Computable Fields and Galois Theory

Russell Miller[*]

June 12, 2008

1 Introduction

An irreducible polynomial has a solution in radicals over a field $\mathcal{F}$ if and only if the Galois group of the splitting field of the polynomial is solvable. This result is widely considered to be the crowning achievement of Galois theory, and is often the first response when a mathematician wants to describe the beauty of mathematics. Yet with the development of a rigorous theory of algorithms, we can ask further questions about the process of finding roots of a polynomial. Are there methods other than solution in radicals which might suffice? For that matter, when one does not even know which radicals are contained in a given field, how useful is it to have a solution in radicals?

In this article we begin to address such questions, using computability theory, in which we determine, under a rigorous definition of algorithm, which mathematical functions can be computed and which cannot. The main concepts in this area date back to Alan Turing [15], who during the 1930's gave the definition of what is now called a *Turing machine*, along with its generalization, the *oracle Turing machine.* In the ensuing seventy years, mathematicians have developed a substantial body of knowledge about computability and the complexity of subsets of the natural numbers. It should be noted that for most of its history, this subject has been known as *recursion theory*; the terms *computable* and *recursive* are to be treated as interchangeable.

Computable model theory applies the notions of computability theory to arbitrary mathematical structures. Pure computability normally considers

[*]The author was partially supported by PSC-CUNY grants numbered PSCREG-38-967, 68470-00-37, and 80209-04-12.

functions from $\mathbb{N}$ to $\mathbb{N}$, or equivalently, subsets of finite Cartesian products $\mathbb{N} \times \cdots \times \mathbb{N}$. Model theory is the branch of logic in which we consider a structure (i.e. a set of elements, called a domain, with appropriate functions and relations on that domain) and examine how exactly the structure can be described in our language, using symbols for those functions and relations, along with the usual logical symbols such as negation, conjunction, $(\exists x)$, and $(\forall x)$. To fit this into the context of computability, we usually assume that the domain is $\mathbb{N}$, the natural numbers (including 0), so that the functions and relations become the sort of objects studied by computability theorists. In this article, we mostly consider the specific case of a computable field, which will be defined below, after a brief introduction to computability.

2 Basic Computability Theory

We provide here definitions, emphasizing intuition, and some basic theorems. Rigorous versions can be found in any standard text on computability, including [6], [12], and [13].

For our purposes, a *Turing machine* is an ordinary computer, operating according to a finite program, which accepts a natural number as input and runs its program in discrete steps on that input. The machine has arbitrarily much memory available to it (on a tape, in the usual conception), but in one step it can only write a single bit (0 or 1) in a single location, and so after finitely many steps, it will only have used finitely much of its memory. One specific instruction in the program tells the machine to halt; if this instruction is ever reached, then the program beeps to tell us that it is finished, and its output is the total number of 1's written on its memory tape. Since only finitely many steps can have taken place so far, this output must also be a natural number. Computer scientists worry exceedingly about how many steps are required for the machine to halt, and how much of the memory tape is used before it halts, but for a computability theorist, the principal question is whether the machine ever halts or not, and if so, what its output is. Of course, the instructions can go into an infinite loop, or avoid the halting instruction in other ways, so it is quite possible that a program on a given input never halts at all.

A function $f : \mathbb{N} \to \mathbb{N}$ is said to be *computable* if there is a Turing machine which computes f. Specifically, on each input $n \in \mathbb{N}$, the program should eventually halt with output $f(n)$. More generally, we consider *partial* *functions* $\varphi : \mathbb{N} \to \mathbb{N}$, for which (despite the similarity of notation) the domain is allowed to be any subset of $\mathbb{N}$. The possibility that a program never halts makes the partial functions a more natural class for our definitions, since every Turing machine computes some partial function, namely that φ whose domain is the set of inputs on which the machine halts, with $\varphi(n)$ being the output of the machine for each such input n. We write $\varphi(n)\downarrow$, and say that $\varphi(n)$ *converges*, if $n \in \mathrm{dom}(\varphi)$; otherwise $\varphi(n)$ *diverges*, written $\varphi(n)\uparrow$.

The importance of the restriction that a Turing machine must use a *finite* program is now clear, for an infinite program could simply specify the correct output for each possible input. On the other hand, with this restriction, we have only countably many programs in all: there are only finitely many possible instructions, so for each $n \in \mathbb{N}$, only finitely many programs can contain exactly n instructions. Hence only countable many partial functions are computable, with the remaining (uncountably many) ones being noncomputable. It is not hard to define a noncomputable function: just imitate Cantor's diagonal proof that $\mathbb{R}$ is uncountable.

A subset $S \subseteq \mathbb{N}$ is said to be *computable* iff its characteristic function χ_S is computable. (Of course, the characteristic function is *total*, i.e. its domain is all of $\mathbb{N}$.) Easy examples include the finite sets, the eventually periodic sets, the set of prime numbers, and any set which can be defined in the language of arithmetic without using unbounded quantifiers $\forall$ and $\exists$. (For more of a challenge, find a set which is computable but cannot be defined using only bounded quantifiers.) Another useful concept is *computable enumerability*: S is computably enumerable, abbreviated *c.e.*, if it is empty or is the range of some total computable function f. Intuitively, this says that there is a mechanical way to list out the elements of S: just compute $f(0)$, then $f(1)$, etc., and write each one on our list. Computably enumerable sets are "semi-computable," in the following sense, which the reader should try to prove.

Fact 2.1 *A subset $S \subseteq \mathbb{N}$ is computable iff both S and its complement $\overline{S}$ are computably enumerable.*

Fact 2.2 *A set S is computably enumerable iff S is the range of a partial computable function, iff there is a computable set R such that $S = \{x : \exists y_1 \cdots \exists y_k\ (x, y_1, \dots y_k) \in R\}$, iff S is the domain of a partial computable function.*

For the R in this fact, we need to consider subsets of Cartesian products

$\mathbb{N}^k$ as well. For this we use the function

$$\beta_2(x, y) = \frac{1}{2}(x^2 + y^2 + x + 2xy + 3y),$$

which is a bijection from $\mathbb{N}^2$ onto $\mathbb{N}$. (In checking this, bear in mind that for us 0 is a natural number.) β_2 feels computable; it would be computable if we allowed the input (x, y) to be simulated on the input tape by x 1's, followed by a 0 and then y 1's. Alternatively, notice that if π_1 and π_2 are projections, then both functions $\pi_i \circ \beta_2^{-1}$ are computable, and so given x and y, we could search through all possible outputs $n \in \mathbb{N}$ until we found one for which $\pi_1(\beta_2^{-1}(n)) = x$ and $\pi_2(\beta_2^{-1}(n)) = y$. So we may use this bijection β_2 to treat $\mathbb{N}^2$ as though it were just $\mathbb{N}$, and then define $\beta_3(x, y, z) = \beta_2(x, \beta_2(y, z))$ and so on. Indeed, the bijection β defined by

$$\beta(x_1, \ldots, x_k) = \beta_2(k, \beta_k(x_1, \ldots, x_k))$$

maps the set $\mathbb{N}^*$ of all finite sequences of natural numbers bijectively onto $\mathbb{N}$. This gives us a way of allowing polynomials from $\mathbb{N}[X]$ to be the inputs to a computable function.

