

Fast Algorithms for Towers of Finite Fields and Isogenies

Luca de Feo

▶ **To cite this version:**

Luca de Feo. Fast Algorithms for Towers of Finite Fields and Isogenies. Mathematics [math]. Ecole Polytechnique X, 2010. English. NNT : . tel-00547034v3

HAL Id: tel-00547034
https://pastel.hal.science/tel-00547034v3

Submitted on 30 Mar 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Luca De Feo



Fast Algorithms

for Towers of Finite Fields and Isogenies



Thèse de doctorat

présentée à l'École Polytechnique,
le 13 décembre 2010.

ÉCOLE POLYTECHNIQUE
ParisTech

Fast Algorithms for Towers of Finite Fields and Isogenies

Algorithmes Rapides pour les Tours de Corps Finis et les Isogénies

Thèse

présentée pour obtenir le grade de

Docteur de l'École Polytechnique

spécialité

Informatique

par

Luca De Feo

soutenue publiquement le 13 décembre 2010
devant le jury composé de :

Jean-Marc	Couveignes	(président)
Frédéric	Chyzak	
Erich	Kaltofen	(rapporteur)
François	Morain	(directeur)
Renaud	Rioboo	
Christophe	Ritzenthaler	(rapporteur)
Éric	Schost	(co-directeur)
Franz	Winkler	

Laboratoire d'Informatique de l'X
INRIA Scaclay – Équipe-projet TANC

Original version: December 13, 2010. Revised and corrected: March 30, 2011.

This document was typeset using LaTeX, bibtex, makeindex, the class Memoir and many LaTeX packages. The source code is available for download at `http://www.lix.polytechnique.fr/~defeo`.

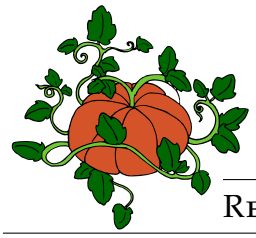
© 2010–2011 Luca De Feo. Part of this document is based on earlier work:

- *Fast arithmetics in Artin-Schreir towers*, in ISSAC '09: Proceedings of the 2009 International Symposium on Symbolic and Algebraic Computation, © ACM, 2009. `http://dx.doi.org/10.1145/1576702.1576722`
- `transalpyne`*: a language for automatic transposition*, in SIGSAM Bulletin 44, no. 1/2, © ACM, 2010. `http://dx.doi.org/10.1145/1838599.1838624`
- *Fast algorithms for computing isogenies between ordinary elliptic curves in small characteristic*, in Journal of Number Theory, in press, © Elsevier, 2010. `http://dx.doi.org/10.1016/j.jnt.2010.07.003`



Drawings by Rachel Deyts, © 2010 Rachel Deyts.

Fast Algorithms for Towers of Finite Fields and Isogenies is licensed by the copyright holders under a Creative Commons Attribution-ShareAlike 3.0 Unported License. This means that you are free to copy, adapt and distribute it, even for commercial purposes, provided that you attribute it to the authors. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one. The full license is available from `http://creativecommons.org/licenses/by-sa/3.0/legalcode`.



Remerciements

Cher lecteur, le document que vous avez entre les mains est le fruit de trois années de travail, que, au contraire de ce que l'on a tendance à imaginer d'un chercheur, je n'ai pas passées enfermé avec mes bouquins et mon ordinateur, isolé du reste du monde. Ainsi, j'aimerais abuser de votre temps de lecture pour remercier les personnes qui, d'une façon ou de l'autre, ont participé à ces années de thèse.

Tout d'abord, il y a mes mentors, François Morain et Éric Schost, à qui vous devez quasiment tous les aspects scientifiques de cette thèse : le sujet, les références et l'exactitude des théorèmes (s'il en reste de faux, c'est entièrement ma faute).

D'autres collègues ont grandement contribué au contenu de ce document, que ce soit en travaillant côte à côte, en me donnant des références bibliographiques, ou bien en relisant et critiquant certaines parties : Mathieu Boespflug, Alin Bostan, Alexandre Benoit, Jacques Carette, Jean-Marc Couveignes, Léonard Gérard, David Gruenewald, Romain Lebreton et Benjamin Smith.

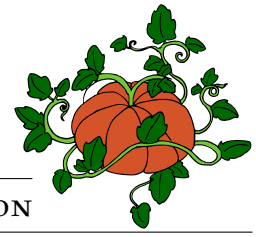
La relecture et la correction du manuscrit n'auraient pas été complètes, sans l'excellent travail des rapporteurs, Erich Kaltofen et Christophe Ritzenthaler, dont les remarques ont grandement contribué à son amélioration.

Mes recherches auraient peut-être été moins abouties si je n'avais pas eu la chance de travailler dans des laboratoires de recherche animés et stimulants. Au sein du LIX, ma pensée va à Daniel Augot, à Jean-François Biasse et aux autres collègues de l'équipe TANC et du séminaire Cal4doc avec qui j'ai eu le plaisir de travailler. Une partie considérable de ce travail a été développée dans le SCL à l'University of Western Ontario, dont les membres ont toujours été chaleureux et accueillants ; j'aimerais remercier en particulier Marc Moreno Maza et Stephen Watt. Edlyn Teske a été une charmante hôtesse lors de ma visite à Waterloo. On sait que les chercheurs sont de piètres planificateurs : sans l'aide d'Evelyne Rayssac tous ces voyages auraient été impossibles.

Si vous trouvez ce livre agréable à lire et plaisant aux yeux, c'est en partie grâce aux talents artistiques de Rachel Deyts. Ses talents linguistiques y sont aussi pour beaucoup, lorsque vous plongez dans mon français. Parler de ses talents culinaires et de compagne nous amènerait trop loin.

La beauté de la science réside dans son étendue. J'ai eu un immense plaisir à passer des heures avec Simone De Liberato à décortiquer les sujets scientifiques les plus variés, de la physique quantique à la théorie de la complexité. Son influence sur mes recherches, bien que difficile à localiser dans cette thèse, est indéniable.

Enfin, j'aimerais remercier les gens avec qui j'ai partagé les *autres* moments de ces trois années de thèse : ma famille, mes amis, le COF, mes camarades de Xdoc et de ELSE.



Introduction

Les corps finis sont au cœur de la technologie moderne, à tel point que la dernière génération de processeurs Intel Core possède une instruction matérielle (CLMUL) pour la multiplications dans $\mathbb{F}_{2^m}$ [GK08]. Cela tient au fait que les corps finis apparaissent partout dans le génie des télécommunications, en particulier en Codes Correcteurs d'Erreurs et Cryptographie. Cette thèse applique des techniques algorithmiques et algébriques avancées aux calculs dans les tours d'extensions sur les corps finis, avec pour but des applications à la cryptographie à base de courbes elliptiques.

Courbes elliptiques. En cryptographie à base de courbes elliptiques, afin de construire un système de chiffrement sûr, il faut sélectionner une courbe au nombre de points divisible par un grand nombre premier. La méthode préférée consiste à sélectionner une courbe au hasard et à appliquer un algorithme de comptage de points pour déterminer sa cardinalité. Le premier algorithme de comptage de points de courbes elliptiques de complexité polynomiale fut donné par Schoof [Sch85], puis amélioré par Atkin et Elkies [Atk88, Elk98, Sch95], et par la suite nommé SEA.

L'algorithme SEA suscita de l'intérêt pour l'utilisation effective des isogénies : ce sont des morphismes de groupes algébriques entre courbes elliptiques. Lorsqu'on calcule des isogénies sur des corps finis, il faut distinguer entre la caractéristique grande et la caractéristique quelconque. Dans le premier cas, on peut utiliser des algorithmes conçus pour la caractéristique 0, et ensuite réduire le résultat ; les méthodes de Elkies [Elk98, Mor95], Atkin [Sch95] et Bostan, Morain, Salvy et Schost [BMSS08] appartiennent à cette famille. Quand la réduction modulo la caractéristique introduit des divisions par 0, ces algorithmes ne s'appliquent plus.

Les deux premiers algorithmes pour calculer des isogénies en caractéristique quelconque furent donnés par Couveignes [Cou94, Cou96] ; les deux ont complexité polynomiale en la caractéristique, ce qui les rend peu pratiques pour des valeurs supérieures à 2 ou 3. Un algorithme spécifique pour la caractéristique 2 fut donné par Lercier [Ler96] : en pratique il est plus rapide que l'algorithme de Couveignes, mais sa complexité n'est pas bien comprise.

Après la découverte de méthodes p-adiques alternatives à SEA [Sat00, FGH00], l'intérêt pour le calcul d'isogénies s'est estompé. Pourtant, deux algorithmes p-adiques pour le calcul d'isogénies en caractéristique quelconque ont récemment été proposés par Joux et Lercier [JL06] et Lercier et Sirvent [LS08] ; ils montrent qu'il est possible d'éviter les divisions par 0 en liftant les courbes dans les p-adiques. Le second algorithme est actuellement celui qui a la meilleure complexité dans le cas de la caractéristique quelconque, sa dépendance en la caractéristique est seulement logarithmique.

Il est tout de même intéressant de remarquer qu'aucun algorithme pour le

calcul d'isogénies n'a une complexité optimale ou quasi-optimale, avec la seule exception de [BMSS08] dans un cas très spécifique.

Le point de départ de ce travail a été le deuxième algorithme de Couveignes [Cou96]. Il calcule une isogénie par interpolation sur les points de p^k-torsion de la courbe, pour k assez grand ; quand ces points ne sont pas définis sur le corps de base, il faut travailler dans des extensions de corps pour les trouver. Les extensions qui apparaissent naturellement dans ce calcul sont des corps de rupture de polynômes de la forme

$$X^p - X - \alpha ;$$

de telles extensions sont appelées d'Artin-Schreier.

Tours de corps finis. Mis à part l'addition, la multiplication et l'inversion, les opérations arithmétiques importantes dans une tour d'extensions finies sont sans aucun doute les traces relatives, les polynômes minimaux et les inclusions de corps. Pour les corps finis, il est possible d'ajouter des groupes de Galois effectifs à la liste, puisqu'il est relativement facile de calculer avec ces objets.

L'arithmétique des tours de corps finis est une question de première importance pour tout système de calcul formel, pourtant elle a reçu peu ou pas d'attention. On sait que Magma permet de gérer des diagrammes quelconques de corps finis depuis longtemps [BCS97a], mais il est difficile de dire quels algorithmes y sont implantés de nos jours et avec quelles complexités. Tous les autres résultats qui peuvent éventuellement s'appliquer aux tours de corps finis ont été obtenus dans le contexte plus général de la résolution de systèmes polynomiaux et de la géométrie algébrique effective, en particulier pour la résolution des ensembles triangulaires [DTGV01, GLS01, BSS03, PS06, LMMS07, DJMMS08, BLMM01, FGLM93, Rou99, ABRW96].

Dans le cas spécifique des tours d'Artin-Schreier, il n'y a pas énormément de littérature non plus. En s'appuyant sur des idées contenues dans [Con00], Cantor [Can89] construisit une tour d'Artin-Schreier avec des propriétés spécifiques, qu'il appliqua à la multiplication par FFT dans $\mathbb{F}_2[X]$. Dans [Cou00], Couveignes donna un algorithme pour le calcul d'isomorphismes entre tours d'Artin-Schreier ; néanmoins, son algorithme nécessite une multiplication rapide dans une tour, appelée « tour de Cantor » dans [Cou00], ayant la même forme que celle de [Can89]. Un tel algorithme n'est malheureusement pas dans la littérature, ce qui rend les résultats de [Cou00] difficiles à exploiter en pratique.

Le principe de transposition. Un des outils algorithmiques que nous allons étudier en détail et appliquer tout le long du document est le *principe de transposition*, qui est à la théorie des langages ce que la dualité est à l'algèbre.

Le principe de transposition fut découvert dans la théorie des circuits électriques par Bordewijk [Bor57], puis prouvé dans sa forme générale par Fiduccia [Fid73] ; mais ce n'est que bien plus tard, à travers les travaux de Kaltofen, Yagati, Shoup, von zur Gathen et autres [KY89, vzGS92, Sho94, Sho95, Sho99, HQZ04], qu'il est devenu populaire en calcul formel. L'un des énoncés possibles est le suivant :

> Soit $\mathcal{P}$ un ensemble quelconque. À tout algorithme R-algébrique, qui calcule une famille de fonctions linéaires $(f_p : M \to N)_{p \in \mathcal{P}}$, correspond un algorithme R-algébrique A^* qui calcule la *famille duale* $(f_p^* : N^* \to$

$M^*)_{p\in\mathcal{P}}$. Les complexités algébriques en temps et espace de A^* sont bornées par la complexité en temps de A.

Le principe de transposition est important en calcul formel car il permet d'obtenir des algorithmes asymptotiquement bons qui n'auraient pas paru évidents autrement. Un grand pas en avant dans sa compréhension fut fait par Bostan, Lecerf et Schost [BLS03] qui, en généralisant un travail de Shoup [Sho95], remarquèrent que la transposition peut être appliquée de façon systématique à un langage de programmation restreint. Il est aussi intéressant de remarquer que le principe de transposition a des liens importants avec la différentiation automatique [BS83, KY89, Kal00, GG05, Ser08].

