A Complexity Lower Bound Based On Software Engineering Concepts

Andrés Rojas Paredes

Universidad de Buenos Aires,
Facultad de Ciencias Exactas y Naturales, Departamento de Computación.
Pabellón 1, Ciudad Universitaria, Buenos Aires, Argentina.
arojas@dc.uba.ar

Abstract. We consider the problem of polynomial equation solving also known as quantifier elimination in Effective Algebraic Geometry. The complexity of the first elimination algorithms were double exponential, but a considerable progress was carried out when the polynomials were represented by arithmetic circuits evaluating them. This representation improves the complexity to pseudo–polynomial time.

The question is whether the actual asymptotic complexity of circuit–based elimination algorithms may be improved. The answer is no when elimination algorithms are constructed according to well known software engineering rules, namely applying information hiding and taking into account non–functional requirements. These assumptions allows to prove a complexity lower bound which constitutes a mathematically certified non–functional requirement trade–off and a surprising connection between Software Engineering and the theoretical fields of Algebraic Geometry and Computational Complexity Theory.

Keywords: Non-functional requirement trade–off, information hiding, arithmetic circuit, complexity lower bound, polynomial equation solving, quantifier elimination in algebraic geometry

1 Introduction

The main issue of this paper is to describe the Software Engineering aspects of the mathematical computation model introduced in [9]. This model captures the notion of a circuit–based elimination algorithm in order to solve a thirty years old problem in algebraic complexity theory (see e.g. [8], [10]): in *arithmetic circuit–based* effective elimination theory the elimination of a single existential quantifier block in the first order theory of algebraically closed fields of characteristic zero is *intrinsically hard* (i.e. it has an *exponential* complexity lower bound). This conclusion may also be expressed in terms of a trade–off between two non–functional requirements: on one hand we have a complexity requirement and on the other a property of mathematical functions called *geometrical robustness*. This complexity lower bound in terms of software engineering concepts appears

for first time in [5] in the context of polynomial interpolation. In this work we study a more general case in the context of quantifier elimination.

Complexity lower bounds are undoubtedly theoretical research. But there is also a practical aim behind that. Consider the process in software design where a software architecture is developed in order to solve a certain computational problem. Assume also that one of the non–functional requirements of the software design project consists of a restriction on the run time computational complexity of the program which is going to be developed (this was the case during the implementation of the polynomial equation solver Kronecker by G. Lecerf, see [11]). Our practical aim is to provide the software engineer with an efficient tool which allows him to answer the question whether his software design process is entering at some moment in conflict with the given complexity requirement. If this is the case, the software engineer will be able to change at this early stage his design and may look for an alternative software architecture. The following example illustrates this description.

Example 1 (Finite Set). Suppose that our task is to implement a finite set S of cardinality n, e.g. a subset of the natural numbers $\mathbb{N}$, and that we have to satisfy the requirement that membership to the finite set S is decided using only $O(\log n)$ comparisons. If the set S is implemented by an unordered array, we will be unable to satisfy our complexity requirement. So we are forced to think in alternative implementations of the abstract concept of a finite set, e.g. by ordered arrays, special trees or any other data type which is well suited for our task.

Example 1 represents a case where it may be impossible to satisfy a given complexity requirement by means of a previously fixed software architecture. Our aim is to formalise such impossibility by means of a complexity lower bound which is usually difficult to infer when the number of components of the system under consideration is large or when the predicate to decide or the function to compute becomes more sophisticated like in polynomial equation solving. This leads to the idea to fix in advance only a small selection of architectural features, e.g. the abstraction levels or part of the language of our system (not the algorithms themselves). The computation model we are going to explain in following sections takes into account these considerations.

This work is organised as follows: in Section 2 we introduce quantifier elimination as the subject of our complexity studies and the algorithmic approach which is based on the transformation of arithmetic circuits. In Section 3 we describe the tool used to obtain the announced complexity lower bound. Our tool is a computation model which captures the notion of non–functional requirement in circuit–based elimination algorithms. Finally we present the new result in this work: we make the following question: What does it happen if our algorithms are not circuit–based and we found a representation which is more efficient than circuits? The answer is that our complexity results are valid for arbitrary continuous representations if the algorithms follow the principle of *information hiding*. We illustrate this conclusion with a relevant example from the theory of Abstract Data Types (see, e.g. [13] and [12]).

In the rest of the paper we shall use notions and notations from algebraic geometry and algebraic complexity theory which are all standard (see for example [14] and [3]).