Now that we know how to handle tuples from $\mathbb{N}$ as inputs, we can see that there is a *universal Turing machine*. The set of all possible programs is not only countable, but can be coded bijectively into the natural numbers in such a way that a Turing machine can accept an input $e \in \mathbb{N}$, decode e to figure out the program it coded, and run that program. The universal Turing machine accepts a pair (e, x) as input, decodes e into a program, and runs that program on the input x. It defines a partial computable function $\varphi : \mathbb{N}^2 \to \mathbb{N}$ which can imitate every partial computable function ψ: just fix the correct e, and we have $\psi(x) = \varphi(e, x)$ for every $x \in \mathbb{N}$ (and with $\psi(x)\uparrow$ iff $\varphi(e, x)\uparrow$, moreover). This enables us to give a computable list of all partial computable functions, which we usually write as $\varphi_0, \varphi_1, \ldots$, with $\varphi_e(x) = \varphi(e, x)$ for a fixed universal partial computable function φ. In contrast, there is no computable list of all the total computable functions (i.e. those with domain $\mathbb{N}$); if there were, one could use it to diagonalize and get a new total computable function not on the list!

Likewise, using Fact 2.2, this yields a computable list of all computably enumerable sets $W_0, W_1, \ldots$, with $W_e = \operatorname{dom}(\varphi_e)$. We can view them as the rows of the universal c.e. set $W = \operatorname{dom}(\varphi) = \{(e, x) : \varphi_e(x)\downarrow\} \subseteq \mathbb{N}^2$. On the other hand, there is no such computable listing of all computable sets.

The principal remaining fact we will need is simply stated.

Fact 2.3 *There exists a c.e. set which is not computable.*

A simple definition of one such is

$$K = \{e : \varphi_e(e)\!\downarrow = 0\}.$$

The idea is that φ_e cannot equal χ_K, because $e \in K$ iff $\varphi_e(e) = 0$ iff φ_e guesses that e is *not* in K. Of course, if $\varphi_e(e)$ never converges, we never add e to K, and again we see that $\chi_K \neq \varphi_e$, simply because $e \in \mathrm{dom}(\chi_K)$. Many similar sets can be defined; the famous one, the domain of the function φ computed by the universal Turing machine, is usually called the *Halting Problem*, since it tells you exactly which programs converge on exactly which inputs. If it were computable, then we could use it to compute K, which is impossible. Indeed, if $\mathrm{dom}(\varphi)$ were computable, then every c.e. set would be computable.

3 Computable Fields

When considering fields, we normally work in a language which includes the addition and multiplication symbols, regarded as binary functions, and two constant symbols to name the identity elements, along with all standard logical symbols. A field $\mathcal{F}$ then consists of a set F of elements (called the *domain* of the field, but not to be confused with the separate notion of a ring without zero-divisors), with two elements of F distinguished as the two constants, and with two binary functions on F, represented by the symbols $+$ and $\cdot$, all satisfying the standard field axioms.

For a field to be computable, we want to be able to compute the two field operations in this language. Our notion of computability is defined over $\mathbb{N}$, so we index the elements of the field using $\mathbb{N}$. Of course, this immediately eliminates all uncountable fields from the discussion! Section 7 mentions a possible approach to this problem.

Definition 3.1 A *computable field* is a field $\mathcal{F}$ with domain $\{a_0, a_1, \ldots\}$ and with two computable total functions f and g such that for all $i, j \in \mathbb{N}$,

$$a_i + a_j = a_{f(i,j)} \quad \text{and} \quad a_i \cdot a_j = a_{g(i,j)}.$$

This simply says that we can compute the basic field operations in $\mathcal{F}$ using Turing machines. In fact, in most of computable model theory one takes $\mathbb{N}$

itself to be the domain. However, we will wish to use the symbols 0 and 1 to refer to the identity elements of the field (and perhaps 2 to refer to the sum $1 + 1$, etc.), so we use the notation a_i to avoid confusion.

Since we have constant symbols $0 = a_r$ and $1 = a_s$ for some r and s, we ought officially to say that these numbers r and s are computable as well. However, any single number is in some sense always computable; we can add this information to any finite program and still have a finite program. If the language had happened to have infinitely many constant symbols, presumably indexed somehow by $\mathbb{N}$, then we would have required that their values in the domain be computable from those indices. Moreover, we have a stronger answer to this question: knowing how to compute f and g allows us to identify the identity elements. Just compute $f(0,0), f(1,1), \ldots$ until you find the (unique) r with $f(r,r) = r$, and then a_r must be the zero in $\mathcal{F}$. Similarly, 1 is the unique element $a_s \neq a_r$ for which $g(s,s) = s$.

A related question is whether it matters that we did not include other field operations, such as subtraction or reciprocation, in our language. If we had included them, then we would require those operations to be computable as well, of course. However, our definition was equivalent.

Lemma 3.2 *In a computable field, the unary operations of negation and reciprocation and the binary operations of subtraction and division are all computable.*

Proof. To find the negative of any a_n, just compute $f(0,n), f(1,n), \ldots$ until you find an m with $f(m,n)$ equal to that r with $a_r = 0$. Then a_m is the negative of a_n, and defining subtraction is now easy as well. Reciprocation is similar, using g, and division follows, except that now the program must check that the input n is not r itself. (Otherwise the search would continue forever!) If the input is r, the program for reciprocation should just output r itself, or some other artificial device to indicate that it has detected division by zero and does not intend to follow through on its search. ■

As a first example, there is a computable field $\mathcal{F}$ isomorphic to the field $\mathbb{Q}$ of rational numbers. For this we wish to think of each a_i as a fraction with integer numerator and natural-number denominator, without letting our domain repeat any fractions. Since we have the computable bijection β_2 from Section 2, we can define $h(0) = 2 = \beta_2(0,1)$ and $h(n+1)$ to be the least $k > h(n)$ for which $\pi_1(\beta_2^{-1}(k))$ is relatively prime to $\pi_2(\beta_2^{-1}(k))$. This allows us to define computable functions $\texttt{num}(2n) = \pi_1(\beta_2^{-1}(h(n)))$ and $\texttt{denom}(2n) =$

$\pi_2(\beta_2^{-1}(h(n)))$ for all $n \in \mathbb{N}$, giving the numerators and denominators of the field elements $a_0, a_2, a_4, \ldots$. We treat a_{2n+1} as the negative of a_{2n+2}, and define addition and multiplication on the domain $\{a_0, a_1, a_2, \ldots\}$ by doing arithmetic on fractions (which *does* follow a simple algorithm, all experience teaching students to the contrary!).