Dans ce document nous enquêtons plus en détail sur les rapports entre la transposition et les langages de programmation. Nous travaillons dans le cadre de la théorie des langages purement fonctionnels typés [Pie02], car sa structure mathématique élégante nous permet de raisonner sur les programmes à un niveau algébrique.

Organisation du document, résultats. Ce document est divisé en quatre parties. Dans la partie I nous revenons sur les notions fondamentales d'algèbre et calcul formel dont nous allons nous servir par la suite.

La partie II a pour objet le principe de transposition. Au Chapitre 3 nous rappelons le modèle des circuits arithmétiques et le modèle des programmes sans branchements, puis prouvons le théorème de transposition pour chacun. Ensuite nous évoquons les liens avec la différentiation automatique. En complément, dans l'Annexe A, nous donnons une nouvelle preuve du théorème de transposition, à base de sémantique catégorique, et étudions ses implications pour l'implantation d'un DSL en Haskell ; il s'agit un travail commun avec Mathieu Boespflug.

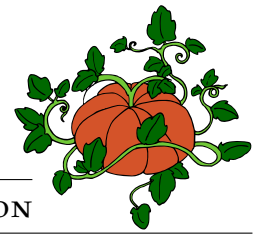
Le Chapitre 4 est une collaboration avec Éric Schost. Nous étudions les liens entre les circuits arithmétiques et les langages fonctionnels, puis montrons que la transposition peut être appliquée algorithmiquement à un langage fonctionnel générique.

La Partie III est dédiée à l'arithmétique dans les tours d'extensions. Nous commençons par rappeler la théorie des idéaux zéro-dimensionnels et la représentation univariée rationnelle au Chapitre 5. Ici, les résultats de la Partie II sont la clef pour obtenir des algorithmes asymptotiquement rapides. Les algorithmes de ce chapitre sont ensuite appliqués au Chapitre 6, où nous fournissons des algorithmes asymptotiquement bons pour les tours d'Artin-Schreier (fruit d'une autre collaboration avec Éric Schost).

Enfin, la Partie IV applique les résultats des chapitres précédents au calcul d'isogénies. Après quelques rappels sur les courbes elliptiques au Chapitre 7, nous passons en revue les algorithmes asymptotiquement meilleurs pour le calcul d'isogénies sur les corps finis. Nous commençons par rappeler l'algorithme BMSS pour le cas de la grande caractéristique [BMSS08] et sa généralisation à la caractéristique quelconque de Lercier et Sirvent [LS08] ; puis nous rappelons l'algorithme original de Couveignes [Cou96] et présentons des variantes améliorées avec un meilleur comportement asymptotique : les clefs pour ces résultats sont le Chapitre 6 et de nouvelles idées algorithmiques pour l'interpolation dans les tours d'extensions. Nous présentons aussi en Section 8.9 une généralisation surprenante de l'algorithme de Couveignes, qui permet le calcul d'isogénies de degré inconnu au même prix que le calcul d'isogénies de degré prescrit. Cette découverte éclaire

davantage la (sous)optimalité de l'algorithme de Couveignes et pourrait avoir des applications en cryptologie [GHS02b, GHS02a, Hes03, Tes06].

La théorie ne suffirait pas sans pratique. De la même façon, ce manuscrit ne serait pas complet s'il n'était accompagné par les paquets logiciels que nous avons développés. La grande majorité des algorithmes présentés ici a été implantée, paquetée et distribuée avec des licences *open source*. Ainsi, tous les algorithmes du Chapitre 6 sont disponibles dans la bibliothèque `FAAST`, écrite en `C++` et disponible à l'adresse `http://www.lix.polytechnique.fr/~defeo/FAAST/`. Au moment où nous écrivons, le compilateur pour le langage `transalpyne` du Chapitre 4 n'est pas encore distribué ; nous travaillons en ce moment à la première *stable release* et espérons commencer la distribution au début de 2011. Il sera disponible à l'adresse `http://transalpyne.gforge.inria.fr/`.



Introduction

Finite field arithmetic is at the heart of modern technology; this is so true, that the last generation of Intel Core processors supports a hardware instruction (CLMUL) for multiplication in $\mathbb{F}_{2^m}$ [GK08]. The reason is that finite fields appear everywhere in telecommunications engineering, in particular in Error Correcting Codes and Cryptography. This thesis applies advanced algorithmic and algebraic techniques to computations in towers of extensions of finite fields, in view of applications to elliptic curve cryptography.

Elliptic curves. In elliptic curve cryptography, in order to build a secure cryptosystem, one must select a curve whose number of points contains a large enough prime factor. The preferred method for doing this is to randomly select a curve and then use a point-counting algorithm to determine its cardinality. The first polynomial time point counting algorithm for elliptic curves was due to Schoof [Sch85], then improved by Atkin and Elkies [Atk88, Elk98, Sch95], henceforth named SEA.

The SEA algorithm raised interest in explicit computations with isogenies, i.e. algebraic group morphisms of elliptic curves. When computing isogenies over finite fields one must distinguish between the large and arbitrary characteristic. In the first case, one can use algorithms that work for characteristic 0, and then reduce the result; the methods of Elkies [Elk98, Mor95], Atkin [Sch95] and Bostan, Morain, Salvy and Schost [BMSS08] belong to this family. When the reduction modulo the characteristic introduces division by 0, these algorithms are not of help.

The first two algorithms to compute isogenies in arbitrary characteristic are due to Couveignes [Cou94, Cou96]: both have a polynomial dependency in the characteristic, which makes them unpractical for values higher than 2 or 3. An algorithm specific to characteristic 2 was given by Lercier [Ler96]; in practice it performs faster than Couveignes' algorithms, but its complexity is not well understood.

After the discovery of p-adic alternatives to the SEA algorithm [Sat00, FGH00] interest in computing isogenies in small characteristic was lost. Nevertheless, two p-adic algorithms were recently proposed by Joux and Lercier [JL06] and Lercier and Sirvent [LS08] to solve the isogeny problem in arbitrary characteristic. They show that it is possible to avoid divisions by 0 by lifting the curves in the p-adics. The last algorithm is currently the one having the best asymptotic complexity for the arbitrary characteristic case; its complexity in the characteristic is only logarithmic.

It is interesting to remark, however, that no algorithm to compute isogenies has optimal or quasi-optimal complexity, with the only exception of [BMSS08] on a very special case.

The starting point of this work was Couveignes' second algorithm [Cou96]. It computes an isogeny by interpolating it over the p^k-torsion points of the elliptic curves for a large enough k; when those points are not defined on the base field, one has to take towers of field extensions to find them. The field extensions that naturally arise when doing this computation are splitting fields of polynomials of the form

$$X^p - X - \alpha;$$

such extension are called Artin-Schreier extensions.

Towers of finite fields. Besides addition, multiplication and inversion, the arithmetic operations of interest in a tower of finite extensions arguably are relative traces, minimal polynomials and embeddings. For finite fields one could add explicit Galois groups to the list as these are relatively easy to compute with.

The arithmetic of towers of finite fields is a central question for any computer algebra system, however it has received little attention, if any. Magma is known for having had support for lattices of finite fields for a long time [BCS97a], but it is hard to tell which algorithms it implements nowadays and what their complexities are. All other results that can possibly apply to towers of finite fields were derived in the more general context of polynomial system solving and effective algebraic geometry, in particular in the resolution of triangular sets [DTGV01, GLS01, BSS03, PS06, LMMS07, DJMMS08, BLMM01, FGLM93, Rou99, ABRW96].

In the specific case of Artin-Schreier towers, the literature is not extensive either. Using ideas from [Con00], Cantor [Can89] constructs a particular Artin-Schreier tower that he applies to FFT multiplication in $\mathbb{F}_2[X]$. In [Cou00], Couveignes gives an algorithm to compute isomorphisms between Artin-Schreier towers; however, his algorithm needs as a prerequisite a fast multiplication algorithm in a tower, called a "Cantor tower" in [Cou00], having the same shape as the one in [Can89]. Such an algorithm is unfortunately not in the literature, making the results of [Cou00] non practical.

Transposition principle. One algorithmic tool that we shall study in depth and apply throughout the whole document is the *transposition principle*, which is the language-theoretic counterpart to algebraic duality.

The transposition principle was discovered in electrical network theory by Bordewijk [Bor57], then proved in its general form by Fiduccia [Fid73]; but it only became popular in computer algebra much later through the works of Kaltofen, Yagati, Shoup, von zur Gathen and others [KY89, vzGS92, Sho94, Sho95, Sho99, HQZ04]. One possible statement is:

> Let $\mathcal{P}$ be an arbitrary set. To any R-algebraic algorithm A computing a family of linear functions $(f_p : M \to N)_{p\in\mathcal{P}}$ corresponds an R-algebraic algorithm A^* computing the *dual family* $(f_p^* : N^* \to M^*)_{p\in\mathcal{P}}$. The algebraic time and space complexities of A^* are bounded by the time complexity of A.

The transposition principle is important in computer algebra because it allows to derive asymptotically good algorithms that were not otherwise evident. One big step forward in the understanding of it was done by Bostan, Lecerf and Schost [BLS03] who, extending work of Shoup [Sho95], remarked that transposition can be systematically applied to a restricted programming language. It is also

remarkable that the transposition principle has a strong connection with automatic differentiation [BS83, KY89, Kal00, GG05, Ser08].

In this document we investigate more in depth the relationships between the transposition principle and programming languages. We use the theory of typed purely functional languages [Pie02] as framework, because its elegant mathematical structure permits us to reason at an algebraic level on programs.

Outline of our contributions. This document is divided in four parts. Part I recalls the basic notions from algebra and computer algebra that we will use later.

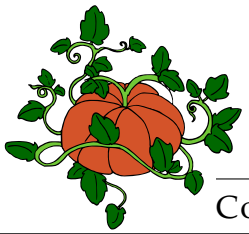
Part II studies the transposition principle. In Chapter 3 we review the arithmetic circuit model and the straight line program model, and prove the transposition theorem in them. Then we discuss the relationships with automatic differentiation. As a complement, in Appendix A we also give a new proof of the transposition theorem, using categorical semantics, and discuss its consequences on the implementation of a DSL in Haskell; this is joint work with Boespflug.

Chapter 4 is a collaboration with Schost. We study the relationships between the arithmetic circuit model and functional programming languages, then we show that transposition can be applied algorithmically to a generic functional language.

Part III is devoted to arithmetics in towers of extensions. We start by reviewing the general theory of zero-dimensional ideals and rational univariate representations in Chapter 5. Here, the results of Part II are the key to obtain asymptotically fast algorithms. The algorithms of this chapter are then applied in Chapter 6, where we provide asymptotically good algorithms for Artin-Schreier towers (fruit of another collaboration with Schost).

Finally Part IV applies the results of the previous chapters to isogeny computation. After some general references on elliptic curves in Chapter 7, we review in Chapter 8 the asymptotically fastest algorithms to compute isogenies over finite fields. We start by reviewing the BMSS algorithm for large characteristic [BMSS08] and its generalization for arbitrary characteristic by Lercier and Sirvent [LS08]; then we review Couveignes' original algorithm [Cou96], and present some improved variants with better asymptotic behavior: the key to this results are Chapter 6 and new ideas on interpolation in towers of extensions. We also present in Section 8.9 a surprising generalization of Couveignes' algorithm that allows to compute isogenies of unknown degree at the same cost of computing an isogeny of a given degree. This discovery sheds new light on the (sub)optimality of Couveignes' algorithm and can possibly find applications in cryptology [GHS02b, GHS02a, Hes03, Tes06].

Without practice, theory would not be as valuable. Similarly, this manuscript would not be complete if it was not accompanied by the software packages we developed. The great majority of the algorithms we present here have been implemented, packaged and distributed under open source licences. So, all the algorithms of Chapter 6 can be found in the `C++` library `FAAST`, available from `http://www.lix.polytechnique.fr/~defeo/FAAST/`. At the moment we write, the compiler for the language `transalpyne` of Chapter 4 is not distributed yet; we are currently working on the first stable release and hope to start distributing it by the beginning of 2011. It will be available from `http://transalpyne.gforge.inria.fr/`.