2 Quantifier elimination and its implementation

2.1 Quantifier Elimination

We start with the subject of our complexity studies. The subject is *quantifier elimination in the particular case of elementary algebraic geometry over* $\mathbb{C}$. Let Φ be an existentially quantified formula. In general terms, the quantifier elimination problem consists in obtaining a quantifier free formula Ψ which is logically equivalent to Φ (this means that Ψ and Φ define the same set). In the particular case of elementary algebraic geometry over $\mathbb{C}$, the formulas Φ and Ψ are composed by polynomial equations. In this context we are going to consider exclusively the polynomials of these equations.

Let n and r be natural numbers. Let T, $U := (U_1, \ldots, U_r)$ be *parameters* and $X := (X_1, \ldots, X_n)$ be *variables* subject to quantification. We focus our attention to polynomials $G_1(X), \ldots, G_n(X)$ and $H(T, U, X)$ which belong to $\mathbb{C}[X]$ and to $\mathbb{C}[T, U, X]$ respectively. These polynomials constitute a so called *Flat Family of Elimination Problems* given by the polynomial equation system $G_1 = 0, \ldots, G_n = 0$ and the polynomial H (see, e.g. [4] and [9] for details). In general terms this system represents the quantified formula $\Phi : (\exists X_1)(\exists X_n)(G_1 = 0 \wedge \ldots \wedge G_r = 0 \wedge Y - H = 0)$.

On the other hand, there exists a polynomial $F \in \mathbb{C}[T, U, Y]$ of minimal degree, called the associated *Elimination Polynomial*, such that the equation $F = 0$ represents a quantifier–free formula Ψ which is equivalent to Φ.

Thus, we arrive to a functional requirement where the flat family of elimination problems given by $G_1 = 0, \ldots, G_n = 0$ and H becomes transformed into the elimination polynomial F. This transformation is carried out by a mathematical function f as Fig. 1 illustrates.

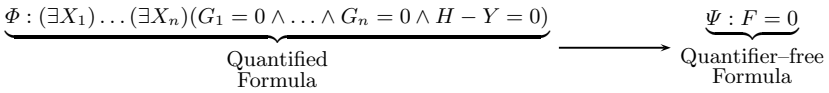


Fig. 1: Quantifier elimination problem.

At this abstract level we do not know, for example, how the polynomials are implemented in the computer. We define now these implementation details.

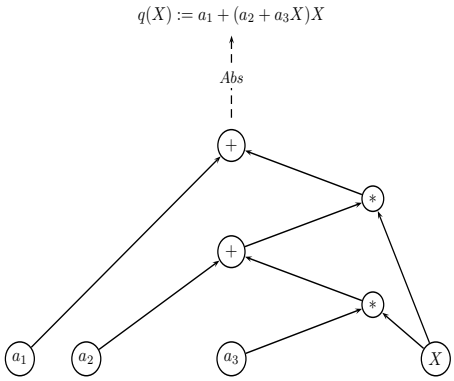
An implementation option is to represent polynomials by their coefficients. Unfortunately the coefficient representation in some elimination polynomials

may conduce to complexity blow ups, e.g. the Pochhammer polynomial

$$\prod_{0 \leq j < 2^n} (Y - j)$$

which has 2^n terms (see [7] for open questions in complexity theory related to this polynomial). This circumstance suggests to represent in elimination algorithms polynomials not by their coefficients but by arithmetic circuits. This idea became fully realised by the "Kronecker" algorithm for the resolution of polynomial equation systems over algebraically closed fields. The algorithm was anticipated in [6] and implemented in a software package of identical name (see [11]). The following example illustrates the notion of arithmetic circuit.

Fig. 2: Arithmetic circuit and Horner scheme.



Example 2 (Horner scheme). Let a_1, a_2, a_3 be constants and X be a variable. Consider the polynomial $p(X) = a_1 + a_2X + a_3X^2$ and the Horner scheme of this polynomial which is $q(X) = a_1 + (a_2 + a_3X)X$. From this scheme we have a directed acyclic graph where each node is an arithmetic operation $+, *$, a constant a_1, a_2, a_3 or a variable X. This arithmetic circuit is a concrete object implementing the abstract object $q(X)$. Fig. 2 illustrates the relation between $q(X)$ and its implementing circuit by means of an abstraction function *Abs*.

2.2 Implementation of quantifier elimination

To understand the role of arithmetic circuits in elimination algorithms we fix the notion of polynomials in terms of abstract data types and classes implementing them. Here we follow the terminology in [13].