The reader is invited to attempt to build computable fields of isomorphism types such as $\mathbb{Q}(X)$, $\mathbb{Q}(\sqrt{2})$, $\mathbb{Q}(X_1, X_2, \ldots)$, and other well-known countable fields. $\mathbb{Z}_p(X_1, X_2, \ldots)$ is also possible, of course. Definition 3.1 ought to be tweaked to allow finite fields, of course, since such fields in some sense must be computable, with the algorithms for the field operations being given by finite tables of values. We avoided this in order to keep the definition simple, but it would be better to allow domains of the form $\{a_0, \ldots, a_m\}$ as well.

Notice, however, that it is quite possible for a computable field to be isomorphic to a field that is not computable. Strictly speaking, $\mathbb{Q}$ itself is not computable, since its domain is not in the proper form. More than this, however, there are isomorphic copies of $\mathbb{Q}$ with domain $\{a_0, a_1, \ldots\}$ in which the operations are not computable. Indeed, any of the uncountably many permutations of $\mathbb{N}$ (i.e. bijections from $\mathbb{N}$ onto itself) produces a distinct copy of the same field, with the same domain but the operations lifted via the permutation, and almost all of these are noncomputable. So we should not say that the isomorphism type of a field is computable; rather we say that a field (or its isomorphism type) is *computably presentable* if it is isomorphic to a computable field.

Indeed, we already have the tools to build a countable field which is not computably presentable. Consider the noncomputable c.e. set K from Fact 2.3. Write p_n for the n-th prime number ($p_0 = 2$, $p_4 = 11$, etc.), and let $\mathcal{E}_{\overline{K}}$ be the following extension of $\mathbb{Q}$:

$$\mathcal{E}_{\overline{K}} = \mathbb{Q}[\sqrt{p_n} \ : \ n \notin K].$$

Now the set of primes is computable, and so in any field of characteristic 0, it is easy to list out the prime numbers $p_0, p_1, \ldots$, just by adding 1 to itself. (Specifically, the function h with $p_n = a_{h(n)}$ is computable.) If $\mathcal{F}$ were a computable field isomorphic to $\mathcal{E}_{\overline{K}}$, then the following process would contradict the noncomputability of K. Each time a field element appears in $\mathcal{F}$ whose square equals p_n for any n, enumerate that n into a set W. By the definition of $\mathcal{E}_{\overline{K}}$, this W would equal the complement $\overline{K}$, and we would have a computable enumeration of this complement, which is impossible, by Fact 2.1.

In light of this, it may seem surprising that the field $\mathcal{E}_K = \mathbb{Q}[\sqrt{p_n} : n \in K]$ is computably presentable. Yet it is: we will build a computable presentation of $\mathcal{E}_K$. To "build" a field usually means that we give finitely much of it at a time, first defining the addition and multiplication functions only on the domain $\{a_0, \ldots, a_j\}$, then extending them to $\{a_0, \ldots, a_{j+1}\}$, and so on. We do this according to an algorithm, and if we wish later to compute the field addition or multiplication, we simply run this same algorithm until it defines the sum or product we seek. (Of course, our algorithm must decide within finitely much time which a_k is to be the sum $a_i + a_j$; and once it has decided, it may not change its mind!)

So we start building a computable presentation of $\mathbb{Q}$, similar to the one given above, and simultaneously start enumerating K. (Think of a timesharing procedure, allowing us to do both these processes at once.) Whenever a new element n appears in K, we continue building our field until the element p_n has appeared in it, then stop for long enough to define the multiplication so that the next new element is the square root of p_n. Then we continue building the field, always treating this new element as the square root of p_n when defining the addition and multiplication. Since every $n \in K$ eventually appears in our enumeration, this builds a computable field isomorphic to $\mathcal{E}_K$.

The key here is that the statement "p_n has a square root" is an existential formula, with free variable n:

$$(\exists x)[x \cdot x = (1 + 1 + \cdots + 1) \quad (p_n \text{ times})].$$

The statement within the square brackets defines a computable set, since one can rewrite the $(1 + 1 + \cdots + 1)$ as $a_{h(n)}$, with h as defined just above. Therefore, in a computable field, the set of numbers n satisfying this existential formula is a computably enumerable set, by Fact 2.2. K itself is c.e., so having this set equal K (as in $\mathcal{E}_K$) is not a problem. Indeed, we could build a similar field $\mathcal{E}_W$ for any c.e. set W. However, since the complement $\overline{K}$ is not c.e., the field $\mathcal{E}_{\overline{K}}$ is not computably presentable.

$\mathcal{E}_K$ is not without its complications, however. A standard question for a field $\mathcal{F}$ is the existence of a *splitting algorithm* for $\mathcal{F}$. That is, given a polynomial $p(X) \in \mathcal{F}[X]$, we want an algorithm which produces the irreducible factors of $p(X)$ in $\mathcal{F}[X]$. Of course, if one knows that $p(X)$ is not itself irreducible, then one can search through $\mathcal{F}[X]$ for a nontrivial factorization (using our function β from Section 2 to list out all elements of $\mathcal{F}[X]$), and then repeat the process recursively for each factor. So the splitting algorithm comes down to being able to decide whether a given polynomial is irreducible or not. Formally, a computable field $\mathcal{F}$ has a splitting algorithm iff the set of irreducible polynomials in $\mathcal{F}[X]$ is a computable set. (Again, we use β as a canonical translation between $\mathbb{N}^*$, i.e. the set of polynomials, and $\mathbb{N}$.)

Kronecker gave a splitting algorithm for $\mathbb{Q}$ itself. In fact, he showed that every finite extension of $\mathbb{Q}$ has a splitting algorithm, using the following.

