

```
algorithm not_in_extension(expr, Extension, substitutions)
   p := zero_polynomial(expr, x);
   fac_list := alg_factor(p, x, Extension);
   m := get_minimal_polynomial(expr, x, fac_list, substitutions);
return(degree(m, x) > 1);
```

The subroutine *zero_polynomial* returns a polynomial which has *expr* as a zero. E.g. $x^2 - 5$ is returned for $expr = 5^{1/2}$. Then the polynomial is factored over the algebraic extension. *Get_minimal_polynomial* returns the factor from *fac_list* which has *expr* as a root. Finally the routine returns *true* if the minimal polynomial for *expr* over *Extension* has degree greater than one (i.e. if *expr* is not in *Extension*). The information from *alg_factor* and *get_minimal_polynomial* is remembered for use in other subroutines with Maple's *option remember*.†

† When a Maple routine has *option remember*, each time it is called the value returned is saved, so that, if it is called again with the same arguments, no recomputation is required.

Subroutine get_minimal_polynomial(expr, x, fac_list, substitutions)

The routine *get_minimal_polynomial* returns the factor from *fac_list* which has *expr* as a zero under assumptions on how to evaluate radicals. This factor will be the minimal polynomial for the expression *expr*.

```
get_minimal_polynomial(expr, x, fac_list, substitutions)
   subs_fac_list := subs(x=expr, fac_list);
   subs_fac_list := full_subs(substitutions, subs_fac_list);
   index := zero_factor(subs_fac_list);
   min_poly := fac_list[index];
return(min_poly);
```

Substitutions are made on the list of factors so that it is once again in terms of radicals. The routine *zero_factor* returns the index of substituted factor which evaluates to zero under certain assumptions about how to evaluate radicals. The factor in *fac_list* which corresponds to this factor is the required minimal polynomial.

The routine *zero_factor* assumes that, when a radical is given, the branch containing the positive real axis is implied. Thus, to get the negative square root of 2, $-2^{1/2}$ must be stated. This assumption is made to make radicals single valued functions. It would also be possible to implement the radical simplification algorithm so that it took each of the branches of each of the radicals in the expression, and returned all possible simplifications resulting from the different choices of branches. This would result in an algorithm which takes exponentially more time than choosing just one branch consistently, and so the above assumption was made. It is still possible to get simplifications corresponding to other branches by specifying the branch explicitly (as in the case of $-2^{1/2}$).

This routine is currently under development. Two different approaches are being tried. The first involves probabilistic algorithms for determination of equivalence of expressions, as in Gonnet (4). Using this method we get a probabilistic algorithm for simplification of radicals. However, since the zero testing algorithm will never answer *not—equal zero* when the factor is zero, but may return *equal zero* when the factor is not zero, we can reapply the test until only one factor tests as being equal to zero, and we will know that this is the desired factor. This makes it no longer a deterministic algorithm, but the probability of it looping forever is extremely small. The only remaining problem to solve, is adapting the testing algorithm to test for equivalence under the assumption the radicals imply the branch which includes the positive real axis. In Gonnet he states that the test returns *equal* only if the relation is true for all possible branches of the radical. A solution to this is still under development.

Another approach to the problem involves interval arithmetic (12). By calculating intervals containing each of the substituted factors, and refining them until only one contains zero, the correct factor can be determined easily. This is also under development.

Subroutine minimal_polynomial(expr, Extension, substitutions)

This subroutine does essentially the same as *not_in_extension*, except that, instead of returning *true* or *false*, it returns the minimal polynomial. The routines all remember the results of their computations with Maple's *option remember* facility.

Subroutine express_in_extension(expr, Extension, substitutions)

This subroutine is called when it is known that *expr* can be expressed in *Extension*. It returns the representation of *expr*.

```
algorithm express_in_extension(expr, Extension, substitutions)
   p := zero_polynomial(expr, x);
   fac_list := alg_factor(p, x, Extension);
   m := get_minimal_polynomial(expr, x, fac_list, substitutions);
   e := alg_canonical(solve(m, x), Extension);
return(e);
```

Before calling this routine it is known that *expr* can be represented in *Extension*, and so the m calculated by *get_minimal_polynomial* is linear in x. Thus e is the expression of *expr* in *Extension*.