Suppose that we have an abstract data type specification for polynomials in terms of query and creator functions (observers and constructors in the terminology of [12]). Thus the elimination problem of Fig. 1 may be expressed as a specification in terms of abstract data types.

Consider now the classes implementing the abstract data type of polynomials. We have a class for polynomials and a class for circuits. The connection between these two classes is that the class of circuits is a private part of the class of polynomials. This private part is used to implement the interface of the class of polynomials in terms circuits. In this context polynomials are encapsulated circuits which are mapped into instances of the abstract data type of polynomials by an abstraction function *Abs*.

Now recall our functional requirement: transform en elimination problem given by polynomials $G_1, \ldots, G_n$ and H into an elimination polynomial F. Since

polynomials become implemented by circuits, an elimination algorithm works directly with circuits taking care of satisfy class invariants and the abstraction function *Abs*. In this sense, an elimination algorithm $\mathcal{A}$ transforms an input circuit β representing $G_1, \ldots, G_n, H$ into an output circuit γ representing the elimination polynomial F as Fig. 3 illustrates.

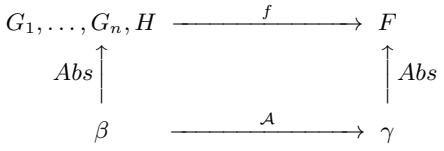


Fig. 3: Elimination problem and its implementation.

The transformation of β into γ is carried out by means of circuit operations, e.g. join of circuits which mimics the composition of functions (see also union of circuits and recursive routine in [9]). If we require the algorithm $\mathcal{A}$ to be *branching parsimonious* (see Section 3.1 below), then $\mathcal{A}$ captures all known circuit–based elimination algorithms including the polynomial equation solver Kronecker.

At this point the question is how we measure the complexity of algorithm $\mathcal{A}$. We shall mainly be concerned with the *size* of the output circuit γ. Here we refer with "size" to the number of internal nodes which count for the given complexity measure. Our basic complexity measure is the *non–scalar* one (also called *Ostrowski measure*) over the ground field $\mathbb{C}$. This means that we count, at unit costs, only essential multiplications and divisions (see [3] for details).

3 Software engineering–based approaches to complexity lower bounds

3.1 A circuit–based computation model

The polynomials $G_1, \ldots, G_n, H$ and F described before belong to mathematical structures $\mathbb{C}[X]$, $\mathbb{C}[T, U, X]$ and $\mathbb{C}[T, U, Y]$ respectively. In these mathematical structures polynomials have a natural property called *geometrical robustness* which interpreted as a non–functional requirement constitutes a key ingredient in our complexity result (see Theorem 1 below). This property is invisible if we only consider abstract data type specifications in the sense of [13]. Thus, in order to include geometrical robustness in the specification of elimination problems we model the notion of abstract data type of polynomials with the corresponding mathematical structure and we call this structure an abstract data type. For example, the polynomials $G_1, \ldots, G_n, H$ will be instances of the abstract data type $\mathcal{O} \subset \mathbb{C}[T, U, X]$ and F will be an instance of the abstract

data type $\mathcal{O}^* \subset \mathbb{C}[T, U, Y]$ and the elimination problem will be specified by a geometrically robust map $f : \mathcal{O} \to \mathcal{O}^*$.

Geometrical robustness The map f is a function (mathematical application) which we require to be constructible, i.e. definable by a boolean combination of polynomial equations. The mapping is called geometrically robust if it is continuous (see [9] and [5] for an algebraic characterisation of robustness). Since geometrical robustness is a property belonging to the specification level of our elimination task, we have to describe how this non–functional property is realised by the circuit–based algorithms implementing the elimination.

Branching parsimoniousness The intuitive meaning of geometrical robustness is reflected by the algorithmic notion of *branching parsimoniousness*. We call an algorithm branching parsimonious if it avoids unnecessary branchings. We may restrict branchings by means of only considering division–free circuits, or circuits where divisions by zero were replaced by suitable limits and divisions may only involve parameter nodes (nodes without variables). In this sense our circuits are *essentially division–free* and will be called *robust* if all intermediate results (functions represented by each node) are geometrically robust.