Theorem 3.3 (Kronecker [5]; see also [1] or [16]) *If a computable field $\mathcal{L}$ has a splitting algorithm, then so does $\mathcal{L}(X)$ for any X transcendental over $\mathcal{L}$. When x is algebraic over $\mathcal{L}$, again $\mathcal{L}(x)$ has a splitting algorithm, but it requires knowledge of the minimal polynomial of x over $\mathcal{L}$.*

The algorithms for algebraic and transcendental elements are different, so for an arbitrary extension $\mathcal{L}(t)$, one needs to know whether t is algebraic or not, and if it is, one also needs to know the minimal polynomial of t over $\mathcal{L}$. It is possible to exploit this need directly to produce computable fields without splitting algorithms, but in fact we have an example already. If the computable field $\mathcal{E}_K$ had a splitting algorithm, then for any input n, we could find the element p_n in $\mathcal{E}_K$ and ask whether the polynomial $(X^2 - p_n)$ splits in $\mathcal{E}_K[X]$ or not. The answer would tell us whether $n \in K$, by definition of $\mathcal{E}_K$, and so K would be computable, contrary to Fact 2.3. So we have shown:

Lemma 3.4 *There exists a computable field without a splitting algorithm.* ■

4 Algorithms and Galois Theory

The famous results of classical Galois theory are concerned not with general algorithms for finding roots of polynomials, but rather with specific formulas for those roots, or the absence of any such formulas. The most famous result, the Ruffini-Abel Theorem, states that there is no formula using radicals for a root of a general fifth-degree polynomial. It does not consider other possible algorithms for computing such a root. On the other hand, radicals are often taken for granted in these formulas, with the underlying assumption (from an algorithmic point of view) that radicals are somehow known: that for any element x of the field, one can effectively find an n-th root of x, for any n. Of course, a field need not even contain n-th roots of all its elements, and in the computable field $\mathcal{E}_K$ already constructed above, the set of elements possessing square roots in the field is not a computable set. So, in considering Galois-style questions for computable fields, we may wind up with different answers than the classical Galois-theoretic results.

Of course, in computability theory, our algorithms often involve simple (perhaps even simple-minded) search procedures. An analogous situation arose in Hilbert's Tenth Problem, which demands an algorithm for determining whether an arbitrary Diophantine equation (i.e. a polynomial in $\mathbb{Z}[X_1, \ldots, X_n]$, for any n) possesses a solution in $\mathbb{Z}^n$. Intuitively, the real question is to find such a solution, not merely to prove that one exists. However, a simple search procedure suffices: just check, for each $m \geq 0$ in order, whether any $\vec{a} \in \mathbb{Z}^n$ with $\sum_i |a_i| = m$ solves the equation. This algorithm certainly produces a solution, assuming only that one exists; the difficulty is that if no solution exists, the algorithm will simply run forever, without ever giving an answer. So the question of effectively finding a solution reduces to Hilbert's question of how to decide whether such a solution exists. Matijasevič proved in [7], building on work of Davis, Putnam, and Robinson, that no algorithm exists which will compute the set of Diophantine equations possessing solutions, by showing that such an algorithm would allow us to compute K (and all other c.e. sets).

Dealing with polynomials $p(X)$ with coefficients in a computable field $\mathcal{F}$ creates a similar situation. We can search for a root of p in $\mathcal{F}$, just by computing $p(a_0), p(a_1), \ldots$ until we find a root. Again, the real question is determining whether this search will ever halt, i.e. whether $\mathcal{F}$ contains a root of $p(X)$. By definition, $\mathcal{F}$ has a *root algorithm* if the set $\{p(X) \in \mathcal{F}[X] : (\exists a \in \mathcal{F})\ p(a) = 0\}$ is computable.

A splitting algorithm for $\mathcal{F}$ would solve this problem, of course, by giving us a factorization of $p(X)$ into components irreducible in $\mathcal{F}[X]$: p would have as many roots as it has linear components. So a splitting algorithm yields a root algorithm. We will consider the converse in Section 5.

Thinking of Galois theory, we could also define a *radical algorithm*, for deciding whether polynomials of the form $(X^n - a)$ have roots in $\mathcal{F}$, for arbitrary n and a. In fact, we will also break down this question by degree, defining the following sets:

$$P_n(\mathcal{F}) = \{p(X) \in \mathcal{F}[X] : \deg(p) = n \ \&\ (\exists a \in \mathcal{F})\ p(a) = 0\}$$

$$R_n(\mathcal{F}) = \{x \in \mathcal{F} : (\exists y \in \mathcal{F})\ y^n = x\}.$$

For $n > 1$, none of these sets need be computable. The field $\mathcal{E}_K$ showed this for R_2, hence also for P_2, and similar constructions hold for larger n. On the other hand, the quadratic formula proves that if $R_2(\mathcal{F})$ is computable, then so is $P_2(\mathcal{F})$, since one need only decide whether the discriminant of a quadratic polynomial lies in $R_2(\mathcal{F})$. Since the converse is immediate, P_2 and R_2 may be said to be *equicomputable.*

(A more precise term than equicomputable is *Turing-equivalent*, and indeed in this case *computably isomorphic.* These indicate that, under various definitions, $P_2(\mathcal{F})$ and $R_2(\mathcal{F})$ are "equally hard" to compute. Turing-equivalence, for example, means that if one had an oracle that would answer questions of the form "Is $p(X)$ in $R_2(\mathcal{F})$?" for arbitrary $p(X)$, one could write a program which would ask such questions of the oracle and use the answers to decide membership of arbitrary polynomials in $P_2(\mathcal{F})$; and vice versa with an oracle for $P_2(\mathcal{F})$. Thus, even in the case when these are both noncomputable, they are at the same level in the hierarchy of noncomputability known as the Turing degrees. In this article, for the sake of simplicity, we have studiously avoided the notions of Turing degree and oracle computability; they may be found in any standard text on the subject.)