Contents

Intoduction (Français) 2

Intoduction (English) 6

List of Algorithms 11

I Prerequisites 13

1 Algebra 14
- 1.1 Linear algebra . . . 14
- 1.2 Basic Galois theory . . . 16
- 1.3 Basic algebraic geometry . . . 18

2 Algorithms and complexity 21
- 2.1 Asymptotic complexity . . . 21
- 2.2 Fundamental algorithms . . . 21

II The transposition principle 32

3 Algebraic Complexity and duality 33
- 3.1 Arithmetic circuits . . . 33
- 3.2 Multilinearity . . . 41
- 3.3 Straight Line Programs . . . 45
- 3.4 Automatic differentiation . . . 50

4 Automatic transposition of code 54
- 4.1 Inferring linearity . . . 54
- 4.2 `transalpyne` . . . 59

III Fast arithmetics using univariate representations 67

5 Trace computations 68
- 5.1 Decomposition of a zero-dimensional ideal . . . 69
- 5.2 Trace formulas . . . 71
- 5.3 Sitckelberger's theorem . . . 73
- 5.4 Rational Univariate Representation . . . 74

5.5 The univariate case . . . 77
5.6 Shoup's algorithm . . . 78
5.7 From univariate to bivariate and back again . . . 81

6 Artin-Schreier towers 83
6.1 Introduction . . . 83
6.2 Preliminaries . . . 84
6.3 A primitive tower . . . 87
6.4 Level embedding . . . 92
6.5 Frobenius and pseudotrace . . . 98
6.6 Arbitrary towers . . . 101
6.7 Experimental results . . . 106

IV Applications to isogenies and cryptography 110

7 Elliptic curves and isogenies 111
7.1 Definitions . . . 111
7.2 Curves over $\mathbb{C}$ . . . 117
7.3 Curves over finite fields . . . 119
7.4 Modular polynomials . . . 119

8 Computing isogenies over finite fields 120
8.1 Overview . . . 120
8.2 Vélu formulas . . . 121
8.3 BMSS . . . 123
8.4 Lercier-Sirvent . . . 126
8.5 Couveignes' algorithm . . . 128
8.6 The algorithm C2-AS . . . 134
8.7 The algorithm C2-AS-FI . . . 136
8.8 The algorithm C2-AS-FI-MC . . . 141
8.9 Isogenies of unknown degree . . . 142

9 Experimental results 146
9.1 Implementation of Couveignes' algorithm . . . 146
9.2 Implementation of C2-UD . . . 148
9.3 Implementation of Lercier-Sirvent . . . 149
9.4 Benchmarks . . . 149

A Categorical considerations 154
A.1 Categorical semantics of arithmetic circuits . . . 154
A.2 Coevaluation . . . 155
A.3 The tranposition theorem . . . 157
A.4 From circuits to function-level programming . . . 157
A.5 *Self-transposing* polynomial multiplication . . . 159

B Linearity inference of Karatsuba multiplication 162

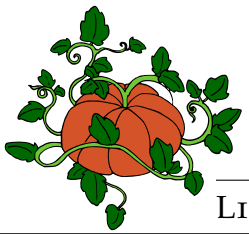
C Proof of Vélu's formulas 165

Conclusion 167

List of symbols 169

Index 171

Bibliography 174

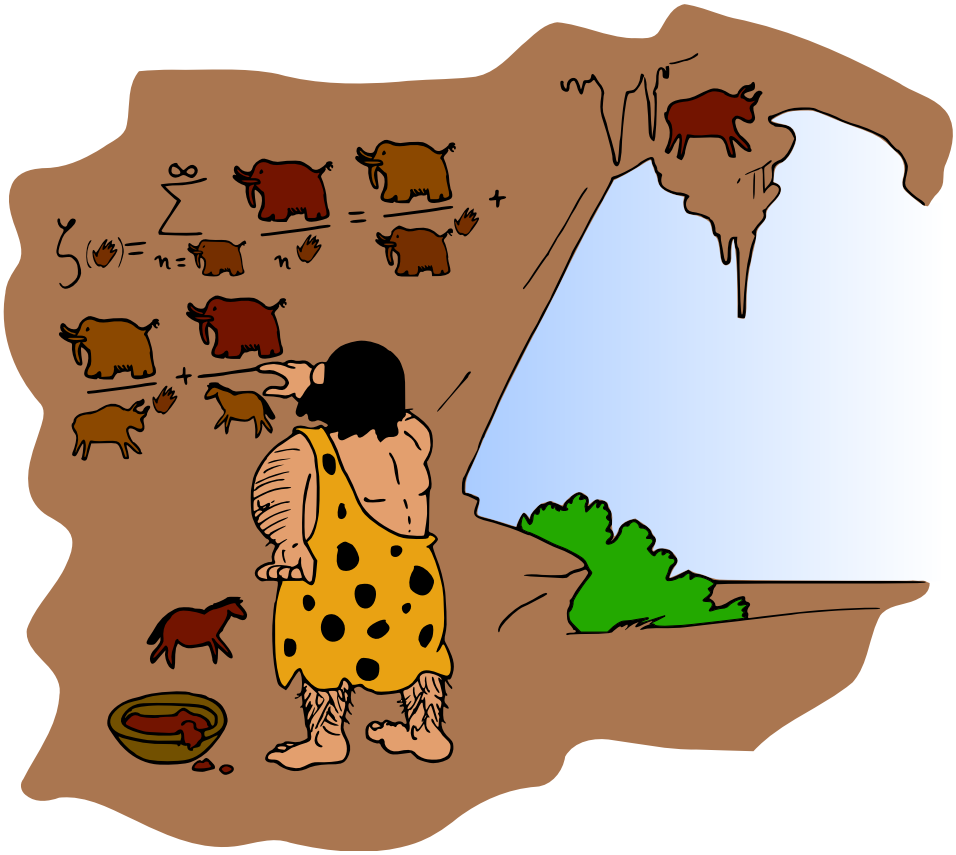


List of Algorithms

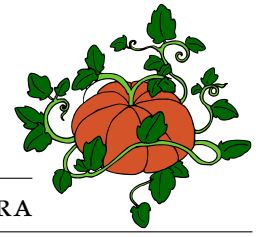
2.1 Iterated Frobenius . . . 25
2.2 Pseudotrace . . . 26
2.3 Subproduct tree . . . 27
2.4 Multipoint evaluation . . . 27
2.5 Interpolation . . . 28
2.6 Extended Euclidean algorithm . . . 28
3.1 R-algebraic transform . . . 49
3.2 Transposition of Algorithm 3.1 . . . 49
5.1 RUR . . . 80
6.1 NaiveCompose . . . 90
6.2 Compose . . . 90
6.3 MinimalPolynomial . . . 92
6.4 Push-down-rec . . . 95
6.5 Push-down . . . 95
6.6 Push-down-rec* . . . 96
6.7 Push-down* . . . 97
6.8 Lift-up . . . 97
6.9 IterFrobenius . . . 99
6.10 LittlePseudotrace . . . 100
6.11 Pseudotrace . . . 100
6.12 ApproximateAS . . . 103
6.13 Artin-Schreier . . . 104
6.14 ApplyIsomorphism . . . 104
6.15 ApplyInverse . . . 105
6.16 $\mathrm{RUR}_{\mathbf{s}_i}$ . . . 106
8.1 Solve differential equation . . . 124
8.2 BMSS . . . 126
8.3 Lercier-Sirvent . . . 127
8.4 Truncated subproduct tree . . . 140
8.5 Truncated fast interpolation . . . 140

Part I

Prerequisites



1 Chapter



Algebra

Here we recall the concepts from abstract algebra that will constitute the background for all the chapters that follow. One chapter is certainly not enough to present such a vast subject, hence we just recall the few definitions and properties that will help the reader understand the results presented in this document. The material of this chapter is mainly drawn from [Lan02, LN96, Sil86].

1.1 Linear algebra

In Part II we shall apply some classical linear algebraic tools to free modules over non-commutative ring. We recall here the fundamental concepts.

1.1.1 Bra-ket notation

It will be convenient to (ab)use Dirac's bra-ket notation to represent elements of modules. If $(M, +, \cdot)$ is a left R-module and $x \in (M, +)$ is an element of its underlying group, by $|x\rangle_R$ we mean the element obtained by lifting x in $(M, +, \cdot)$. We call $|x\rangle$ a *ket* and read it as "ket x".

The external multiplication by an element $a \in R$ will be written $a\,|x\rangle_R$; if $f : M \to N$ is a left module morphism, we write $f\,|x\rangle_R$ for $f(|x\rangle_R)$. By a slight abuse of notation we may write $|ax\rangle_R$ and $|f(x)\rangle_R$ for $a\,|x\rangle_R$ and $f\,|x\rangle_R$ respectively. When R is clear from the context, a ket can be simply written as $|x\rangle$.

In a symmetric way, elements of right R-modules will be written ${}_R\langle x|$, which we call a *bra* and read as "bra x". External multiplication will be written as ${}_R\langle x|\,a$ and application of a right module morphism as ${}_R\langle x|\,f$.

Let M be a right module and N a left module. A *bilinear form* on $M \times N$ is a map $f : M \times N \to R$ such that for any $x \in M$, the map

$$|y\rangle \mapsto f(x, y)$$

is a left module morphism, and for any $y \in N$, the map

$$\langle x| \mapsto f(x, y)$$

is a right module morphism. If f is a bilinear form, we write $\langle x|y\rangle_f$ for $f(x, y)$, or simply $\langle x|y\rangle$ when f is clear from the context. Note that textbooks usually define bilinear forms only when R is commutative, in our more general setting some common properties of bilinear forms fail to hold, for example $\langle xa|y\rangle$ is not necessarily equal to $\langle x|ay\rangle$.

If M is a left (right) module, we denote by $M^* = \hom(M, R)$ the *dual module* of M, it is a right (left) module. Any bilinear form f gives rise to a morphism

$\phi_f : M \to N^*$ of right modules where $\langle x|\,\phi_f$ is the linear form $y \mapsto \langle x|y\rangle$. Similarly, f gives rise to a morphism $\phi^f : N \to M^*$ of left modules. The maps $f \mapsto \phi_f$, $f \mapsto \phi^f$ and their obvious inverses induce group isomorphisms between $\hom(M, N^*)$, $\hom(N, M^*)$ and the group of bilinear forms on $M \times N$. A bilinear form f is said to be *non-degenerate* if ϕ_f and ϕ^f are module isomorphisms.

1.1.2 Matrices and morphisms

$M = M_1 \oplus \cdots \oplus M_n$ be a left module and $N = N_1 \oplus \cdots \oplus N_m$ be a right module. Let ι_i be the injections $M_i \to M$ and let π_j be the projections $N \to N_i$, then a linear map $f : M \to N$ is uniquely determined by the maps $\pi_j \circ f \circ \iota_i$. If we consider $m \times n$ matrices whose (j, i)-th coefficient is in $\hom(M_i, N_j)$, then we verify that there is a group isomorphism between $\hom(M, N)$ and this group of matrices. Furthermore, let $f : M \to N$ and $g : N \to O$ and let M_f and M_g be the matrices that are associated respectively, then the matrix associated to $g \circ f$ is $M_g M_f$, where the product of two entries is defined as composition of morphisms. This induces a ring isomorphism between $\mathrm{End}(M)$ and the ring of square matrices with entries in $\hom(M_i, M_j)$.

Consider R as an R-module over itself, a linear map from R to itself is uniquely determined by the image of 1, hence $\mathrm{End}(R) \cong R^{\mathrm{op}}$. As a consequence, there is a group isomorphism $\hom(R^n, R^m) \cong \mathcal{M}_{m\times n}(R^{\mathrm{op}})$, and matrix multiplication is equivalent to composition as above. Hence, if M is a free module, for any fixed basis $\mathbf{B}$ of cardinality n we have an isomorphism of rings $\mathrm{End}_R(M) \cong \mathcal{M}_n(R^{\mathrm{op}})$; in particular $\mathrm{Aut}(M) \cong \mathrm{GL}_n(R^{\mathrm{op}})$ as groups.

Let R be commutative, then $R^{\mathrm{op}} = R$. We denote by $M_{\mathbf{B}}(f)$ the matrix associated to $f \in \mathrm{End}_R(M)$ with respect to the basis $\mathbf{B}$. If $\mathbf{B}'$ is another basis, it has the same cardinality as $\mathbf{B}$. Then, there is an invertible matrix B such that $A \mapsto B^{-1}AB$ is the automorphism of $\mathcal{M}_n(R)$ that sends $M_{\mathbf{B}}(f)$ over $M_{\mathbf{B}'}(f)$. Hence, any property of matrices that is invariant by similarity, can be defined for linear operators. We define the *trace* of a linear operator as $\mathrm{Tr}\, f = \mathrm{Tr}\, M(f)$, and its *determinant* as $\det f = \det M(f)$.

1.1.3 Duality

We fix a non-degenerate bilinear form f on $M \times N$. Let $g \in \mathrm{End}(M)$, then the map

$$(x, y) \mapsto \langle g(x)|y\rangle_f$$

is a bilinear form. On the other hand, let h be a bilinear form on $M \times N$, for any $x \in M$ the map $h_x : |y\rangle \mapsto \langle x|y\rangle_h$ is a linear form on N, thus $h_x \in N^*$. From the non-degeneracy of f we deduce that there is an unique element $x' \in M$ such that $\langle x'|y\rangle_f = \langle x|y\rangle_h$ and it is clear that the map $\langle x| \mapsto \langle x'|$ is an endomorphism of M. It is evident that the two maps are each other's inverse, thus we have a group isomorphism between $\mathrm{End}(M)$ and the group of bilinear forms. An analogous argument shows that $\mathrm{End}(N)$ is isomorphic to the group of bilinear forms and ultimately $\mathrm{End}(M) \cong \mathrm{End}(N)$.

A consequence of this is that for any linear operator $g \in \mathrm{End}(M)$ there is an operator $g^* \in \mathrm{End}(N)$ such that

$$\langle g(x)|y\rangle_f = \langle x|g^*(y)\rangle_f$$

for any $x \in M$ and $y \in N$. We define similarly h^* when $h \in \mathrm{End}(N)$, obviously $(g^*)^* = g$. The operator g^* is called the *dual* of g with respect to f. In general, whenever it is clear from the context that g belongs to $\mathrm{End}(M)$ (or to $\mathrm{End}(N)$), we simply write

$$\langle x|g|y\rangle \stackrel{\text{def}}{=} \langle g(x)|y\rangle = \langle x|g^*(y)\rangle\,.$$

More generally, Let $f : M \times M' \to R$ and $g : N' \times N \to R$ be two non-degenerate bilinear forms, by the same technique as above we can show that there is a group isomorphism between $\hom_R(N, M')$, $\hom(M, N')$ and the bilinear forms on $M \times N$. Then, for any $h : N \to M'$ there is an unique $h^* : M \to N'$ such that

$$\langle x|h|y\rangle \stackrel{\text{def}}{=} \langle x|h(y)\rangle_f = \langle h^*(x)|y\rangle_g\,.$$

We also call h^* the *dual* of h.