Subroutine alg_normal(expr, Extension)

This is the algebraic normalisation algorithm from Chapter Three. It returns the result in the form:

$$\frac{numer}{denom}$$

where *numer* and *denom* are elements of the algebraic field in canonical form, but the inverse of *denom* is not computed. In most cases this gives a "simpler" result than putting it in a canonical form, as computation of an algebraic inverse usually results in large expression swell. If a canonical form is required, the algorithm *alg_canonical* can be used instead of *alg_normal*.

Example

The following example shows how the *radical_simplify* algorithm would simplify the following expression:

$$1/2 - 1/2\sqrt{-3} + (-1/2 - 1/2\sqrt{-3})^{1/2}$$

The list of radicals returned would be:

$$[\sqrt{-3}, (-1/2 - 1/2\sqrt{-3})^{1/2}]$$

The extension would first be $Q(\alpha_1)$ where $\alpha_1 = \sqrt{-3}$, and then $x^2 + 1/2 + 1/2\alpha_1$ would be factored over the extension, giving:

$$x^2 + 1/2 + 1/2\alpha_1 = (x - 1/2 + 1/2\alpha_1)(x + 1/2 - 1/2\alpha_1)$$

Each factor would be substituted and tested for zero equivalence, resulting in:

$$factor_1 \approx 0.0$$

$$factor_2 \approx 1.0\text{–}1.732i$$

The first factor is chosen, and since it is linear, we know that the expression is in the current extension. Solving tells us that:

$$(-1/2 - 1/2\sqrt{-3})^{1/2} = 1/2 - 1/2\sqrt{-3}$$

Finally the substitution and simplification is performed to get:

$$1/2 - 1/2\sqrt{-3} + (-1/2 - 1/2\sqrt{-3})^{1/2} = 1 - \sqrt{-3}$$

Extension to Handle Variables

The previous algorithm was designed to work with algebraic numbers only. When the radicals involve variables we are dealing with algebraic functions. To adapt the algorithm to handle these cases, all that must be changed is the routine which determines the minimal polynomial. The test to determine which substituted factor is zero must be modified, as they now involve variables. It is also not obvious which factor should be chosen. For example, should $\sqrt{x^2}$ be x or $-x$? A discussion of this problem is beyond the scope of this thesis, and can be found in Caviness & Fateman (2).

Once these problems have been resolved, the routine *zero_factor* can be implemented using concepts similar to those in Gonnet (4), concerning determination of equivalence of expressions, or possibly these concepts combined with interval arithmetic as mentioned previously.

Comments and Improvements

For expressions with only a few radicals the simplifications are carried out quickly. However, if there are many radicals the algorithm has trouble in the algebraic factorisation. It is possible to greatly improve the efficiency by avoiding as many factorisations as possible.

To avoid doing a factorisation you must know either that an expression is independent of the current extension, or that it is in the extension, and you are able to calculate its representation. An example of the first method is to add all the prime roots of prime numbers first, since it is known that none of these can depend on the others. An example of the second method involves noting that expressions resulting from solving a cubic equation have certain algebraic numbers in them which can be expressed in terms of others also appearing in the expression. It is a simple matter of checking if an expression matches a certain format, and if the other required algebraic numbers are already in the extension, and then the representation of the expression in the extension can be

calculated directly. This method has been implemented, and makes simplification of expressions which involve roots of a cubic quite efficient.

Caviness & Fateman (2) give other methods for avoiding factorisations, but these apply only to radicals which are not nested.

When attempting to simplify an expression which involves many different radicals, there comes a point when the algebraic factorisations which would have to be done become essentially impossible. At this point it would be wise to issue a warning to the effect that not all algebraic dependencies may be noted, and carry on without further factorisations. If the order of adding the radicals to the extension is chosen intelligently, it may be possible to do this without missing many algebraic dependencies.

8. Concluding Remarks

We have described the theory and design behind a system for computation with elements of algebraic number and function fields as it is implemented in the Maple computer algebra system. We have also given an application of the system: namely, a method for simplification of expressions involving radicals. The system is still under development and we hope to achieve gains in efficiency through improvements in current algorithms and the development of new ones.

This thesis presents the necessary information to get started with computing in algebraic fields. There is certainly much work still to be done, and it is hoped that this thesis, by organising the background information, will assist in the development of the field.