$P_3(\mathcal{F})$ and $R_3(\mathcal{F})$ are not in general equicomputable, however. The cubic formula uses not only cube roots, but also square roots, and so $P_3(\mathcal{F})$ is computable if both $R_2(\mathcal{F})$ and $R_3(\mathcal{F})$ are. The converse fails: indeed, our field $\mathcal{E}_K$ serves yet again as the counterexample. Given $p(X)$ of degree 3, we can enumerate elements $n_1, n_2, \ldots$ of K until we find a subfield $(\mathcal{E}_K)_j = \mathbb{Q}[\sqrt{p_{n_1}}, \ldots, \sqrt{p_{n_j}}]$ which contains all coefficients of $p(X)$. Now we know a splitting algorithm for $(\mathcal{E}_K)_j$, no matter how large j may be, and so we may check whether $p(X)$ is reducible in $(\mathcal{E}_K)_j[X]$. If so, then $p(X)$ has a root in $(\mathcal{E}_K)_j$, since one factor must be linear. If not, then it cannot have any root anywhere in $\mathcal{E}_K$, since any new root r appearing in some further finite extension of $\mathbb{Q}$ would have minimal polynomial $p(X)$ over $(\mathcal{E}_K)_j$, which is impossible, because $p(X)$ has degree 3 and all subsequent extensions of $(\mathcal{E}_K)_j$ are extensions of degree 2^k for some k. Thus $P_3(\mathcal{E}_K)$ is computable, and hence so is $R_3(\mathcal{E}_K)$, yet $R_2(\mathcal{E}_K)$ is not.

The formula for roots of fourth degree polynomials can be expressed using only square roots and cube roots, of course, and so $P_4(\mathcal{F})$ is computable whenever $R_2(\mathcal{F})$ and $R_3(\mathcal{F})$ both are. We encourage the reader to consider possible converses, perhaps involving computability of $P_3(\mathcal{F})$ as well (and $P_2(\mathcal{F})$, except that this is equicomputable with $R_2(\mathcal{F})$, of course). The complete omission of $R_4(\mathcal{F})$ from this discussion is justified by the following general lemma and its corollary.

Lemma 4.1 *Fix any computable field $\mathcal{F}$ and any $n, k > 0$ in $\mathbb{N}$. Then $R_{nk}(\mathcal{F})$ is computable iff both $R_n(\mathcal{F})$ and $R_k(\mathcal{F})$ are.*

Proof. For the forwards direction, let m be the number of (nk)-th roots of unity in a computable field $\mathcal{F}$ with $R_{nk}(\mathcal{F})$ computable, and let x be an arbitrary element of $\mathcal{F}$. We check whether $x^k \in R_{nk}(\mathcal{F})$. If not, then certainly $x \notin R_n(\mathcal{F})$. If so, then either $x = 0$ (so $x \in R_n(\mathcal{F})$), or $\mathcal{F}$ must contain exactly m elements $y_1, \ldots, y_m$ with $y_i^{nk} = x^k$, because the quotients $\frac{y_i}{y_1}$ are precisely the (nk)-th roots of unity. We check whether $y_i^n = x$ for any $i \leq m$. If so, then of course $x \in R_n(\mathcal{F})$. If not, then $x \notin R_n(\mathcal{F})$, because any y with $y^n = x$ would have $y^{nk} = x^k$, forcing $y = y_i$ for some $i \leq m$. Thus $R_n(\mathcal{F})$ is computable, and likewise for $R_k(\mathcal{F})$.

The converse is similar, once we know the number of n-th roots of unity in $\mathcal{F}$: check whether a nonzero x has any n-th roots in $\mathcal{F}$, and if so, find them all and check whether any of them has a k-th root. ■

Corollary 4.2 *For any computable field $\mathcal{F}$ and any $n > 0$, $R_n(\mathcal{F})$ is computable iff, for all primes p dividing n, $R_p(\mathcal{F})$ is computable.* ■

As a technical aside, the proof of Lemma 4.1 was *nonuniform*; it required specific knowledge about $\mathcal{F}$ and about (nk). The forwards direction, for instance, only claims that for every $\mathcal{F}$ with $R_{nk}(\mathcal{F})$ computable, there exists an algorithm for computing $R_n(\mathcal{F})$, and this is true. In order to know which algorithm it is, however, one needs to know the number of (nk)-th roots of unity in $\mathcal{F}$, and in general this number is not computable: for $nk > 2$, there is no algorithm which takes as input the programs for addition and multiplication in a computable field and outputs the number of (nk)-th roots of unity in that field. So the proof was not uniform in $\mathcal{F}$. Nor was it uniform in (nk): the reader may already be able to construct a single computable field $\mathcal{E}$ for which every individual $R_p(\mathcal{E})$ is computable, but the set $\{\langle x, n\rangle : x \in R_n(\mathcal{E})\}$ is not.

5 Rabin's Theorem

To go further, we will want to consider algebraically closed fields (or ACF's). The definitive result on computable algebraic closures of computable fields was given by Michael Rabin. For a full proof, see [11]; some discussion also appears in [9]. We give his name to the type of embedding we wish to consider.

Definition 5.1 Let $\mathcal{F}$ and $\mathcal{E}$ be computable fields. A function $g : \mathcal{F} \to \mathcal{E}$ is a *Rabin embedding* if:

- g is a homomorphism of fields; and
- $\mathcal{E}$ is both algebraically closed and algebraic over the image of g; and
- g is a computable function. (More precisely, there is a total computable h with $g(a_n) = b_{h(n)}$ for all n, where $\mathcal{F}$ has domain $\{a_0, a_1, \ldots\}$ and $\mathcal{E}$ has domain $\{b_0, b_1, \ldots\}$.)

Thus $\mathcal{E}$ is the algebraic closure of $\mathcal{F}$ in a strong way: we actually have $\mathcal{F}$ as a subfield of $\mathcal{E}$, using the computable isomorphism g, and that subfield is computably enumerable, since (the indices of) its elements form the range of a total computable function. It is not hard to show that all countable algebraically closed fields are computably presentable, but when $\mathcal{F}$ has infinite transcendence degree (and we cannot compute a transcendence basis for $\mathcal{F}$; see [8] or [9]), it is not obvious that a Rabin embedding of $\mathcal{F}$ exists. Moreover, we would like the image of our embedding to be a computable subfield of $\mathcal{F}$, not just a c.e. subfield. Rabin resolved these difficulties with his celebrated theorem. Part 1 is the heart of the theorem, but Part 2 is the more pleasing result and is more often cited. The proof of Part 2 is readily understandable and readable at this level; a sketch appears in [9].

Theorem 5.2 (Rabin [11]) *Let $\mathcal{F}$ be any computable field.*

1. *There exists a computable ACF $\overline{\mathcal{F}}$ with a Rabin embedding of $\mathcal{F}$ into $\overline{\mathcal{F}}$.*
2. *For every Rabin embedding g of $\mathcal{F}$ (into any computable ACF $\mathcal{E}$), the image of g is a computable subset of $\mathcal{E}$ iff $\mathcal{F}$ has a splitting algorithm.*

So Part 1 implies that $\mathcal{F}$ can always be taken to be a c.e. subfield within some computable algebraic closure of itself. For algebraic number fields, we may fix this closure, but not for fields in general.