The canonical example of non-degenerate bilinear forms is obtained by considering the family of forms on $M^* \times M$ defined by

$$\langle \ell|x\rangle = \ell(x).$$

For any $f : M \to N$, we define the dual map $f^* : N^* \to M^*$ as the map that sends a form $\ell \in N^*$ over the form $\ell \circ f$ in M^*; it is easy to verify that

$$\langle \ell|f|x\rangle = \langle \ell|f(x)\rangle = \langle f^*(\ell)|x\rangle = \ell(f(x)).$$

If M is a free module and $\mathbf{B} = \{\boldsymbol{e}_1, \ldots, \boldsymbol{e}_n\}$ a basis, the *dual basis* $\mathbf{B}^*$ is the unique basis $\{\boldsymbol{e}_1^*, \ldots, \boldsymbol{e}_1^*\}$ of M^* such that

$$\langle \boldsymbol{e}_i^*|\boldsymbol{e}_j\rangle = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

If elements of M and M^* are represented, respectively, as vectors over $\mathbf{B}$ and $\mathbf{B}^*$, then the bilinear form $\langle \ell|x\rangle = \ell(x)$ is given by the inner product

$$\left\langle \ell_1 \quad \cdots \quad \ell_n \middle| \begin{matrix} x_1 \\ \vdots \\ x_n \end{matrix} \right\rangle = \sum_i x_i \ell_i$$

(notice how the product is swapped, this is because $\mathrm{End}(R) \cong R^{\mathrm{op}}$). Now, if M and N are free modules with a fixed basis, a linear map $f : M \to N$ is isomorphic to a matrix with entries in R^{op}. Then the application $f\,|x\rangle$ is just matrix-vector multiplication, while $\langle \ell|\,f^*$ is vector-matrix multiplication by the same matrix. This justifies the notation $\langle \ell|A|x\rangle$ where A is the matrix associated to f.

1.2 Basic Galois theory

In Parts III and IV we shall need some basic Galois theory of finite fields. We recall here the general concepts.

1.2.1 Galois extensions

Let $\mathbb{K}$ be a field. The *splitting field* of a family of polynomials $(Q_i)_{i\in I}$ in $\mathbb{K}[X]$ is defined as an extension $\mathbb{L}$ of $\mathbb{K}$ where all the Q_i's factor completely into linear factors, and such that $\mathbb{L}$ is generated over $\mathbb{K}$ by the roots of the Q_i; the splitting field is unique up to isomorphism. An algebraic field extension $\mathbb{L}/\mathbb{K}$ such that $\mathbb{L}$ is the splitting field of a family of polynomials in $\mathbb{K}[X]$ is called a *normal extension*.

Let $\mathbb{L}/\mathbb{K}$ be an algebraic field extension, an element $x \in \mathbb{L}$ is said to be *separable* over $\mathbb{K}$ if its minimal polynomial over $\mathbb{K}$ has no multiple roots in $\mathbb{L}$. $\mathbb{L}/\mathbb{K}$ is said to be *separable* if every $x \in \mathbb{L}$ is separable over $\mathbb{K}$. An algebraic field extension is said to be a *Galois extension* if it is both separable and normal.

THEOREM 1.1 *Let $\mathbb{L}/\mathbb{K}$ be a finite Galois extension, then there exists an element $x \in \mathbb{L}$, called a* primitive element, *such that $\mathbb{L} \cong \mathbb{K}[x]$.*

Let $\mathbb{L}/\mathbb{K}$ be a Galois extension, the group of automorphisms of $\mathbb{L}$ that fix $\mathbb{K}$ is called the *Galois group* of $\mathbb{L}/\mathbb{K}$ and is denoted by $\mathrm{Gal}(\mathbb{L}/\mathbb{K})$. Let G be a group of automorphisms of a field $\mathbb{K}$, by $\mathbb{K}^G$ we denote the subfield of $\mathbb{K}$ consisting in the elements such that $\sigma(x) = x$ for any $\sigma \in G$. Obviously, $\mathbb{K} = \mathbb{L}^{\mathrm{Gal}(\mathbb{L}/\mathbb{K})}$.

THEOREM 1.2 *Let $\mathbb{L}/\mathbb{K}$ be a finite Galois extension. Let H be a subgroup of $G = \mathrm{Gal}(\mathbb{L}/\mathbb{K})$, the map $H \mapsto \mathbb{L}^H$ is a bijection between the subgroups of G and the subfields of $\mathbb{L}$ containing $\mathbb{K}$. The extension $\mathbb{L}^H/\mathbb{K}$ is Galois if and only if H is a normal subgroup of G; in this case its Galois group is isomorphic to G/H.*

Let $\mathbb{L}/\mathbb{K}$ be a Galois extension and let $x \in \mathbb{L}$. The elements $\sigma(x)$ for $\sigma \in \mathrm{Gal}(\mathbb{L}/\mathbb{K})$ are called the *conjugates* of x under the action of $\mathrm{Gal}(\mathbb{L}/\mathbb{K})$; they are the roots of the minimal polynomial of x over $\mathbb{K}$.

Let $\mathbb{K}$ be a field, an element $x \in \mathbb{K}$ such that $x^n = 1$ is called an n-th *root of unity*. If the characteristic of $\mathbb{K}$ does not divide n, the polynomial $X^n - 1$ has n distinct roots in $\bar{\mathbb{K}}$ and they form a multiplicative group, denoted by μ_n; it is a cyclic group, its generators are called the *primitive* roots of unity. If $\mathbb{K}$ has characteristic $p > 0$, then $X^{p^m} - 1$ has only one root, namely 1, thus μ_{p^m} is the trivial group.

The *Euler function* $\varphi : \mathbb{N} \to \mathbb{N}$ is defined as

$$\begin{aligned} \varphi(1) &= 1, \\ \varphi(p^r) &= p^{r-1}(p-1) && \text{for } p \text{ prime}, r \geqslant 1, \\ \varphi(nm) &= \varphi(n)\varphi(nm) && \text{when } \gcd(n,m) = 1. \end{aligned}$$

The Euler function counts the number of generators of the cyclic group with n elements, thus, when the characteristic of the field does not divide n, the number of primitive roots of unity is equal to $\varphi(n)$.

THEOREM 1.3 *Let x be a primitive n-th root of unity in an algebraic closure of $\mathbb{Q}$, then*

$$[\mathbb{Q}(x) : \mathbb{Q}] = \varphi(n).$$

If x is an n-th root of unity, its minimal polynomial over $\mathbb{Q}$ is called the n-th *cyclotomic polynomial* and is denoted by Φ_n; it is a monic polynomial with coefficients in $\mathbb{Z}$. Φ_n is an irreducible factor of $X^n - 1$ over $\mathbb{Q}$, its roots are all the primitive n-th roots of unity, hence $\deg \Phi_n = \varphi(n)$.

Let $\mathbb{L}/\mathbb{K}$ be a finite extension and let $x \in \mathbb{L}$, the map $M_x : a \mapsto xa$ is an automorphism of the $\mathbb{K}$-vector space $\mathbb{L}$. The minimal polynomial of its matrix with respect to any basis is equal to the minimal polynomial of x over $\mathbb{K}$. The trace of M_x is called the *trace* of x and is denoted by $\mathrm{Tr}_{\mathbb{L}/\mathbb{K}}(x)$; its determinant is called the *norm* of x and is denoted by $\mathrm{N}_{\mathbb{L}/\mathbb{K}}(x)$.

Proposition 1.4 *Let $\mathbb{L}/\mathbb{K}$ and $\mathbb{K}/k$ be finite extensions and let $G = \mathrm{Gal}(\mathbb{L}/\mathbb{K})$. We have the following identities*

$$\mathrm{Tr}_{\mathbb{L}/\mathbb{K}}(x) = \sum_{\sigma \in G} \sigma(x), \qquad \mathrm{Tr}_{\mathbb{L}/k} = \mathrm{Tr}_{\mathbb{K}/k} \circ \mathrm{Tr}_{\mathbb{L}/\mathbb{K}},$$
$$\mathrm{N}_{\mathbb{L}/\mathbb{K}}(x) = \prod_{\sigma \in G} \sigma(x), \qquad \mathrm{N}_{\mathbb{L}/k} = \mathrm{N}_{\mathbb{K}/k} \circ \mathrm{N}_{\mathbb{L}/\mathbb{K}}.$$

The trace is a morphism of $\mathbb{K}$-vector spaces from $\mathbb{L}$ to $\mathbb{K}$, the norm is a multiplicative morphism of groups from $\mathbb{L}^*$ to $\mathbb{K}^*$.

1.2.2 Finite fields

Let $\mathbb{K}$ be a *finite field*. It has necessarily characteristic $p > 0$, thus it must contain $\mathbb{Z}/p\mathbb{Z}$ as a subfield. $\mathbb{Z}/p\mathbb{Z}$ is called the *prime field* of $\mathbb{K}$ and is denoted by $\mathbb{F}_p$. Since $\mathbb{K}$ is a vector space over $\mathbb{F}_p$, it must have cardinality $q = p^n$ for some n, hence its multiplicative group has order $q - 1$.

As a consequence, the elements of $\mathbb{K}^*$ must be roots of the polynomial $X^{q-1} - 1$. The fact that p does not divide $q - 1$ implies that $\mathbb{K}$ is isomorphic to $\mathbb{F}_p[\zeta]$, where ζ is a primitive $(q-1)$-th root of unity in $\bar{\mathbb{F}}_p$. This implies that, up to isomorphism, there is an unique finite field containing q elements, we denote by $\mathbb{F}_q$ this field.

Using the same arguments, it is easy to show that for any $m \geqslant 1$, $\mathbb{F}_{q^m}$ contains a subfield isomorphic to $\mathbb{F}_q$. The map $\varphi_q : \mathbb{F}_{q^m} \to \mathbb{F}_{q^m}$ sending $x \mapsto x^q$ is a morphism of fields that fixes $\mathbb{F}_q$, it is called the *Frobenius automorphism* of $\mathbb{F}_{q^m}/\mathbb{F}_q$. We now give the main result about the Galois theory of finite fields.

Proposition 1.5 *The Galois group of $\mathbb{F}_{q^m}/\mathbb{F}_q$ is a cyclic group of order m; it is generated by the Frobenius automorphism φ_q.*

1.3 Basic algebraic geometry

1.3.1 Noetherian rings

A ring R is called *Noetherian* if any ascending chain of ideals eventually terminates. Being Noetherian is a very stable condition: fields and principal ideal domains, quotients of Noetherian rings, rings of polynomials in finitely many variables over a Noetherian ring, are all Noetherian. In particular, all the rings we will work with in this document are Noetherian.

A proper ideal I is *maximal* if it is not strictly contained in any proper ideal, this is equivalent to R/I being a field. A proper ideal is *prime* if R/I is an integral domain; *primary* if $ab \in I$ implies that $a \in I$ or $b^n \in I$ for some n. The *radical* of an ideal I is the ideal

$$\sqrt{I} = \{f \mid f^r \in I \text{ for some } r \geqslant 0.\}. \tag{1.1}$$

An ideal is said to be radical if $\sqrt{I} = I$. The radical of a primary ideal is prime.

An ideal I is said to be *reducible* if it is strictly contained in two ideals I_1, I_2 such that $I = I_1 \cap I_2$, *irreducible* otherwise. Any primary ideal is irreducible; we have the following two fundamental results about reducibility.

PROPOSITION 1.6 *Let R be Noetherian. Any radical ideal I admits an unique decomposition*

$$I = P_1 \cap \cdots \cap P_n \tag{1.2}$$

with P_i prime and $P_i \not\subset P_j$ for $i \neq j$.

THEOREM 1.7 (Primary decomposition) *Let R be Noetherian. Any ideal I admits a decomposition*

$$I = Q_1 \cap \cdots \cap Q_n \tag{1.3}$$

into primary ideals. Furthermore, $\sqrt{Q_i}$ is uniquely determined.

Now we state a fundamental lemma that we will repeatedly use in the next chapters.

LEMMA 1.8 (Chinese remainder theorem) *Let $I_1, \ldots, I_n$ be pairwise coprime ideals (i.e. $I_i + I_j = R$ if $i \neq j$). Then the canonical morphism $A \to \prod_j A/I_j$ gives an isomorphism of rings*

$$A/I_1 \cap \cdots \cap I_n \cong \prod_j A/I_j; \tag{1.4}$$

and the intersection $I_1 \cap \cdots \cap I_n$ equals the product $I_1 \cdots I_n$.