Future Work

One stumbling block in the development of algorithms for algebraic fields is the large expression swell often encountered. This is a problem especially when computing inverses. Although this growth is unavoidable when computing an inverse, it is hoped that algorithms can be developed which avoid the explicit computation of inverses as much as possible.

The area where most development is hoped for and expected is in factoring algebraic polynomials. We currently have three different types of algorithms and it is hoped that improvements to these, or entirely new algorithms, will be discovered. In particular, extensions of the algorithms by Wang (16), Weinberger & Rothschild (17) and Lenstra (9, 10) for use in multiple algebraic extensions could be developed. Perhaps heuristics or hybrid algorithms will prove useful for this problem.

There is also much work to be done on the use of computation in algebraic fields for simplification of expressions involving radicals. There are undoubtedly many methods which can be used to avoid expensive factorisations, which have yet to be developed. Also, it would be helpful to develop ways to determine which factorisations to do when it is obvious that it will be too expensive to do all those which would be necessary to be certain that the expression is put in normal or canonical form.

Appendix — Maple Code

The following is the Maple code for the implementation of the algorithms presented in this thesis. Some of the algorithms are not implemented exactly as presented, and all are still under development. The current versions of the code are available from the author. There are know bugs in the code, and the comments do not always agree with the state of the code.

Canonical Form

```
#
# this will normalise multivariate polynomials over an algebraic
# function field
#
anormal := proc(in_f,F,X)
   local f, c, d, nf, x, ind;
   f := expand(in_f);
   ind := indets(f) minus get_indets(F);
   if ind = {} then RETURN(base_anormal(f,F,X)) fi;
   x := op(1,ind);
   nf := 0;
   while f <> 0 do
      d := degree(f,x);
```

```
      c := coeff(f,x,d);
      f := expand(f - c*x^d);
      nf := nf + anormal(c,F,X)*x^d
   od;
   nf;
end:

#
# this will normalise elements of algebraic function fields
# the input must be the result of additions or multiplications
# of elements of the field. in particular, to do a division,
# you must first calculate an inverse with ainv.
#
# this means that the denominator must be a polynomial in the
# variables for the algebraic functions, over the rationals.
#
base_anormal := proc(aa,G,X)
   local a,i,nalgs,d;
   option remember;
   nalgs := nops(G);
   normal(aa);
   if a = 0 then RETURN(0) fi;
   a := numer(aa);
   d := denom(aa);
   for i from nalgs by -1 to 1 do
      a := prem(a,G[i],X[i])
   od;
   normal(a/d);
end;
```

Inverses

```
#
#--> ainv: inverse of an algebraic number
#
#      Calling sequence:  ainv(a, G, X)
#
#      Purpose:  Computes the inverse of a in the extension field
#                defined by the grobner basis given by G and X
#
#      Input: G, X - The grobner basis and list of variables defining
#                the algebraic extension
#
#              a - algebraic number
#
#      Output: function value - inverse of a over the extension
#
#                              TJS (Jun. 1986)
#
ainv := proc(b,F,X)
   local i, inv, p, pi;
   option remember;
   p := b;
   inv := 1;
   for i from nops(X) by -1 to 1 do
      gcdex(p,F[i],X[i],'pi');
      inv := anormal(inv * numer(pi), F, X);
      p := anormal(denom(pi), F, X);
   od;
   normal(inv/p);
end;
```