Corollary 5.3 *1. The computably presentable algebraic field extensions of $\mathbb{Q}$ are precisely the c.e. subfields of any single computable presentation of $\overline{\mathbb{Q}}$, even up to computable isomorphism.*

2. *The computably presentable field extensions of $\mathbb{Q}$ of transcendence degree $\leq n$ are precisely the c.e. subfields of any single computable presentation of $\overline{\mathbb{Q}(X_1, \ldots, X_n)}$, again up to computable isomorphism.*

3. *The computably presentable fields of characteristic 0 are precisely the c.e. subfields of all computable presentations of $\overline{\mathbb{Q}(X_1, X_2, \ldots)}$, up to computable isomorphism.*

The algebraic closure of any single field $\mathbb{Q}(X_1, \ldots, X_n)$, for any $n \geq 0$, is said to be *computably categorical*: every pair of computable presentations of this algebraic closure have a computable isomorphism between them. This is why, in (1) and (2), a single copy of the field suffices. On the other hand, the field in (3) is not computably categorical. In fact, since the purely transcendental extension $\mathbb{Q}(X_1, X_2, \ldots)$ has a computable presentation with a computable transcendence basis and a splitting algorithm, Rabin's Theorem implies the existence of a computable presentation $\mathcal{C}$ of the algebraic closure of this field with its own computable transcendence basis. One checks that then every c.e. subfield of $\mathcal{C}$ must then also have a computable transcendence basis. However, in [8] Metakides and Nerode constructed a computable field $\mathcal{F}$ with no computable transcendence basis, and so this $\mathcal{F}$ has no Rabin embedding g into $\mathcal{C}$, since the preimage of a computable transcendence basis for $g(\mathcal{F})$ would be a computable transcendence basis for $\mathcal{F}$. Of course, by Theorem 5.2, $\mathcal{F}$ does have a Rabin embedding into a different computable ACF isomorphic to $\mathcal{C}$. This suggests why the statement in (3) of Corollary 5.3 is not as strong as (1) and (2).

For the converse of each part of the corollary, notice that any c.e. subfield of any computable field with domain $\{a_0, a_1, \ldots\}$ can be pulled back to a domain $\{b_0, b_1, \ldots\}$ using an enumeration of the subfield, with the operations lifted from the subfield to its pullback. Since the pullback is computable, the lifted operations are also computable.

With Rabin's Theorem we may also answer a question posed above. This corollary nicely illustrates the usefulness of the theorem, since it is not at all easy to prove the answer directly. (A direct proof using symmetric polynomials appears in [3].)

Corollary 5.4 *A computable field $\mathcal{F}$ has a splitting algorithm iff it has a root algorithm.*

Proof. We discussed the forward direction earlier, but the converse should surprise the attentive reader: just because we know that a given polynomial in $\mathcal{F}[X]$, say of degree 26, has no roots in $\mathcal{F}$, how can we know whether it factors there? To see how, fix a Rabin embedding g of $\mathcal{F}$ into some computable algebraically closed $\overline{\mathcal{F}}$. Now for any $x \in \overline{\mathcal{F}}$, we may find some $p(X) \in \mathcal{F}[X]$, with image $\overline{p}(X) \in (g(\mathcal{F}))[X]$ under g, for which $\overline{p}(x) = 0$. We determine the roots of $p(X)$ in $\mathcal{F}$ by recursion, searching for a root r_{n+1} of $\frac{p(X)}{(X-r_1)\cdots(X-r_n)}$ in $\mathcal{F}$, starting with $n = 0$, until the root algorithm for $\mathcal{F}$ says that $\frac{p(X)}{(X-r_1)\cdots(X-r_n)}$ has no roots in $\mathcal{F}$. Then $x \in g(\mathcal{F})$ iff $x = g(r_i)$ for some $i \leq n$. Thus $g(\mathcal{F})$ is computable, and Rabin's Theorem yields a splitting algorithm for $\mathcal{F}$. ■

We will also need the following theorem, from chapter 17 of the excellent reference [2] by Fried and Jarden. For deeper investigations into the Galois theory of computable fields, this result is essential.

Theorem 5.5 *Galois groups of Galois extensions of computable subfields of $\overline{\mathbb{Q}}$ are computable, uniformly in a splitting algorithm for the subfield and in any finite generating set of the extension within $\overline{\mathbb{Q}}$.*

Combined with Rabin's Theorem and classical Galois theory, this theorem yields a nice result for radical closures, a topic we will use in the next section.

Definition 5.6 For any subfield $\mathcal{E}$ of an algebraically closed field $\mathcal{F}$, the *radical closure* of $\mathcal{E}$ is the smallest subfield of $\mathcal{F}$ which contains $\mathcal{E}$ and, for every $n > 1$, contains n distinct n-th roots of each of its nonzero elements. $\mathcal{E}$ is *radically closed* if $\mathcal{E}$ is its own radical closure.

Of course, classical Galois theory shows that the radical closure can be a proper subfield of the algebraic closure, and specifically that certain polynomials of degree 5 fail to have roots in the radical closure.

Corollary 5.7 *Fix any computable field $\mathcal{F}$ with a splitting algorithm. Then the radical closure of $\mathcal{F}$ also has a computable presentation with a splitting algorithm, within which $\mathcal{F}$ is a computable subfield.*

Proof. Rabin's Theorem yields a Rabin embedding g of $\mathcal{F}$ into a computable algebraically closed field $\overline{\mathcal{F}}$, and shows that $g(\mathcal{F})$ is computable. For any $x \in \overline{\mathcal{F}}$, we can use the splitting algorithm for $\mathcal{F}$ to find an irreducible polynomial

$p(X) \in \mathcal{F}[X]$ whose image under g has x as a root. The splitting field $\mathcal{F}_p \subset \overline{\mathcal{F}}$ of p over $g(\mathcal{F})$ is a finite extension of $g(\mathcal{F})$, hence also computable, and once we have found all roots of p in $\overline{\mathcal{F}}$, Theorem 5.5 then allows us to compute the (finite) Galois group G for $\mathcal{F}_p$ over $g(\mathcal{F})$, viewed as a permutation group on the roots of p. But now it is a well-known result of Galois theory that x lies in the radical closure of $g(\mathcal{F})$ iff G is solvable, and we may determine solvability of G simply by repeatedly computing commutator subgroups $G^{(m)}$ until either $G^{(m+1)} = G^{(m)}$ or $G^{(m)}$ is trivial.