1.3.2 Algebraic varieties

We now consider the polynomial ring $\mathbb{K}[x_1, \ldots, x_n]$, where $\mathbb{K}$ is a perfect field with algebraic closure $\bar{\mathbb{K}}$. To any ideal I, we associate its *set of zeros*

$$V(I) = \{x \in \bar{\mathbb{K}}^n \mid f(x) = 0 \text{ for any } f \in I\}. \tag{1.5}$$

Reciprocally, to any $V \subset \bar{\mathbb{K}}^n$ we associate the ideal *vanishing at* V

$$I(V) = \{f \in \bar{\mathbb{K}}[x_1, \ldots, x_n] \mid f(x) = 0 \text{ for any } x \in V\}. \tag{1.6}$$

A subset of $\bar{\mathbb{K}}^n$ is called an *affine algebraic set* if it is the set of zeros of an ideal of $\bar{\mathbb{K}}[x_1, \cdots, x_n]$. The *affine space* of dimension n, denoted by $\mathbb{A}^n$, is the affine algebraic set associated to the zero ideal.

An algebraic set V is *defined over* $\mathbb{K}$ if $I(V)$ has a set of generators in $\mathbb{K}[x_1, \ldots, x_n]$; in this case we denote by $V(\mathbb{K})$ the subset $V \cap \mathbb{K}^n$.

An algebraic set is *irreducible* if it cannot be written as the union of two proper algebraic sets; equivalently, it is irreducible if $I(V)$ is prime. An irreducible affine algebraic set is called an *affine variety*.

There is also an equivalent notion of *projective variety* for homogeneous ideals. The projective variety associated to the zero ideal is called the *projective space* of dimension n, and is denoted by $\mathbb{P}^n$.

In the sequel we shall drop the qualificatives "affine" or "projective", and simply speak of *algebraic varieties* whenever definitions/theorems are identical.

Theorem 1.9 (Nullstellensatz) *Let* I *be an ideal and* V *an algebraic set. We have the following identities*

$$I(V(I)) = \sqrt{I}, \qquad V(I(V)) = V. \tag{1.7}$$

If V is a variety defined over $\mathbb{K}$, its *coordinate ring* is

$$\mathbb{K}[V] \stackrel{\text{def}}{\equiv} \mathbb{K}[x_1, \ldots, x_n]/I(V); \tag{1.8}$$

the *function field* $\mathbb{K}(V)$ is its field of fractions.

The *dimension* of a variety V is the length d of the longest chain of distinct non-empty subvarieties of V

$$V_d \subset \cdots \subset V_1 \subset V. \tag{1.9}$$

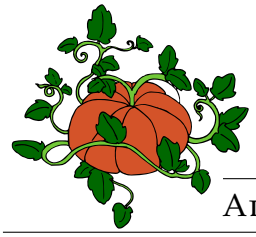
Equivalently, it is the length of the longest strictly decreasing chain of prime ideals in $\mathbb{K}[V]$. Yet another way of defining it, is the degree of transcendence of $\mathbb{K}(V)$ over $\mathbb{K}$.

If V_1 and V_2 are two varieties, an *affine rational map* is a map

$$\begin{aligned} \phi : V_1 &\to V_2, \\ x &\mapsto (f_1(x), \ldots, f_n(x)), \end{aligned} \tag{1.10}$$

with $f_1, \ldots, f_n \in \bar{\mathbb{K}}(V_1)$ and such for any point P at which $f_1, \ldots, f_n$ are defined, $\phi(P) \in V_2$. An equivalent definition exists for *projective rational maps*.

A rational map that is defined at any point of V_1 is called a *morphism*. A rational map (a morphism) is *defined over* $\mathbb{K}$ if $f_1, \ldots, f_n \in \mathbb{K}(V)$.



Chapter 2

Algorithms and complexity

2.1 Asymptotic complexity

We shall measure the complexity of the algorithms that appear in this document using the classical O (big-Oh) notation. Given two functions $f, g : \mathbb{N} \to \mathbb{N}$, by $f \in O(g)$ we mean that there are an x_0 and a constant M such that

$$f(x) < Mg(x) \quad \text{for any } x > x_0. \tag{2.1}$$

Similarly, we shall use the Ω and Θ notations to state lower bounds and tight bounds: the definition of $f \in \Omega(g)$ is like Eq. (2.1), but with the inequality turned the other way. The definition of Θ is $f \in \Theta(g)$ if and only if $f \in O(g)$ and $f \in \Omega(g)$.

We shall also make use of the notation $\tilde{O}_x$ (soft-Oh of x) that forgets polylogarithmic factors in the variable x, thus $O(xy \log x \log y) \subset \tilde{O}_x(xy \log y) \subset \tilde{O}_{x,y}(xy)$. We simply write $\tilde{O}$ when the variables are clear from the context.

Many algorithms below rely on fast multiplication; thus, we let $M_R : \mathbb{N} \to \mathbb{N}$ be a *multiplication function*, such that polynomials in $R[X]$ of degree less than n can be multiplied in $M_R(n)$ arithmetic operations. We drop the index R when the ring is clear from the context. To simplify expressions, following [vzGG99, §8.3], we shall assume that $M(n)$ is

- superlinear: $M(n)/n \geqslant M(m)/m$ if $n \geqslant m$,
- at most quadratic: $M(mn) \leqslant m^2 M(n)$.

We shall see soon that these assumptions are reasonable ones.

The cost of *modular composition* –that is computing $f \circ g \bmod h$, for $f, g, h \in R[X]$ of degrees at most n, and h monic– will be written $C(n)$. We shall make the assumption that $C(n) \geqslant M(n)$ for any n; the next section will review algorithms for modular composition.

2.2 Fundamental algorithms

In this section we review some fundamental algorithms that we will repeatedly use in the rest of the document. Most of the algorithms we present are taken from [vzGG99]; another source of inspiration is [BCG+10].

2.2.1 Polynomial multiplication

Multiplication of polynomials with coefficients in a ring is a fundamental underpinning to which most of the algorithms in computer algebra reduce.

In the previous section we introduced the notation $M(n)$ to denote the number of operations in R required to multiply two polynomials of degree at most n in $R[X]$. Using the school-book algorithm, we have $M(n) \in O(n^2)$. The first major step forward in the complexity of multiplication was done by Karatsuba [KO63]. He observed that using the formula

$$f = f_1 X^n + f_2, \qquad g = g_1 X^n + g2,$$
$$fg = f_1 g_1 X^{2n} + \big((f_1 + f_2)(g_1 + g_2) - f_1 g_1 - f_2 g_2\big) X^n + f_2 g_2,$$

multiplication can be computed recursively using only 3 recursive calls. It follows that $M(n) \in O(n^{\log_2 3})$.

When the base ring R is a field containing a primitive n-th root of unit ω, polynomials can be multiplied by evaluating at the powers of ω, multiplying each evaluation, and interpolating back. The map that sends a polynomial of degree n over its evaluations at the n-th roots of unit is called *discrete Fourier transform*, there are many algorithms of complexity $O(n \log n)$ to compute it, they all go under the generic name of *fast Fourier transform* (FFT).

Thus, multiplication in certain fields can be carried out in $O(n \log n)$ operations. Schönhage and Strassen's method [SS71], along with its generalizations [Sch77, CK91], adjoins enough roots of unit to any ring R by taking an extension of it; this yields an algorithm of complexity $O(n \log n \operatorname{loglog} n)$ to multiply polynomials of degree n in $R[X]$.

2.2.2 Formal power series

We denote by $R[[X]]$ the ring of *formal power series* on R. Its elements are the sequences $(f_i)_{i>0}$ of elements of R, they are denoted by

$$f(X) = \sum_{i \geqslant 0} f_i X^i. \tag{2.2}$$

Multiplication and evaluation are defined in the obvious way. An element $f \in R[[X]]$ is invertible if and only if f_0 is an unit of R.

Since formal power series are infinite objects, to be used in a discrete algorithm they must be approximated. We denote by $f \bmod X^n$ the polynomial

$$f \bmod X^n = \sum_{0 \leqslant i < n} f_i X^i. \tag{2.3}$$

We write $f = g + O(X^n)$, where g is a polynomial or a power series, whenever

$$f \bmod X^n = g \bmod X^n,$$

and we say that g approximates f to the precision n.

Using polynomial multiplication, the product of two series known up to precision n can be computed in $M(n)$ operations.

Derivative, integral. We define the derivative of a power series as

$$f'(X) = \sum_{i \geqslant 0} (i+1) f_{i+1} X^i; \tag{2.4}$$

if R contains $\mathbb{Q}$, we also define the integral as

$$\int f(X) = \sum_{i \geqslant 0} \frac{f_i}{i+1} X^{i+1}. \tag{2.5}$$

Derivatives and integrals up to precision n can be computed in $O(n)$ operations by their definition.

Logarithm, exponential. In what follows, we suppose that R contains $\mathbb{Q}$. The logarithm of a power series f such that $f(0) = 1$ is defined as

$$\log f = \int \frac{f'}{f}. \tag{2.6}$$

The exponential of a power series f such that $f(0) = 0$, is defined as

$$\exp(f) = 1 + f/1! + f^2/2! + \cdots \tag{2.7}$$

First order linear differential equations. All the usual identities involving multiplication, derivatives, integrals, logarithms and exponentials are verified on power series. An immediate consequence of this is a formula to solve first order linear differential equations due to Brent and Kung [BK78].

Let $f, g \in R[[X]]$, the equation

$$y' = f(X)y + g(X) \tag{2.8}$$

with initial condition $y(0) = a$ has solution

$$y(X) = \frac{1}{j(X)} \left(a + \int g(X) j(X) \right), \tag{2.9}$$

where $j = \exp(-\int f)$; the verification is immediate.

In the next subsection we shall see that multiplicative inverses, logarithms, exponentials and powers up to precision n can all be computed in $O(M(n))$ operations, thus formula (2.9) can also be applied at the same cost.

Note. If R does not contain $\mathbb{Q}$, but has characteristic 0, it is easy to use the previous definitions by working in $R[2^{-1}, 3^{-1}, \ldots]$ and taking the result back in R when needed. In characteristic different from 0, these definition do not make sense anymore because Eq. (2.5) introduces a division by 0. However, when $2, 3, \ldots, n$ are invertible in R, we can still do computations on power series truncated to the order n.

2.2.3 Newton's iteration

Let $\Phi : \mathbb{R} \to \mathbb{R}$ be a C^1 function, Newton's iteration is a classical method to approximate a root x of Φ. Start from an approximation x_0, and *linearize* Φ to compute

$$x_1 = x_0 - \frac{\Phi(x_0)}{\Phi'(x_0)}, \tag{2.10}$$

then iterate this step until the desired precision is obtained. When x_0 is taken close enough to a root, and when the derivative at this root is non-zero, Newton's

iteration converges *quadratically* to the solution, meaning that at each iteration the distance to the solution is squared.

In computer algebra, Newton's iteration is applied to operators $\Phi : R[[X]] \to R[[X]]$ on formal power series; in this context, *quadratic* convergence means that the number of correct terms is doubled at each iteration. Many fast algorithms for some fundamental operations on power series and polynomials are obtained by this method, here we summarize the most important ones.

Inversion. If $f \in R[[X]]$ is invertible, the operator $\Phi(y) = 1/y - f$ applied to $y_0 = 1/f(0)$ converges quadratically to the inverse of f. Since the iteration associated to Φ is

$$y_{i+1} = y_i(2 - y_i f), \tag{2.11}$$

the cost of inverting a power series is $O(M(n))$. From Eq. (2.6) we deduce that computing the logarithm of a power series has the same cost.

Another important consequence of this algorithm is that the Euclidean division of polynomials of degree at most n can also be performed in $O(M(n))$ operations.

Exponential. If f is such that $f(0) = 0$, we compute its exponential using the operator $\Phi(y) = f - \log y$, which gives the iteration

$$y_{i+1} = y_i(1 + f - \log y). \tag{2.12}$$

Thus, the cost of computing an exponential is $O(M(n))$ too. Using the formula

$$f^\alpha = \exp(\alpha \log f), \tag{2.13}$$

we deduce that, in characteristic 0, computing arbitrary rational powers of power series costs $O(M(n))$ too.

2.2.4 Modular composition

Given polynomials $f, g, h \in R[X]$ of degree at most n with h monic, the *modular composition* requires to compute

$$f(g(X)) \mod h(X); \tag{2.14}$$

the special case where $h = X^n$ permits us to compute the *composition of power series* truncated to the order n.

Modular composition is a fundamental algorithm with lots of applications, the most relevant being polynomial factorization [vzGS92, KS98] and computation of minimal polynomials (see Remark 5.23).