The notion of branching parsimoniousness as a tactic In the context of software architecture, the satisfaction of quality attributes requires techniques which are called *tactics*. For example, a system is easily modified when it is structured, modularised and well documented. A tactic is, according to [2], a design decision that influences the control of a quality attribute response. Considering this definition we may describe branching parsimoniousness as a tactic for elimination algorithms. We require an algorithm to be branching parsimonious in order to achieve the non–functional requirement of geometrical robustness. In this sense we say that branching parsimoniousness is a tactic to achieve geometrical robustness. For example, the reader may identify branching parsimoniousness with *modularity* which is a tactic to achieve the modifiability quality attribute.

Now recall our elimination algorithm $\mathcal{A}$ in Fig. 3 which transform the circuit β (representing $G_1, \ldots, G_n, H$) into circuit γ (representing F). The elimination algorithm $\mathcal{A}$ implements the additional property of geometrical robustness if we require $\mathcal{A}$ to be banching parsimonious.

Thus in the input we have an essentially division–free, robust parameterized arithmetic circuit β of size $O(n)$ with basic parameters T, $U := U_1, \ldots, U_n$ and input $X := X_1, \ldots, X_n$ which computes polynomials $G_1, \ldots, G_n \in \mathbb{C}[X]$ and $H \in \mathbb{C}[T, U, X]$ constituting a flat family of zero–dimensional elimination problems with associated elimination polynomial $F \in \mathbb{C}[T, U, Y]$.

The branching parsimoniousness allows to affirm that each circuit operation gives as result a robust circuit. Thus we conclude that the property of geometrical robustness is transmitted from the input β to the output γ. Then $\gamma := \mathcal{A}(\beta)$ is an essentially division–free, robust parameterized arithmetic circuit with basic parameters $T, U_1, \ldots, U_n$ and input Y representing the elimination polynomial F.

These notations and assumptions, in particular the property of robustness in the output γ, allows to conclude the following theorem.

Theorem 1 ([9], Theorem 10). *The circuit γ has, as ordinary arithmetic circuit over $\mathbb{C}$, non–scalar size at least $\Omega(2^n)$.*

Theorem 1 corresponds to circuit–based algorithms, now we ask what does it happen if we found a representation which is more efficient than arithmetic circuits? We argue that Information Hiding–based algorithms have the same complexity status. This implies that our complexity results are valid for arbitrary continuous representations. This is part of future work but we give preliminary results in the following section.

3.2 Towards an Information Hiding–based computation model

Since polynomials $G_1, \ldots, G_n, H$ and F are objects belonging to suitable abstract data types, we may define the function f of Fig. 3 in terms of query and creator functions (observers and constructors) of the given abstract data type specification obtaining a transformation which does not involve circuits directly because they become encapsulated. To illustrate this kind of transformation consider the following example.

Example 3. Suppose a case where f is the identity function of binary trees. In this context let us consider the following abstract functions of the corresponding abstract data type specification: $root()$, $left()$, $right()$ and $isNil?()$ as query functions (observers) and $bin()$ and $nil()$ as creator functions (constructors). Then, we propose the following definition for f:

$$f(X) = \begin{cases} nil() & \text{if } isNil?(X) \\ bin(root(X), id(left(X)), id(right(X))) & \text{otherwise} \end{cases} \quad (1)$$

This specification of function f may be implemented in such a way that, at an abstract level the implementation is the identity function of binary trees, whereas at a concrete level the implementation is a transformation of the representation of binary trees (compare this with the transformation of circuits in elimination algorithm $\mathcal{A}$). This hidden transformation is carried out by the classes implementing the abstract data type of binary trees where the implementation of f may be called `f`. Notice that we write the implementation in `verbatim` font in order to distinguish the difference with abstract data type expressions which we write in *cursive* font.

Let `Tree<E>` be a class implementing the abstract data type of binary trees. Let `Tree1<E>` and `Tree2<E>` be subclasses of class `Tree<E>` with the following property: `Tree1<E>` implements trees as arrays (the internal representation of trees is given by arrays) and `Tree2<E>` implements trees as nested nodes. Let `root`, `left`, `right` and `isNil` be routines in the class `Tree<E>` implementing the corresponding query functions (observers) in the abstract data type specification.

Let `p1` be a variable of type `E` and `p2` y `p3` be variables of type `Tree2<E>`, then `Tree2<E>()` and `Tree2<E>(p1,p2,p3)` are constructors of the class `Tree2<E>` implementing the creator functions $nil()$ and $bin()$ respectively. Then the implementation in java code is as follows:

```
Tree<E> f(Tree<E> t){
        if(t.isNil()) return new Tree2<E>();
        else return new Tree2<E>(t.root(),
                (Tree2<E>) f(t.left()),
                (Tree2<E>) f(t.right()) );
}
```
(2)

Notice that the effective transformation of the representation is carried out when an instance of `Tree1<E>` is passed as parameter and the constructor of the other class is applied, say the constructor of `Tree2<E>`.