This constitutes an algorithm for determining membership of an arbitrary $x \in \overline{\mathcal{F}}$ in the radical closure of $g(\mathcal{F})$ within $\overline{\mathcal{F}}$. By Corollary 5.3, the radical closure is itself computably presentable, via a computable isomorphism (under which the preimage of the computable subfield $g(\mathcal{F})$ is also computable), and has a splitting algorithm, by Rabin's Theorem. ■

6 Effective Unsolvability of the Quintic

Finally we show that the famous Galois-theoretic result of unsolvability of the quintic equation by radicals also holds when "solvability" is taken to refer to algorithms for computable fields, as discussed in Section 4. Indeed, the field $\mathcal{E}$ we construct will be radically closed, with $R_n(\mathcal{E}) = \mathcal{E}$ for every n. So, as in the classical result, the unsolvability remains even when we are given the ability to find every radical we could want.

Theorem 6.1 *There exists a computable field $\mathcal{E}$ with $R_n(\mathcal{E}) = \mathcal{E}$ for every n, such that $P_5(\mathcal{E})$ is not computable.*

Proof. We start by fixing a single polynomial $p(X, Y)$ of degree 5 in Y, with rational coefficients, such that when p is viewed as a polynomial $p_X(Y)$ over the field $\mathbb{Q}(X)$, no root of p_X lies in the radical closure of $\mathbb{Q}(X)$, nor in the algebraic closure of $\mathbb{Q}$. An example is $p(X, Y) = XY^5 + Y^5 - Y - 1$, which can be shown using methods from Section 8.10 of [16] to have the symmetric group S_5 as its Galois group over $\mathbb{Q}(X)$.

(Details: we apply the result on p. 198 of [16] with $\mathfrak{R} = \mathbb{Q}[X]$ and $\mathfrak{p} = (X)$ to see that the Galois group of $p_X(Y)$ over $\mathbb{Q}(X)$ contains the Galois group of $(Y^5 - Y - 1)$ over $\mathbb{Q}$, which is shown on the following page in [16] to be S_5. Since S_5 is not solvable, no root of p_X lies in the radical closure of $\mathbb{Q}(X)$. Moreover, if $p_X(r) = 0$, then $(r^5 X) + r^5 - r - 1 = 0$, and since X is transcendental over $\mathbb{Q}$, r cannot lie in $\overline{\mathbb{Q}}$.)

Now consider a computable field $\mathbb{Q}(X_0, X_1, \ldots)$, with computable transcendence basis $\{X_0, X_1, \ldots\}$. By Theorems 3.3 and 5.2, its algebraic closure has a computable presentation, $\mathcal{C}$, for which there exists a Rabin embedding g of $\mathbb{Q}(X_0, \ldots)$ into $\mathcal{C}$ with computable image. We define the polynomials $p_e(Y) = p(g(X_e), Y) \in \mathcal{C}[Y]$, for every e.

We build a computably enumerable subfield $\mathcal{F} \subseteq \mathcal{C}$, by enumerating a generating set W and closing under the field operations (including negation and reciprocals) and also under the operation of taking n-th roots, for every $n > 0$. To begin with, W contains all the elements X_i of the (computable) transcendence basis given above. (More precisely, W contains their images in $\mathcal{C}$.) Then we computably enumerate the set K from Fact 2.3. For each new number e which appears in our enumeration of K, we search through $\mathcal{C}$ for the five roots of p_e and enumerate all those roots into W. Having done so, of course, we continue closing $\mathcal{F}$ under all the operations, including taking radicals. The subfield $\mathcal{F}$ is the set of all elements of $\mathcal{C}$ which either enter W at some stage, or are included in our closure process, so we have given a computable enumeration of $\mathcal{F}$.

Officially $\mathcal{F}$ itself is not a computable field, but we may use Corollary 5.3 to pull $\mathcal{F}$ back to a computable field $\mathcal{E}$ with a computable isomorphism g from $\mathcal{E}$ onto $\mathcal{F}$. Since we closed $\mathcal{F}$ under radicals, we have $R_n(\mathcal{F}) = \mathcal{F}$ for all n, and thus $R_n(\mathcal{E}) = \mathcal{E}$ as well. Similarly, if the set $P_5(\mathcal{E})$ were computable, then we could also compute whether an arbitrary $q(Y) \in \mathcal{F}[Y]$ lies in $P_5(\mathcal{F})$, just by checking whether its preimage lies in $P_5(\mathcal{E})$. (To get the preimage, just search for the preimages under g of the coefficients of q.)

Now we claim that each natural number e lies in K iff the polynomial p_e lies in $P_5(\mathcal{F})$. Since from e we can easily compute p_e, the computability of $P_5(\mathcal{F})$ would therefore imply computability of the noncomputable set K. First, if $e \in K$, then we enumerated a root of p_e into W, so $p_e \in P_5(\mathcal{F})$. Next suppose $e \notin K$, and let r_e be any root of p_e in $\mathcal{C}$. Then X_e is algebraically dependent on r_e in $\mathcal{C}$, and since $e \notin K$, r_e never entered W. Moreover, our choice of $p(X, Y)$ ensured that r_e cannot lie in the radical closure of $\mathbb{Q}(X_e)$, nor in $\overline{\mathbb{Q}}$. Therefore, if r_e ever entered $\mathcal{F}$ under our closure operations, it did so due to some roots $r_{i_1}, \ldots, r_{i_m} \in W$ from some polynomials p_{i_k}, with all $i_k \neq e$. But then r_e would sit in the algebraic closure of $\mathbb{Q}(X_{i_1}, \ldots, X_{i_m})$, contradicting the algebraic independence of X_e from the set $\{X_{i_1}, \ldots, X_{i_m}\}$. Thus no roots of p_e lie in $\mathcal{F}$, so $p_e \notin P_5(\mathcal{F})$. Therefore, as we claimed, $e \in K$ iff $p_e \in P_5(\mathcal{F})$, so $P_5(\mathcal{F})$ is not computable, and neither is $P_5(\mathcal{E})$. ■

We invite the reader to attempt to prove Theorem 6.1 for algebraic field extensions of $\mathbb{Q}$. Of course, it is not necessary to have $R_n(\mathcal{E}) = \mathcal{E}$ for all n, but one would like to have $R_n(\mathcal{E})$ be computable uniformly in n. That is, there should be a single algorithm which accepts a pair $\langle n, x\rangle$ as input and decides whether $x \in R_n(\mathcal{E})$. (The nonuniform version simply requires every set $R_n(\mathcal{E})$ to be computable, allowing a completely different algorithm for each n.)