Since many algorithms in this document make use of modular composition, we introduced the notation $C(n)$ for its complexity. A naive algorithm implies $C(n) \in O(nM(n))$. The first improvement to this bound was given by Brent and Kung in [BK78]: they devised a baby step-giant step algorithm of complexity $O\left(\sqrt{n}M(n) + n^{(\omega+1)/2}\right)$; in the same paper they also gave an algorithm of complexity $O\left(\sqrt{n \log n}M(n)\right)$ for composition of power series. Bernstein [Ber98] found the bound $O(M(n) \log n)$ for the composition of power series in case the characteristic of the base ring is small, however for a long time Brent and Kung's

ALGORITHM 2.1: Iterated Frobenius

INPUT : $0 < i < d$, $a \in \mathbb{F}_q[X]/f(x)$, $\Phi_1(X) = X^q \bmod f(x)$.
OUTPUT : $\varphi_q^i(a)$.

```
1: let i = Σ b_j 2^j be the binary expansion of i;
2: k ← 1;
3: FOR j = ⌊log_2 i⌋ − 1 TO 0 DO
4:    IF b_j = 0 THEN
5:       Φ_2k ← Φ_k ∘ Φ_k mod f;
6:       k ← 2k;
7:    ELSE
8:       Φ_2k ← Φ_k ∘ Φ_k mod f;
9:       Φ_2k+1 ← Φ_2k ∘ Φ_1 mod f;
10:      k ← 2k + 1;
11: return a ∘ Φ_i mod f.
```

algorithm and its variants [HP98, KS98] have stood as the only generic algorithm for modular composition. A major breakthrough has been recently achieved by Kedlaya and Umans [Uma08, KU08], who give an algorithm for modular composition over a finite field $\mathbb{F}_q$ of *binary* complexity $n^{1+o(1)} \log^{1+o(1)} q$, using a reduction to multivariate multipoint evaluation.

Computing iterated Frobenius and pseudotrace. Fast modular composition can be used to compute Frobenius automorphisms and *pseudotraces* in finite fields. This algorithm is due to von zur Gathen and Shoup [vzGS92], who applied it to polynomial factorization. We will repeatedly use it in Chapters 6 and 8.

Consider the field extension $\mathbb{F}_{q^d}/\mathbb{F}_q$, its Galois group is generated by the Frobenius automorphism

$$\begin{aligned} \varphi_q : \mathbb{F}_{q^d} &\to \mathbb{F}_{q^d}, \\ x &\mapsto x^q. \end{aligned} \tag{2.15}$$

For any $n < d$, we also define the n-th *pseudotrace*[1] as

$$\begin{aligned} T_n : \mathbb{F}_{q^d} &\to \mathbb{F}_{q^d}, \\ x &\mapsto \sum_{i=0}^{n-1} x^{q^i}. \end{aligned} \tag{2.16}$$

Notice that, when $n = d$, the pseudotrace coincides with the trace $\mathrm{Tr}_{\mathbb{F}_{q^d}/\mathbb{F}_q}$; in this case one can use much faster algorithms.

We suppose that elements of $\mathbb{F}_{q^d}$ are represented as residue classes in $\mathbb{F}_q[X]/f(X)$ for some irreducible polynomial f, then the Frobenius automorphism can be computed with $O(\log q)$ multiplications in $\mathbb{F}_{q^d}$ plus one modular composition as

$$\Phi_1(X) = X^q \bmod f(X), \tag{2.17}$$

$$\varphi_q(a) = X^q \circ a \bmod f = a \circ X^q \bmod f = a \circ \Phi_1 \bmod f. \tag{2.18}$$

[1] In [vzGS92], this map goes under the name of *trace map*.

Algorithm 2.2: Pseudotrace

Input : $0 < i < d$, $a \in \mathbb{F}_q[X]/f(x)$, $\Phi_1(X) = X^q \bmod f(x)$.
Output : $T_n(a)$.

let $n = \sum b_j 2^j$ be the binary expansion of n;
$\Theta_o \leftarrow 0$, $\Theta_1 \leftarrow a \circ \Phi_1$;
$k = b_0$;
for $j = 1$ to $\lfloor \log_2 n \rfloor$ do
  $\Phi_{2^j} \leftarrow \Phi_{2^{j-1}} \circ \Phi_{2^{j-1}} \bmod f$;
  $\Theta_{2^j} \leftarrow \Theta_{2^{j-1}} + \Theta_{2^{j-1}} \circ \Phi_{2^{j-1}} \bmod f$;
  if $b_j = 1$ then
    $\Theta_{2^j+k} \leftarrow \Theta_{2^j} + \Theta_k \circ \Phi_{2^j} \bmod f$;
    $k \leftarrow 2^j + k$;
return Θ_n.

Iterating i times φ_q can be done with only $O(\log i)$ modular compositions via square-and-multiply as shown in Algorithm 2.1.

Thus the cost of computing the i-th iterated Frobenius is

$$O(C(d) \log i) \tag{2.19}$$

operations in $\mathbb{F}_q$ plus a precomputation costing $O(M(d) \log q)$.

In Algorithm 2.2 we apply the same idea to compute the n-th pseudotrace in $O(C(d) \log n)$ operations; note that we use a dynamic programming technique to keep the complexity into this bound. The key equation is

$$T_{n+m}(a) = T_n(a) + \varphi_q^n(T_m(a)). \tag{2.20}$$

2.2.5 *Interpolation and Chinese remainder algorithm*

Let $\mathbb{K}$ be a field, and let $x_1, \ldots, x_n \in \mathbb{K}$ be distinct points. Let f be the polynomial that vanishes on $x_1, \ldots, x_n$, by the Chinese remainder theorem we know that

$$\mathbb{K}[X]/f(X) \cong \prod_i \mathbb{K}[X]/(X - x_i). \tag{2.21}$$

If a is an element on the left side of the isomorphism (i.e. a polynomial modulo f), moving it to the right is evaluation at the points $x_1, \ldots, x_n$. The inverse operation is interpolation.

We now give two algorithms to perform this change of representation efficiently. We suppose for simplicity that $n = 2^k$. The first step is to construct the *subproduct tree*.

Computing the subproduct tree takes $O(M(n) \log n)$ operations and requires an equivalent storage. Having precomputed it, it is immediate to evaluate a polynomial at the points $x_1, \ldots, x_n$.

If a has degree at most n, computing the multipoint evaluation also takes $O(M(n) \log n)$ operations using a fast Newton iteration for Euclidean division.

For the inverse operation, we use the *Lagrange interpolants*

$$s_i = \prod_{j \neq i} \frac{(X - x_j)}{x_i - x_j}. \tag{2.22}$$

Algorithm 2.3: Subproduct tree

Input : $n = 2^k, x_1, \ldots, x_n \in \mathbb{K}$.
Output : The *subproduct tree*.
1: let $f_i^{(k)} = (X - x_i)$ for $0 < i \leqslant 2^k$;
2: **for** $j = k - 1$ **to** 0 **do**
3: **for all** $i \in [1, \ldots, 2^j]$ **do**
4: $f_i^{(j)} \leftarrow f_i^{(j+1)} f_{2i}^{(j+1)}$.

Algorithm 2.4: Multipoint evaluation

Input : The subproduct tree, $a \in \mathbb{K}[X]/f(X)$.
Output : $a(x_1), \ldots, a(x_n)$.
1: **for** $j = 1$ **to** k **do**
2: **for all** $i \in [1, \ldots, 2^{j-1}]$ **do**
3: $a_i^{(j)} \leftarrow a_i^{(j+1)} \mod f_i^{j+1}$;
4: $a_{2i}^{(j)} \leftarrow a_i^{(j+1)} \mod f_{2i}^{j+1}$;
5: return $a_1^{(k)}, \ldots, a_n^{(k)}$.

They have the property that

$$\begin{aligned} s_i &\equiv 0 \mod (X - x_j) \quad \text{if } i \neq j, \\ s_i &\equiv 1 \mod (X - x_i); \end{aligned} \tag{2.23}$$

so that

$$a \equiv \sum_i a(x_i) s_i \mod f. \tag{2.24}$$

The key observation is that

$$f' = \sum_i \prod_{j \neq i} (X - x_j), \tag{2.25}$$

hence

$$s_i = \frac{\prod_{j \neq i}(X - x_j)}{f'(x_i)}. \tag{2.26}$$

To interpolate the polynomial a from the values $a(x_i)$, we start by computing the values $f'(x_1), \ldots, f'(x_n)$ by the previous algorithm. Then we reconstruct a using the subproduct tree.

Thus, interpolation too can be computed with $O(M(n) \log n)$ operations in $\mathbb{K}$. These algorithms can be generalized to compute the Chinese remainder isomorphism and its inverse for arbitrary moduli $f_1, \ldots, f_r \in \mathbb{K}[X]$ such that $\gcd(f_i, f_j) = 1$ for $i \neq j$. The complexity is again $O(M(n) \log n)$, where n is the sum of the degrees of $f_1, \ldots, f_n$. See [vzGG99, §10] for details.

2.2.6 *Euclidean algorithm, Cauchy interpolation and rational fraction reconstruction*

Let $\mathbb{K}$ be a field, given two polynomials $f, g \in \mathbb{K}[X]$ of degrees m, n, the Euclidean algorithm permits us to compute their GCD using $O(mn)$ operations in $\mathbb{K}$. Let r

Algorithm 2.5: Interpolation

Input : $a(x_1), \ldots, a(x_n)$, the subproduct tree.
Output : $a = \sum_i a(x_i)s_i$.

```
compute f'(x_1), ..., f'(x_n) using multipoint evaluation;
p_i^(k) ← a(x_i)/f'(x_i) for 0 < i ⩽ 2^i;
for j = k − 1 to 0 do
    for all i ∈ [1, ..., 2^j] do
        p_i^(j) ← p_i^(j+1) t_2i^(j+1) + p_2i^(j+1) t_i^(j+1);
return p_0.
```

Algorithm 2.6: Extended Euclidean algorithm

Input : $f, g \in \mathbb{K}[X]$.
Output : $u, v, r \in \mathbb{K}[X]$ such that $fu + gv = r$.

```
let r_0 ← f, u_0 ← 1, v_0 ← 0;
let r_1 ← f, u_1 ← 0, v_1 ← 1;
i ← 1;
while r_i ≠ 0 do
    compute r_{i−1} = q_i r_i + r_{i+1} by Euclidean division;
    compute u_{i+1} ← u_{i−1} − q_i u_i, v_{i+1} ← v_{i−1} − q_i v_i;
    i ← i + 1;
return u_i, v_i, r_i.
```

be the GCD of f and g, a *Bézout relation* is an equation of the form

$$fu + gv = r, \tag{2.27}$$

with $u, v \in \mathbb{K}[X]$. If we ask $\deg(ur) < \deg(g)$ and $\deg(vr) < \deg(a)$, the Bézout relation is unique; computing it is called the extended GCD problem (XGCD) and can be computed by the *extended Euclidean algorithm*.

One important application of XGCD's is computing modular inverses: let $f, g \in \mathbb{K}[X]$ with $\deg(g) < \deg(f)$ and f prime to g, then r is a unit in $\mathbb{K}$, and a Bézout relation implies

$$g\frac{v}{r} \equiv 1 \mod f. \tag{2.28}$$

More generally, the polynomials computed at each iteration by the extended Euclidean algorithm satisfy

$$fu_i + gv_i = r_i \qquad \text{for any } i; \tag{2.29}$$

each of these is also called a Bézout relation. These relations have two major applications: *Cauchy interpolation* and *rational fraction reconstruction*.

Given n pairs $(x, e) \in \mathbb{K} \times \mathbb{K}$ with all x distinct and an integer $\ell < n$, Cauchy interpolation computes, if it exists, a rational fraction $\frac{r}{v} \in \mathbb{K}(X)$ with $\deg r < \ell$ and $\deg v \leqslant n - \ell$, such that $\frac{r(x)}{v(x)} = e$ for any (x, e). Let $f = \prod(X - x)$, by interpolation one obtains the unique polynomial $g \in \mathbb{K}[X]/f$ such that $g(x) = e$ for any (x, e). Then a Bézout relation for f and g gives

$$\frac{r_i}{v_i} \equiv g \mod f \qquad \text{for any } i, \tag{2.30}$$

thus in particular $\frac{r_i(x)}{v_i(x)} = e$ for any (x, e); this phase often goes under the name of rational fraction reconstruction too. It can be proven that a solution to the Cauchy interpolation problem exists if and only if one of the intermediate results of the extended Euclidean algorithm is such that $\deg(r_i) < \ell$ and $\deg(v_i) \leqslant n - \ell$.

Rational fraction reconstruction (RFR) is very similar to Cauchy interpolation, and it can be viewed as a generalization of it using multiplicities. Let $g \in \mathbb{K}[[X]]$ be a power series, we want to compute a rational fraction $\frac{r}{v} \in \mathbb{K}(X)$ with $\deg(r) < \ell$ and $\deg(v) \leqslant n - \ell$, such that $\frac{r}{v} = g + O(X^n)$ in $\mathbb{K}[[x]]$. Such a rational fraction is called a *Padé approximant* of type $(\ell - 1, n - \ell)$ of g. Again, it can be shown that a Padé approximant of type $(\ell - 1, n - \ell)$ exists if and only if

$$\frac{r_i}{v_i} \equiv g \mod X^{n+m+1} \tag{2.31}$$

is one of the intermediate results computed by the extended Euclidean algorithm.

The extended Euclidean algorithm is not optimal. We address the reader to [vzGG99, §11.1] for the description of an algorithm that takes $f, g \in \mathbb{K}[X]$ of degree at most n and $\ell \leqslant n$, and computes, using $O(M(n)\log n)$ operations, the rows u_i, v_i, r_i and $u_{i+1}, v_{i+1}, r_{i+1}$ of the Extended Euclidean algorithm such that $\deg(r_i) \geqslant n - \ell$ and $\deg(r_{i+1}) < n - \ell$. A consequence of this algorithm is that both Cauchy interpolation and rational fraction reconstruction can be computed in $O(M(n)\log n)$ operations.