Quotient and Remainder

```
#
#--> aquo: quotient of polynomials over an algebraic extension field
#
#      Calling sequence:  aquo(a, b, x, G, X, 'r')
#
#      Purpose:  Computes the quotient of a and b over the extension field
#                defined by the grobner basis given by G and X.
#
#      Input: G, X - The list of polynomials and list of variables
#                defining the grobner basis for the extension
#
#              a, b - univariate polynomials over the algebraic
#                extension field. Represented by bivariate polynomials.
#
#              x - an indeterminate with respect to which the division
#                is done.
#
#      Output: function value - remainder of a and b over the extension
#
#              'r' - (call-by-name) the remainder (optional)
#
#                              TJS (Jun. 1986)
#
aquo := proc(a,b,x,G,X,remai)
        local r, q;
        r := arem(a,b,x,G,X,q);
        if nargs > 5 then remai := r; fi;
        q;
end:
#
#--> arem: remainder of polynomials over an algebraic extension field
```

```
#
#       Calling sequence:  arem(a, b, x, G, X, 'q')
#
#       Purpose:  Computes the remainder of a and b over the extension field
#       defined by the grobner basis given by G and X.
#
#       Input: G, X - The list of polynomials and list of variables
#       defining the grobner basis for the extension
#
#          a, b - univariate polynomials over the algebraic
#       extension field. Represented by bivariate polynomials.
#
#          x - an indeterminate with respect to which the division
#       is done.
#
#       Output: function value - remainder of a and b over the extension
#
#          'q' - (call-by-name) the quotient (optional)
#
#                                TJS (Dec. 1985)
#
arem := proc(ia,ib,x,G,X,quoa)
   local r, m, mi, q, l, a, b, lci;
   b := ib;
   a := ia;
   if degree(b,x)=0 then
      if nargs > 5 then
         quoa := anormal( ainv(b,G,X) * a,G,X)
      fi;
      r := 0;
   else
      b := mon(b,x,G,X,'lci');
      l := denom(a)*denom(b);
      a := numer(a)*denom(b);
```

```
      b := numer(b)*denom(a);
      r := prem(a,b,x,'m','q');
      r := normal(r/l);
      mi := ainv(m,G,X);
      if nargs > 5 then
         quoa := anormal(q*lci*mi,G,X)
      fi;
      r := anormal(mi*r,G,X);
   fi;
   collect(r,x);
end:
```

Greatest Common Divisor

```
#
#--> agcd: gcd of polynomials over an algebraic extension field
#
#      Calling sequence:  gcda(a, b, G, X)
#
#      Purpose:  Computes the gcd of a and b over the extension field
#                defined by the polynomial m.
#
#      Input: G, X - The list of polynomials and list of variables
#                defining the grobner basis for the extension
#
#              a, b - univariate polynomials over the algebraic
#                extension field. Represented by bivariate polynomials.
#
#      Output: function value - gcd of a and b over the extension
#
#                              TJS (Jun. 1986)
```

```
#
agcd := proc(a,b,G,X)
        local ind, x, u, v, r, t;
        ind := indets(a) minus get_indets(G);
        if ind = {} then
                v := 1;
        else
                x := op(1,ind);
                u := collect(b,x);
                v := collect(a,x);
                if degree(u,x) < degree(v,x) then
                        t := u;
                        u := v;
                        v := t;
                fi;
                r := prem(u,v,x);
                r := anormal(r,G,X);
                content(r,x,'r');
                while r <> 0 do
                        u := v;
                        v := r;
                        r := prem(u,v,x);
                        r := collect(anormal(r,G,X),x);
                        content(r,x,'r');
                od;
        fi;
        mon(v,x,G,X);
end:
```

Factorisation

```
#
#--> factora: Factor a polynomial over an algebraic extension
#
#      Calling sequence:  factora(f, G, X)
#
#      Purpose:  Factors the polynomial f over the algebraic extension
#                defined by G and X.
#
#      Input: G, X - The list of polynomials and list of variables
#                defining the grobner basis for the extension
#
#              f - univariate polynomial over the algebraic extension
#                field. Represented by a bivariate polynomial.
#
#      Output: function value - f, factored over the extansion
#
#                              TJS (Jun. 1986)
#
afactor := proc(inf, G, X)
        local x, r, s, l, i, g, h, hi, f, lf, fs, j, mf;
        x := op(1,indets(inf) minus get_indets(G));
        mf := mon(inf,x,G,X,'lf');
        mf := expand(mf/icontent(mf));
        f := factor(mf);
        if f = mf then
                fs := {f}
        else
                fs := {op(f)}
        fi;
        h := 1;
        for j from 1 to nops(fs) do
```

```
                f := op(j,fs);
                f := collect(f,x);
                if degree(f,x) > 1 then
                        r := map_to_Q(f,x,G,X,'g','s');
                        r := r/icontent(r);
                        l := factor(r);
                        if l = r then
                                hi := mon(f,x,G,X);
                                h := h*hi
                        else
                                g := collect(g,x);
                                for i from 1 to nops(l) do
                                        hi := collect(op(i,l),x);
                                        hi := agcd(hi,g,G,X);
                                        hi := subs(x = s,hi);
                                        hi := anormal(hi,G,X);
                                        hi := mon(hi,x,G,X);
                                        h := h * hi
                                od
                        fi;
                else
                        hi := mon(f,x,G,X);
                        h := h*hi
                fi;
        od;
        lf*h;
end:
```