7 Notes

The foregoing discussion in no way replaces classical Galois theory, of course. First of all, the results of the classical theory were necessary: they provided the polynomials whose roots all lie outside the radical closure of the field we built. Second, our results serve mainly to reinforce the classical conclusion that there is something special about the degree 5 for polynomials: the process of searching for roots runs into trouble when one reaches that degree. Finally, our discussion only applied to computable fields. These were sufficient to provide the example we wanted, in Theorem 6.1, but one would like to extend the discussion. Other countable fields can be considered if one relativizes the notion of computability to allow an oracle, and the results from preceding sections would generally carry over to that case. However, computable model theory has always restricted itself to countable structures, essentially because the nature of Turing machines and computations in finite time allows only countably many inputs to such a machine. The author is enthusiastic about his own current work on *locally computable structures*, i.e. mathematical structures, quite possibly uncountable, whose finitely generated substructures are all computably presentable in a uniform way. As work on this topic progresses, notions from this article might come to apply to many uncountable fields as well. Details are available in [10], and a brief summary appears in [9].

In this article, every time we have wanted to produce an example of noncomputability, we have used the set K. The reader should not be misled into thinking that K is the only noncomputable set available. Indeed, one can build infinitely many computably enumerable sets, no two of which are Turing-equivalent to each other (as defined in Section 4), and beyond those, there are uncountably many subsets of $\mathbb{N}$ which are not computably enumerable and which have their own degrees. Much of computable model theory considers ways in which structures, e.g. fields, representing all these different degrees of complexity may be produced. For simplicity, we have avoided such questions here.

On the other hand, the search procedures used here may often seem irritatingly slow. Does one really need to search through $\mathbb{Q}$ element by element to find the solution to a polynomial equation? When one starts to consider questions about the amount of time and memory required for a search, one has crossed into theoretical computer science, where such questions are studied intensely, and where simple, slow search procedures as in this article are deemed useless. In contrast, computability theorists wish to consider all possible algorithms, and the easiest way to do so is to strip away their complexity and reduce them all to search procedures. Having done so, one can readily produce noncomputable objects, by ensuring that no search procedure works. It has been said that the discipline should really be called *noncomputability theory*, since it puts so much effort into building and studying noncomputable objects. However, such a name would be not only unduly negative, but also inaccurate: even when studying noncomputable objects, we are usually asking which objects contain enough information to compute other objects (i.e. which objects have higher Turing degree), so the real subject is still computability.

The results in this article should be assumed to be folklore unless specific attribution is made. Some of the questions considered may not have been raised before, but by the standards of hard-core computable model theory, the answers given are not particularly complex. The author would be grateful for any information about sources which may already have considered the material in Sections 4 and 6.

Many thanks are due to Professors Phyllis Cassidy, Richard Churchill, Li Guo, William Keigher, Jerry Kovacic, and William Sit, the organizers of the the Second International Workshop on Differential Algebra and Related Topics, who invited the author to give a tutorial there on April 12, 2007. This article grew out of that presentation, with the encouragement of Prof. Andy Magid. The reference [9], by the same author, uses the same notation and terminology, and provides a fair amount of complementary information, for those wishing to see more.

References

[1] H.M. Edwards, *Galois Theory* (New York: Springer-Verlag, 1984).

[2] M.D. Fried & M. Jarden, *Field Arithmetic* (Berlin: Springer-Verlag, 1986).

[3] A. Frohlich & J.C. Shepherdson; Effective procedures in field theory, *Phil. Trans. Royal Soc. London, Series A* **248** (1956) 950, 407-432.

[4] W. Hodges; *A Shorter Model Theory* (Cambridge: Cambridge University Press, 1997).

[5] L. Kronecker; Grundzüge einer arithmetischen Theorie der algebraischen Größen, *J. f. Math.* **92** (1882), 1-122.

[6] M. Lerman, *Degrees of Unsolvability: Local and Global Theory* (Berlin: Springer-Verlag, 1983).

[7] Ju.V. Matijasevič; The Diophantineness of enumerable sets (Russian), *Doklady Akademii Nauk SSSR* **191** (1970), 279-282; (English translation) *Sov. Math. Dokl.* **11** (1970), 354-357.

[8] G. Metakides & A. Nerode; Effective content of field theory, *Annals of Mathematical Logic* **17** (1979), 289-320.

[9] R.G. Miller; Computability and differential fields: a tutorial, in *Differential Algebra and Related Topics: Proceedings of the Second International Workshop*, eds. L. Guo & W. Sit, to appear. Also available at `qcpages.qc.cuny.edu/~rmiller/research.html`.

[10] R.G. Miller, Locally computable structures, in *Computation and Logic in the Real World - Third Conference on Computability in Europe, CiE 2007*, eds. B. Cooper, B. Löwe, & A. Sorbi, *Lecture Notes in Computer Science* **4497** (Springer-Verlag: Berlin, 2007), 575-584. Also available at `qcpages.qc.cuny.edu/~rmiller/research.html`.

[11] M. Rabin; Computable algebra, general theory, and theory of computable fields, *Transactions of the American Mathematical Society* **95** (1960), 341-360.

[12] H. Rogers, Jr.; *Theory of Recursive Functions and Effective Computability* (New York: McGraw-Hill Book Co., 1967).

[13] R.I. Soare; *Recursively Enumerable Sets and Degrees* (New York: Springer-Verlag, 1987).

[14] V. Stoltenberg-Hansen & J.V. Tucker, Computable Rings and Fields, in *Handbook of Computability Theory*, ed. E.R. Griffor (Amsterdam: Elsevier, 1999), 363-447.

[15] A. Turing; On computable numbers, with an application to the Entscheidungsproblem, *Proceedings of the London Mathematical Society* (1936), 230-265.

[16] B.L. van der Waerden; *Algebra*, volume I, trans. F. Blum & J.R. Schulenberger (New York: Springer-Verlag, 1970 hardcover, 2003 softcover).

Department of Mathematics
Queens College – C.U.N.Y.
65-30 Kissena Blvd.
Flushing, New York 11367 U.S.A.
Ph.D. Programs in Mathematics & Computer Science
The Graduate Center of C.U.N.Y.
365 Fifth Avenue
New York, New York 10016 U.S.A.

E-mail: Russell.Miller@qc.cuny.edu
Website: qcpages.qc.cuny.edu/~rmiller