2.2.7 Multivariate polynomials

Computing with multivariate polynomials has many complications, compared to the univariate case. Part III will be dedicated to some advanced algorithms for some specific instances of quotient rings of multivariate polynomials. Here, we just recall the basic techniques that permit us to reduce the multivariate to the univariate case.

Multiplication of polynomials in $R[X, Y]$ can be reduced to univariate multiplication by *Kronecker substitution* [Moe76, Kal87, vzGG99, vzGS92, Har09]. If $f, g \in R[X, Y]$ have degree at most m in X and at most n in Y, the product fg can be computed as

$$fg \equiv f(X, X^{2m-1}) \cdot g(X, X^{2m-1}) \mod Y - X^{2m-1}. \tag{2.32}$$

Observing that the reduction modulo $Y - X^{2m-1}$ comes for free, the cost of the previous computation is $O(M(mn))$.

Kronecker substitution also allows to do multiplication in $\big(\mathbb{K}[X]/f(X)\big)[Y]$ by multiplying in $\mathbb{K}[X, Y]$ first and then reducing modulo f. The $O(M(mn))$ operations in $\mathbb{K}$ (m being the degree of f and n being a bound on the degree in Y). The same idea can be applied to multiply elements of $\mathbb{K}[X, Y]/I$, where I is a *triangular ideal*

$$I = \langle f(X), g(X, Y)\rangle \qquad \text{with } g \text{ monic in } Y, \tag{2.33}$$

using $O(M(mn))$ operations, where $m = \deg f$ and $n = \deg_Y g$; the algorithm and the complexity analysis can be found in [PS06, Proposition 4].

All the algorithms presented previously that use multiplication as a black box, can be generalized to the bivariate case using Kronecker substitution. Thus, for example, Euclidean division in $(\mathbb{K}[X]/f(x))[Y]$ can be done in $O(M(mn))$ operations, inversion in $\mathbb{K}[X, Y]/I$ in $O(M(mn)\log mn)$ operations, etc.

Unfortunately, Kronecker substitution does not scale well with the number of variables. For an analysis of the problem and alternatives, see [Sch05, LMMS07]. In general, no quasi-linear time algorithm is known to multiply elements of a finite dimensional algebra $\mathbb{K}[X_1, \ldots, X_n]/I$, even in the case I is triangular.

2.2.8 Transposed algorithms

Finally, we shall recall some known results about *transposed algorithms*. The theory of transposition will be studied in detail in Part II; for the moment we shall just recall that an algorithmic principle, known as the *transposition principle* [Sho94, Sho95, Sho99, Kal00, HQZ04, BLS03], states that

> Let $\mathcal{P}$ be an arbitrary set. To any R-algebraic algorithm A computing a family of linear functions $(f_p : M \to N)_{p\in\mathcal{P}}$ corresponds an R-algebraic algorithm A^* computing the *dual family* $(f_p^* : N^* \to M^*)_{p\in\mathcal{P}}$. The algebraic time and space complexities of A^* are bounded by the time complexity of A.

In Part II we shall see that this principle has an algorithmic content and that the *dual* algorithm can be inferred automatically. For the moment, we are just interested in its existential aspect. We consider two problems for which the transpose problem has been studied.

Transposed multiplication. The first one is polynomial multiplication. Let $a \in R[X]$, the map $M_a : b \mapsto ab$ is a linear map. As usual, we identify the dual module of $R[X]$ to $R[[1/X]]$ via the bilinear form

$$\langle \alpha | b \rangle = [\alpha b]_0, \tag{2.34}$$

where $\alpha \in R[[1/X]]$, $b \in R[X]$ and $[\beta]_i$ is the coefficient of X^i in β.

Then, the dual map to M_a is given by

$$\langle \alpha | M_a | b \rangle = \langle \alpha | ab \rangle = [\alpha ab]_0 = \langle a \cdot \alpha | b \rangle = \langle M_a^*(\alpha) | b \rangle, \tag{2.35}$$

where the product $a \cdot \alpha$ is defined as the usual product, with coefficients of X^i discarded if $i > 0$.

In particular, let a have degree m. If we restrict M_a to $R[X]_n$ (the polynomials of degree at most n), the image of M_a is in $R[X]_{m+n}$. Then, we identify $(R[X]_{m+n})^*$ to $R[1/X]_{m+n}$ and $(R[X]_n)^*$ to $R[1/X]_n$.

Hence, the map

$$\begin{aligned} M_a^* : R[1/X]_{m+n} &\to R[1/X]_n, \\ \alpha &\mapsto a \cdot \alpha \end{aligned} \tag{2.36}$$

is defined by truncating the power series at $1/X^n$:

$$a \cdot \alpha = \sum_{i=0}^{m} a_i X^i \cdot \sum_{j=0}^{m+n} \frac{\alpha_j}{X^j} = \sum_{k=0}^{n} \sum_{i-j=k} \frac{a_i \alpha_j}{X^k}. \tag{2.37}$$

Observe that the coefficients of $a \cdot \alpha$ are the same as the coefficients of $a(\alpha X^{m+n})$ between X^m and X^{m+n}, thus any algorithm for polynomial multiplication can

be used to compute $a \cdot \alpha$ in $M(2m+n)$ operations. This is the reason why *transposed polynomial multiplication* is also called *middle product* in the literature [BLS03, HQZ04]. The generalization to the multivariate case is straightforward.

Observe, however, that this algorithm has nothing to do with the transposition principle: it is just a property of multiplication. Applying the transposition principle, one obtains a tighter bound of $M(\max(m,n))$. The univariate case is treated in [HQZ04, BLS03], it is applied to speed up some Newton iterations on power series; the bivariate case appears in the proof of [PS06, Corollary 2].

Transposed Euclidean division. We now study the dual of Euclidean division with remainder; it is actually only the remainder that we are interested in. Let f be a monic polynomial in $R[X]$, the map

$$\begin{aligned} \mathrm{mod}_f : R[X] &\to R[X]_n, \\ a &\mapsto a \bmod f \end{aligned} \tag{2.38}$$

is a linear map.

Suppose f has degree $n+1$, then the dual map

$$\mathrm{mod}_f^* : R[1/X]_n \to R[[1/X]] \tag{2.39}$$

is such that, for any $\alpha \in \mathbb{R}[X]_n$ and any $i \geqslant 0$,

$$\left\langle \beta \middle| X^i f \right\rangle \stackrel{\text{def}}{=} \left\langle \mathrm{mod}_f^*(\alpha) \middle| X^i f \right\rangle = \left\langle \alpha \middle| X^i f \bmod f \right\rangle = 0, \tag{2.40}$$

where we have set $\beta \stackrel{\text{def}}{=} \mathrm{mod}_f^*(\alpha)$.

Using what we saw previously on transposed multiplication,

$$\left\langle X^i \cdot \beta \middle| f \right\rangle = \left\langle \beta \middle| X^i f \right\rangle = 0 \quad \text{for any } i \geqslant 0. \tag{2.41}$$

Equivalently, the coefficients of β satisfy a linear recurrence with minimal polynomial f. If

$$\alpha = \sum_{i=0}^{n} \frac{\alpha_i}{X^i}, \qquad \beta = \sum_{i \geqslant 0} \frac{\beta_i}{X^i} \tag{2.42}$$

the initial conditions for β are

$$\beta_i = \left\langle \beta \middle| X^i \right\rangle = \left\langle \alpha \middle| X^i \bmod f \right\rangle = \left\langle \alpha \middle| X^i \right\rangle = \alpha_i \tag{2.43}$$

for any $i \leqslant n$.

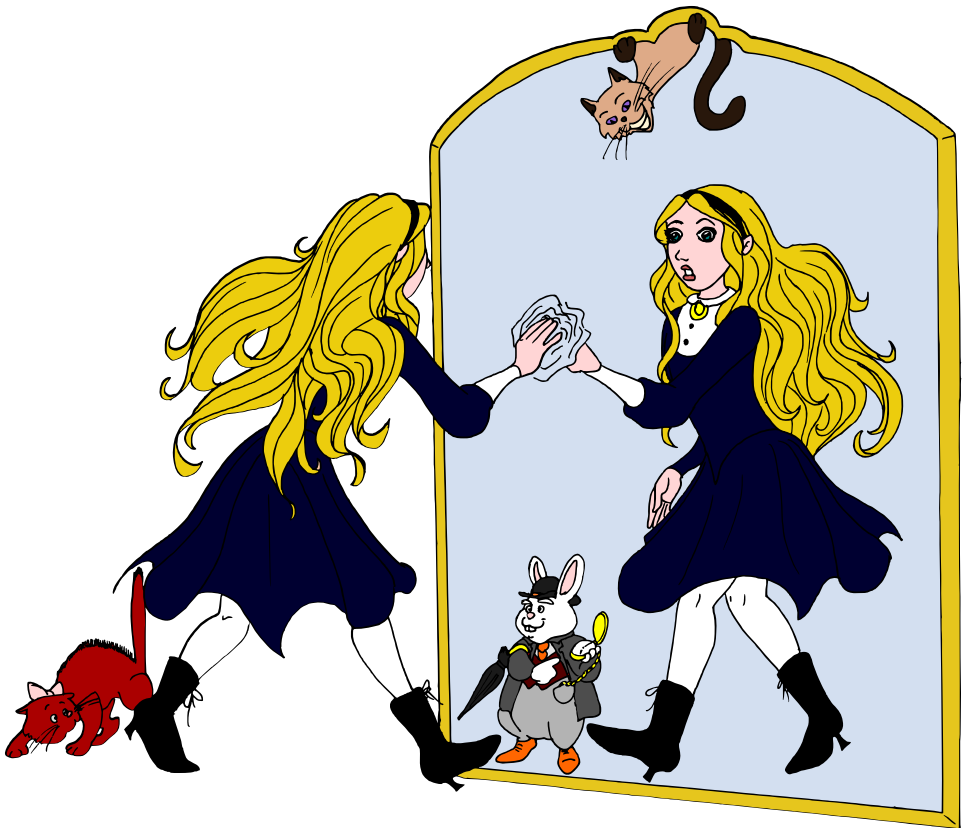
Thus, the dual of modular reduction consists in extending a linear recurring sequence of order n from its first n elements. Any algorithm for such task (for example, [Sho99, §3]) can be used to compute mod_f^*, however one can also directly obtain one such algorithm by applying the transposition principle to any algorithm for Euclidean division. In any case, *transposed modular reduction* to the order N can be computed using $O(M(N))$ operations[2].

Finally, the composition $\mathrm{mod}_f \circ M_a$ gives an algorithm for multiplication in the ring $R[X]/f(X)$. Hence, the dual to this is simply $M_a^* \circ \mathrm{mod}_f^*$. This is called *transposed modular multiplication* and we shall use this repeatedly in the next chapters. The generalization of this to the case of $\mathbb{K}[X,Y]/I$, where I is a triangular ideal, is straightforward but technical; we address to [PS06, Corollary 2] for the details.

[2] One can do better when N is much larger than n, but we shall not need such an improvement in this document.

Part II

The transposition principle





Chapter 3

Algebraic Complexity and duality

The complexity of algebraic algorithms is often more easily described in a non-Turing model where one assumes that any algebraic operation can be done in a unit of time and any other operation is free. *Algebraic complexity* studies precisely the computational models that behave this way.

For algorithms over finite rings, the algebraic complexity gives a precise estimate for the complexity in the Turing or RAM model (also called *binary models*). For other rings, the algebraic estimate may be way off target, but it can nevertheless give useful information.

In this chapter we study models that allow one to study the algebraic complexity of linear operators. We first present the *arithmetic circuit*, then the *straight line program*. Because of their algebraic structure, these models support some algebraic manipulations. Our principal interest will be the *transposition theorem*, stating that it is possible to apply classical duality (in the sense of Section 1.1.3) to programs, while preserving some complexity invariants. The interest for the transposition theorem comes from the applications we have seen in Section 2.2.8 and other more advanced applications that we will see in the next chapters.

Finally, in Section 3.4, we study the relationship between the transposition theorem and the classical theory of *automatic differentiation*.

3.1 Arithmetic circuits

In this section we briefly present the arithmetic circuit model. Since we have in mind applications to the theory of transposition, our presentation slightly deviates from textbooks; for a more classical and extensive treatment see [BCS97b, Vol99].

3.1.1 Basic definitions

In the whole chapter, by R we shall denote a (non necessarily commutative) ring with unit. Unless otherwise stated, we consider R^n with its natural structure of left R-module; when needed, we shall use kets $|x\rangle$ to remove any ambiguity about the fact that we are talking about elements of a left module. We set $R^0 = 0$, the zero module, and we denote by $\perp$ (or $|\perp\rangle$) its unique element.

The dual space $(R^n)^* = \hom(R^n, R)$ has a natural structure of right R-module (equivalently, of left R^{op}-module) by the assignment

$$\langle \ell |\, a : |x\rangle \mapsto \langle \ell | x\rangle\, a, \tag{3.1}$$

where we have used bras to denote elements of $(R^n)^*$ and a bracket to denote the natural bilinear form that results by applying linear forms to elements.

Definition 3.1 (Arithmetic operator, arity) An *arithmetic operator* over R is a morphism of left modules $f : R^i \to R^o$ for some $i, o \in \mathbb{N}$. Here i is called the *in-arity* of f or simply *arity*, o is called the *out-arity* of f.