Norm Subroutine

```
map_to_Q := proc(f, x, G, X, go, so)
   local n_algs, sl, sf, subn, r, i, s, rp,gi,xi;
   n_algs := nops(X);
   sl := [[0,1$'i'=2..n_algs],{},3,3];
   sf := x;
   while degree(sf,x) > 0 do
      sl := next_subs(sl);
      subn := sl[1];
      s := x = x - sum('subn[i]*X[i]','i'=1..n_algs);
      r := subs(s,f);
      for i from n_algs by -1 to 1 do
         rp := r;
         gi := G[i];
         xi := X[i];
         r := resultant(rp,G[i],X[i])
      od;
      sf := gcd(r,diff(r,x));
   od;
   go := anormal(subs(s,f),G,X);
   so := x + sum('subn[i]*X[i]','i'=1..n_algs);
   r;
end:
```

Next Substitution Subroutine

```
next_subs := proc(is_list)
   local ll, inc, lim, counted, s_list, finished, incremented, i;
   inc := is_list[4];
   lim := is_list[3];
   counted := is_list[2];
   s_list := is_list[1];
   ll := nops(s_list);
   finished := false;
   while not finished do
      incremented := false;
      for i to ll while not incremented do
         if s_list[i] < lim then
            s_list := subsop(i=s_list[i]+1,s_list);
             finished := counted intersect {s_list} = {};
            incremented := true
         else
            s_list := subsop(i=1,s_list)
         fi
      od;
      if not incremented then
        lim := lim + inc;
        s_list := subsop(('i'=1)$'i'=1..ll,s_list);
      fi;
   od;
   counted := counted union {s_list};
   [s_list,counted,lim,inc];
end;
```

Radical Simplification

The implementation does not correspond to the presentation. It is doubtful that the following will be useful, but it is presented for completeness. There are known bugs, and there is still much developmental work to be done to implement all of the concepts presented in the thesis.

Top Level Routine

```
rsimp := proc(x)
   local a, alg, i, l, sseq, xs;
   l := mkextn(x,'alg','a','xs');
   xs := anormal(expand(xs),l[1],l[2]);
   sseq := NULL;
   for i to nops(convert(alg,list)) do
      sseq := a.i = alg[i], sseq;
   od;
   subs(sseq,xs);
end:
```

Rearrange Structure of Radicals

```
normrads := proc(iexpr)
#
# This will put the radicals in an expression in a "normal" form. A
# radical of the form  a^(p/q)  will be returned in the form
# [a,q]^p.
#
local expr, r, q, pow;
options remember;
   expr := expand(iexpr);
   if type(expr,rational) then
      expr
   elif type(expr,'+') then
      map(normrads,expr)
   elif type(expr,'^') then
      pow := op(2,expr);
      if type(pow,integer) then
         normrads(op(1,expr))^pow
      elif type(pow,fraction) then
         q := op(1,pow);
         r := op(2,pow);
         powlist([normrads(op(1,expr)),r],q)
      else
         ERROR('invalid arguments')
      fi
   elif type(expr,'*') then
      map(normrads,expr);
   else
      expr
   fi;
end:
```

Get a List of the Radicals

```
getrads := proc(expr)
#
# This will take an expression which has had the radicals put in a
# "normal" form with normrads, and return a list of the radicals
# contained in the expression
#
if type(expr,'+') or type(expr,'*') then
    op(map(getrads,[op(expr)]))
elif type(expr,'^') then
    getrads(op(1,expr))
elif type(expr,list) then
    (getrads(expr[1])), expr
else
   NULL
fi;
end:
```

Remove Multiple Occurrences

```
#
# This takes a list of radicals and removes multiple occurrences
# without disturbing the order
#
radlist := proc(expr)
local rads, newrads, rad;
rads := [getrads(expr)];
map(proc(x) if type(x[1],rational) then x else NULL fi end, rads);
rads := [op(''),op(rads)];
newrads := [];
while rads <> [] do
   rad := rads[1];
   newrads := [op(newrads),rad];
   rads := map(proc(x,r) if x = r then NULL else x fi end, rads, rad)
od;
newrads;
end:
```