Equation 2 illustrates the definition of an algorithm in terms of observers and constructors. In the case of elimination problems such an algorithm has a similar structure but we do not exhibit an example here. This is left for a future work (see [1]) where the notion of information hiding is modelled in full detail. Such a model allows to conclude the following:

- if the complexity measure is given by the number of parameters instead of the size of circuits, we obtain an exponential complexity lower bound for this quantity which implies the result in Theorem 1,
- this allows to conclude that elimination algorithms programmed with information hiding, i.e. hiding the circuits or any other representation of polynomials, have the same complexity status.

Final comments The circuit–based computation model described in Section 3.1 corresponds to the tool for the software engineer we described at the introduction. Of course this model cannot be applied to any software project since it is restricted to the particular case of elimination. However, it gives the key ingredients for the definition of a computation model suitable for complexity questions where another non–functional requirement must be considered.

On the other hand, our description of an Information Hiding–based computation model in Section 3.2 constitutes an stronger result which together with Theorem 1, allows to conclude that the Kronecker algorithm is asymptotically optimal. This suggest that the Kronecker is a good option to use in applications of scientific computing where polynomial equation solving is needed.

Finally, a computation model which captures algorithms constructed in a professional way, namely applying software engineering concepts, in combination with the complexity lower bound obtained in Section 3.1 allows to conclude the following idea which we repeat from [9]: neither mathematicians nor software engineers, nor a combination of them will ever produce a practically satisfactory, *generalistic* software for elimination tasks in Algebraic Geometry. This is a job for *hackers* which may find for *particular* elimination problems *specific* efficient solutions.

Acknowledgements *The author thanks Joos Heintz for his insistent encouragement to finish this work and Pablo Barenbaum, Gastón Bengolea Monzón, Mariano Cerrutti, Carlos Lopez Pombo, Hvara Ocar and Alejandro Scherz, Universidad de Buenos Aires, for discussions about the topic of this paper and/or comments and ideas on earlier drafts.*

References

1. Bank, B., Heintz, J., Pardo, L.M., Rojas Paredes, A.: Quiz games: A new approach to information hiding based algorithms in scientific computing, manuscript Universidad de Buenos Aires (2013)
2. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison–Wesley, Boston, MA, 2. edn. (2003)
3. Bürgisser, P., Clausen, M., Shokrollahi, M.A.: Algebraic Complexity Theory. Grundlehren der mathematischen Wissenschaften, vol. 315. Springer Verlag (1997)
4. Castro, D., Giusti, M., Heintz, J., Matera, G., Pardo, L.M.: The hardness of polynomial equation solving. Foundations of Computational Mathematics 3(4), 347–420 (2003)
5. Giménez, N., Heintz, J., Matera, G., Solernó, P.: Lower complexity bounds for interpolation algorithms. Journal of Complexity 27, 151–187 (2011)
6. Giusti, M., Heintz, J., Morais, J., Morgenstern, J., Pardo, L.: Straight-line programs in geometric elimination theory. Journal of Pure and Applied Algebra 124, 101–146 (1998)
7. Heintz, J., Morgenstern, J.: On the intrinsic complexity of elimination theory. Journal of Complexity 9, 471–498 (1993)
8. Heintz, J., Sieveking, M.: Absolute primality of polynomials is decidable in random polynomial time in the number of variables. Automata, languages and programming (Akko, 1981). Lecture Notes in Computer Science 115, 16–28 (1981)
9. Heintz, J., Kuijpers, B., Rojas Paredes, A.: Software engineering and complexity in effective algebraic geometry. Journal of Complexity (2012)
10. Kaltofen, E.: Greatest common divisors of polynomials given by straight–line programs. J. Assoc. Comput. Mach. 35(1), 231–264 (1988)
11. Lecerf, G.: Kronecker: a Magma package for polynomial system solving. Web page. `http://lecerf.perso.math.cnrs.fr/software/kronecker/index.html`
12. Liskov, B., Guttag, J.: Program development in Java: Specification, and Object–Oriented Design. Addison-Wesley, 3. edn. (2001)
13. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, 2. edn. (2000)
14. Shafarevich, I.R.: Basic algebraic geometry: Varieties in projective space. Springer, Berlin Heidelberg, New York (1994)