Miscellaneous Subroutines

```
nirads := proc(iexpr)
#
# This will put products of roots of rational numbers in a reasonable
# form
#
local expr, r, q, pow, b;
option remember;
   expr := subs(I=(-1)^(1/2),expand(iexpr));
   if type(expr,rational) then
      expr
   elif type(expr,'+') then
```

```
      map(nirads,expr)
   elif type(expr,'^') then
      pow := op(2,expr);
      b := op(1,expr);
      if type(pow,integer) then
         nirads(op(1,expr))^pow
      elif type(pow,fraction) then
         q := op(1,pow);
         r := op(2,pow);
         if type(b,rational) then
            (b^sign(q))^(abs(q)/r)
         else
            nirads(op(1,expr))^(q/r)
         fi
      else
         ERROR('invalid arguments')
      fi
   elif type(expr,'*') then
      map(nirads,expr);
   else
      expr
   fi;
end:

powlist := proc(l,p)
#
# this raises a list to a power because maple is too stupid to realise
# that it knows how to do it
#
local t;
subs(t=l,t^p)
end:

rsimplify := proc(x)
```

```
   local n;
   n := expand((ifactor(numer(op(1,x)))/ifactor(denom(op(1,x))))^op(2,x));
   d := op(2,op(2,x));
   if irem(d,2) = 1 then
     n := subs((-1)^(1/d) = -1, n)
   fi;
   n;
end:
```

Construct the Algebraic Extension

```
mkextn := proc(e,alg,a,oes)
#
# this will take an expression and figure out an algebraic extension
# for it.
#
local talg, G, X, es, expon, i, mpoly, nalg, numpolys, rl, alist;
es := nirads(e);
es := normrads(es);
rl := radlist(es);
X := [a.1];
alist := rl[1];
alg[1] := alist[1]^(1/alist[2]);
expon := alist[2];
es := subs(alist=a.1,es);
rl := subs(alist=a.1,rl);
G := [a.1^expon - alist[1]];
numpolys := 1;
for i from 2 to nops(rl) do
   alist := rl[i];
   rl := subs(alist=a.(numpolys+1),rl);
```

```
   es := subs(alist=a.(numpolys+1),es);
   expon := alist[2];
   nalg := anormal(alist[1],G,X);
   if expon = 3 and rcubic(nalg,G,X,alg,'talg') then
      mpoly := a.(numpolys+1) - talg
   else
      mpoly := a.(numpolys+1)^expon - nalg;
      mpoly := afactor(mpoly,G,X);
      mpoly := mdegfac(mpoly,a.(numpolys+1),nalg^(1/expon),X,alg);
   fi;
   if degree(mpoly,a.(numpolys+1))=1 then
      solve(mpoly,a.(numpolys+1));
      es := subs(a.(numpolys+1)=",es);
      rl := subs(a.(numpolys+1)="",rl)
   else
      G := [op(G),mpoly];
      X := [op(X),a.(numpolys+1)];
      alg[numpolys+1] := nalg^(1/expon);
      numpolys := numpolys+1
   fi
od;
oes := es;
[G,X];
end:
```

First Attempt at the Routine "zero_factor"

```
#
mdegfac := proc(f,x,rad,vars,rads)
local mval, mdop, i, opi, sseq, val, len, mlen;
if type(f,'*') then
   sseq := NULL;
   for i to nops(vars) do
      sseq := vars[i]=rads[i],sseq;
   od;
   mval := -1;
   for i to nops(f) do
      opi := op(i,f);
      if type(opi,'^') then opi := op(1,opi) fi;
      evalf(evalc(subs(x=rad,sseq,opi)));
      val := max(abs(coeff(",I,0)),abs(coeff(",I,1)));
      len := length(val);
      if mval = -1 then
         mval := val;
         mlen := len;
         mdop := opi;
      elif val < mval and len < mlen then
         mval := val;
         mlen := len;
         mdop := opi;
      elif val < mval or len < mlen then
         ERROR('cannot figure out minimal polynomial',val,mval)
      fi;
   od
elif type(f,'^') then
   mdop := op(1,f)
else
   mdop := f
```

```
fi;
mdop;
end:
```

Special Processing For Roots of Cubics

```
rcubic := proc(x,G,X,alg,expr)
   local a, b, found, i, ind, indx, k, tr, var;
   ind := indets(x);
   if nops(ind) <> 1 then RETURN(false) fi;
   var := op(1,ind);
   if not(type(x,linear,var) and type(x,polynom,var,rational)) then
      RETURN(false)
   fi;
   for i to nops(X) do
      if X[i]=var then indx := i; break fi
   od;
   b := alg[indx];
   if not(op(2,b)=1/2 and type(op(1,b),rational))
      then RETURN(false)
   fi;
#
# we now know that x is of the form a + k*var where a and k are
# rational, and var is an algebraic number which is a square root.
#
   a := coeff(x,var,0);
   k := coeff(x,var,1);
   found := false;
#
# now check if (a - k*var)^(1/3) occurs already
#
```

```
   for i to nops(G) do
      if degree(G[i])=3 and normal(X[i]^3-a+k*var-G[i])=0 then
         indx := i;
         found := true;
         break
      fi
   od;
   if not found then RETURN(false) fi;
#
# now check that a^2 - k^2 * var^2 has a rational third root
#
   tr := rsimplify((a^2 - k^2 * op(1,b))^(1/3));
   if not(type(tr,rational)) then RETURN(false) fi;
#
# we now know that x^(1/3) is already in the extension, and its
# representation is; tr / (a - k*var)^(1/3), where the denominator
# is in fact X[indx].
#
   expr := tr*ainv(X[indx],G,X);
   true;
end:
```

References

1 J.A. Abbott, R.J. Bradford, and J.H. Davenport, The Bath Algebraic Number Package, *Bath Computer Science Technical Report No. 2*, University of Bath, (1986).

2 B.F. Caviness and R.J. Fateman, Simplification of Radical Expressions, *Proceedings SYMSAC 1976*, pp. 329-338 Association for Computing Machinery, (1976).

3 B.W. Char, K.O. Geddes, G.H. Gonnet, and S.M. Watt, *Maple User's Guide*, WATCOM (1985).

4 G.H. Gonnet, New Results for Random Determination of Equivalence of Expressions, *Proceedings SYMSAC '86*, pp. 127-131 Association for Computing Machinery, (1986).

5 T.W. Hungerford, *Algebra*, Springer (1974).

6 D.E. Knuth, *The Art of Computer Programming, Vol II, Seminumerical Algorithms*, Addison Wesley (1981).

7 L. Kronecker, Grundzüge einer arithmetischen Theorie der algebraischen Grössen, *Journal für reine und angewandte Mathematik* **92** pp. 1-122 (1882).

8 S. Landau, Factoring Polynomials Over Algebraic Number Fields, *SIAM Journal of Computing* **14:1** pp. 184-195 Society for Industrial and Applied Mathematics, (1985).

9 A.K. Lenstra, Lattices and Factorization of Polynomials over Algebraic Number fields, *Proceedings EUROCAM '82, Lecture Notes in Computer Science* **144** pp. 32-39 Springer, (1982).

10 A.K. Lenstra, Factoring Polynomials Over Algebraic Number Fields, *Proceedings EUROCAL '83, Lecture Notes in Computer Science* **162** pp. 245-254 Springer, (1983).

11 R. Loos, Computing in Algebraic Extensions, *Computing Suppl.* **4** pp. 173-187 Springer, (1982).

12 R.E. Moore, *Interval Analysis,* Prentice-Hall (1966).

13 T.J. Smedley, Coefficient Growth Bounds for Algorithms on Polynomials Over Algebraic Extension Fields, *Research Report CS-86-46,* University of Waterloo, (1986).

14 B.M. Trager, Algebraic Factoring and Rational Function Integration, *Proceedings SYMSAC 1976,* pp. 219-226 The Association for Computing Machinery, (1976).

15 B.L. van der Waerden, *Modern Algebra,* Frederick Ungar (1941).

16 P.S. Wang, Factoring Multivariate Polynomials over Algebraic Number Fields, *Mathematics of Computation* **30:134** pp. 324-336 American Mathematical Society, (1976).

17 P.J. Weinberger and L.P. Rothschild, Factoring Polynomials Over Algebraic Number Fields, *ACM Transactions on Mathematical Software* **2** pp. 335-350 Association for Computing Machinery, (1976).

18 D.Y.Y. Yun, On Square-Free Decomposition Algorithms, *Proceedings SYMSAC 1976*, pp. 26-35 The Association for Computing Machinery, (1976).