Definition 3.2 (Arithmetic basis) An *arithmetic* R*-basis* is a (not necessarily finite) set of arithmetic operators over R. A basis is said to be *commutative* if all its operators are invariant under the natural action of $\mathfrak{S}_n$ over R^n (i.e. under permutation of coordinates).

The arithmetic basis we will work with is the *standard left-linear basis*, denoted by $\mathcal{L}$. It is composed of

$$\begin{array}{lll} +: R \oplus R \to R, & *_a : R \to R, & \& : R \to R \oplus R, \\ \quad (a,b) \mapsto a+b, & \quad b \mapsto ba, & \quad a \mapsto (a,a), \\ & & \\ 0 : 0 \to R, & & \omega : R \to 0, \\ \quad \bot \mapsto 0, & & \quad a \mapsto \bot\,. \end{array} \tag{$\mathcal{L}$}$$

Arithmetic circuits are directed acyclic multigraphs carrying information from an arithmetic basis; the formal definition follows.

Definition 3.3 (Arithmetic node) Let $\mathcal{B}$ be an R-basis. A *node* over $(R, \mathcal{B})$ is a tuple $\nu = (I, O, f)$ such that

- I and O are finite ordered sets,
- f is either an element of $\mathcal{B}$ or the special value $\emptyset$.
- If $f = \emptyset$, one of the two following conditions must hold:
  - I is a singleton and O is empty, in this case we say that ν is an *input node*;
  - I is empty and O is a singleton, in this case we say that ν is an *output node*.
- If $f \neq \emptyset$, the cardinality of I matches the in-arity of f and the cardinality of O matches the out-arity of f; in this case we say that ν is an *evaluation node*.

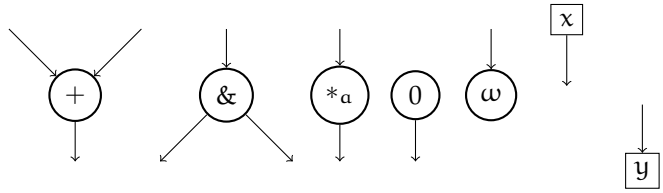


Figure 3.1: Nodes over the standard linear basis: round ones are evaluation nodes, square ones are input and output nodes.

We call *input ports* the elements of I and *output ports* the elements of O, which we denote respectively by $\mathrm{in}(\nu)$ and $\mathrm{out}(\nu)$. The cardinalities of I and O are called, respectively, the *in-degree* and *out-degree* of ν. We call f the *value* of ν and write $\beta(\nu)$ for it.

Nodes over the linear basis are pictured in Figure 3.1.1. We do not explicitly represent the orderings on $\mathrm{in}(+)$ and $\mathrm{out}(\&)$ because they are not relevant for the linear basis: in fact, all operators are commutative.

Definition 3.4 (Arithmetic circuit) Let $\mathcal{B}$ be an R-basis. A (linear) *arithmetic circuit* over $(R, \mathcal{B})$ is a tuple $C = (V, E, \leqslant, \leqslant_i, \leqslant_o)$ such that

1. V is a finite set of nodes over $(R, \mathcal{B})$;
2. $<$ is a total order on V, $<_i$ is a total order on the input nodes in V, $<_o$ is a total order on the output nodes in V;
3. let $I = \biguplus_{v \in V} \mathrm{in}(v)$ and $O = \biguplus_{v \in V} \mathrm{out}(v)$, then E is a bijection from O to I such that $E(o) = i$ implies that $o \in \mathrm{out}(v)$, $i \in \mathrm{in}(v')$ and $v \lesssim v'$.

In practice, the definition says that C is a directed acyclic multigraph (also called *multiDAG*), where V are the vertices, E the edges, and where the degrees of each vertex are prescribed by the arities of the underlying arithmetic node. Moreover, we add an ordering on input nodes (vertices of in-degree 0) and on output nodes (vertices of out-degree 0).

In what follows, we shall call (V, E) the *underlying graph* of C, and use classic graph theoretic terms to refer to its properties. We shall often implicitly make this identification. Thus, we shall represent E as a set of edges (o, i), and make use of the classic concepts of *incident* edge, nodes *connected* by an edge, *paths*, etc.

Figure 3.2 shows two examples of arithmetic circuits. Input and output nodes are ordered from left to right; ports are not ordered because the basis is commutative.

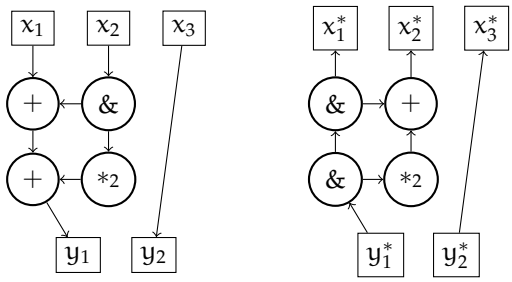


Figure 3.2: Two arithmetic circuits over $\mathcal{L}$. The linear map $y_1 = x_1 + 3x_2, y_2 = x_3$ is computed by the circuit on the left and its dual is computed by the circuit on the right.

Definition 3.5 (Size, depth) Let C be a circuit over $(R, \mathcal{B})$. The *size* of C, denoted by $\mathrm{size}(C)$ is the number of evaluation nodes in V; the *depth* of C, denoted by $\mathrm{depth}(C)$ is the length of the longest directed path –in a graph-theoretic sense– in (V, E).

Sometimes it is useful to only count certain nodes. Let $X \subset \mathcal{B}$, the X-weighted size of C, denoted by $\mathrm{size}_X(C)$ is the number of nodes $v \in V$ such that $\beta(v) \in X$.

3.1.2 Semantic of a circuit

Circuits would be meaningless if they had no semantic attached to them. Intuitively the semantic corresponds to recursively feed inputs to the nodes, evaluate $\beta(v)$ at the inputs and collect the outputs.

Definition 3.6 (Evaluation of an arithmetic circuit) Let C be an arithmetic circuit with i inputs and o outputs, then its evaluation is a morphism $\mathrm{eval}_C : R^i \to R^o$.

In order to define it, we simultaneously define the evaluation eval_v of each $v \in V$ and the evaluation eval_e of each $e \in E$. We will denote by $<_v$ the orders on the input and the output ports of v.

- Let $v \in V$ have out-degree n, let its evaluation be $\mathrm{eval}_v : R^i \to R^n$ and let $\pi_1, \dots, \pi_n$ be the canonical projections from R^n to R. Let $o_1 <_v \cdots <_v o_n$ be the output ports of v and let $e_j = \big(o_j, E(o_j)\big)$ be the corresponding edges stemming from v, then $\mathrm{eval}_{e_j} = \pi_j \circ \mathrm{eval}_v$ for any j.
- Let $x_1 <_i \cdots <_i x_i$ be the input nodes and let $\pi_1, \dots, \pi_i$ be the canonical projections from R^i to R, then $\mathrm{eval}_{x_j} = \pi_j$ for any j.
- For every evaluation node v with in-degree m, let $i_1 <_v \cdots <_v i_m$ be the input ports of v and let $e_j = \big(E^{-1}(i_j), i_j\big)$ be the corresponding edges incident to v, then
$$\mathrm{eval}_v = \beta(v) \circ (\mathrm{eval}_{e_1}, \cdots, \mathrm{eval}_{e_m}). \tag{3.2}$$
- For every output node y, let $e \in E$ be the only edge incident to y, then $\mathrm{eval}_y = \mathrm{eval}_e$.

We can finally define $\mathrm{eval}_C : R^i \to R^o$. Let $y_1 <_o \cdots <_o y_o$ be the output nodes, then
$$\mathrm{eval}_C = (\mathrm{eval}_{y_1}, \cdots, \mathrm{eval}_{y_o}). \tag{3.3}$$
We also say that C *computes* eval_C.

It is immediate to verify that eval_C is a morphism of left modules, because we only used compositions and direct sums to define it. The converse is partially true.

Proposition 3.7 *Any morphism of free finite-dimensional R-modules can be computed by an arithmetic circuit over $(R, \mathcal{L})$.*

Proof. Take the matrix associated to such morphism and create a circuit that performs the matrix-vector product.

This also gives an upper bound of $O(mn)$ on the circuit size of a linear operator $R^m \to R^n$.

We now define a way of substituting nodes, first syntactically, then semantically.

Definition 3.8 (Syntactic substitution) Let $C = (V, E, \leqslant, \leqslant_i, \leqslant_o)$ be a circuit over $(R, \mathcal{B})$ and let $C' = (V', E', \leqslant', \leqslant'_i, \leqslant'_o)$ be a circuit over $(R, \mathcal{B}')$. Let C' have i inputs and o outputs and let $v \in V$ have in-degree i and out-degree o.

Let $\mathcal{I}$ and $\mathcal{O}$ be monotone bijections respectively from $\mathrm{in}(v)$ to $\mathrm{in}(C')$ and from $\mathrm{out}(C')$ to $\mathrm{out}(v)$. We denote by $C[C'/v]$ the circuit $(V'', E'', \leqslant'', \leqslant''_i, \leqslant''_o)$ over $(R, \mathcal{B} \cup \mathcal{B}')$ defined as follows:

- $V'' = V \uplus (V' - \mathrm{in}(C') - \mathrm{out}(C'))$;
- $\leqslant_i''=\leqslant_i$, $\leqslant_o''=\leqslant_o$;
- $v' \leqslant v''$ if and only if one of the following conditions hold:
  - $v', v'' \in V$ and $v' \leqslant v''$;
  - $v', v'' \in V'$ and $v' \leqslant' v''$;
  - $v' \in V$ and $v'' \in V'$ and $v' \leqslant v$;
  - $v' \in V'$ and $v'' \in V$ and $v \leqslant v''$;
- $E''(o) = \begin{cases} E'(o') & \text{if } E(o) \in \mathrm{in}(v) \text{ and } \mathcal{I}(E(o)) = v' \text{ and } \mathrm{out}(v') = \{o'\}, \\ E(o') & \text{if } E'(o) \in \mathrm{out}(C') \text{ and } \mathcal{O}(E'(o)) = o', \\ (E \uplus E')(o) & \text{otherwise.} \end{cases}$

Definition 3.9 (Semantic substitution) Let C be a circuit over $(R, \mathcal{B} \cup \{f\})$ and let F be a circuit over $(R, \mathcal{B})$ such that $\mathrm{eval}_F = f$.

We denote by $C[F/f]$ the circuit over $(R, \mathcal{B})$ where any node v of C such that $\beta(v) = f$ has been syntactically substituted by F.

The proof of the following proposition is immediate.

Proposition 3.10 *Under the conditions of the previous definition,*

$$\mathrm{eval}_{C[F/f]} = \mathrm{eval}_C\,.$$

As a shorthand notation, we will draw octogones to signify that a node has been syntactically substituted by a circuit, without giving the actual shape of the substituting circuit. Figure 3.1.2 shows an example.

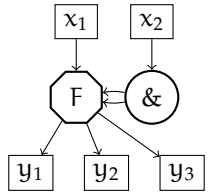


Figure 3.3: Arithmetic circuit where a node has been syntactically substituted by a circuit F with 3 inputs and 3 outputs.

3.1.3 The transposition theorem

We are now ready to state and prove the *transposition theorem* for arithmetic circuits.

We fix a family $(M_i)_{i\in\mathbb{N}}$ of free left R-modules, with $M_i \cong R^i$. Via the isomorphisms, it is straightforward to extend the definition of arithmetic circuit so that eval_c is a morphism $M_i \to M_o$. We also fix a family $(N_i)_{i\in\mathbb{N}}$ of free right R-modules, and a family of non-degenerate bilinear forms

$$(\langle|\rangle_i : N_i \times M_i \to R)_{i\in\mathbb{N}}\,. \tag{3.4}$$

One can think of M_n being R^n, N_n being $(R^n)^*$, and the bilinear forms being the natural ones.

We define the notion of dual circuit, that intuitively corresponds to *turn all the arrows around*.

Definition 3.11 (Dual arithmetic basis) Let $\mathcal{B}$ be an arithmetic basis over R. The *dual basis* $\mathcal{B}^*$ (with respect to $\langle|\rangle_i$) is the basis over R^{op}

$$\mathcal{B}^* = \{f^* \mid f \in \mathcal{B}\}. \tag{3.5}$$

In particular, the dual basis to $(R, \mathcal{L})$ is

$$\begin{aligned} + = \&^* : (R \oplus R)^* \to R^*, \qquad & \& = +^* : R^* \to (R \oplus R)^*, \\ \langle a, b| \mapsto \langle a| + \langle b|, \qquad & \langle a| \mapsto \langle a, a|, \end{aligned}$$

$$\begin{aligned} *_a = (*_a)^* : R^* \to R^*, \\ \langle b| \mapsto \langle b|\, a, \end{aligned} \tag{$\mathcal{L}^*$}$$

$$\begin{aligned} 0 = \omega^* : 0^* \to R^*, \qquad & \omega = 0^* : R^* \to 0^*, \\ \langle \bot| \mapsto \langle 0|, \qquad & \langle a| \mapsto \langle \bot|. \end{aligned}$$

Definition 3.12 (Dual circuit) Let $C = (V, E, \leqslant, \leqslant_i, \leqslant_o)$ be a circuit over $(R, \mathcal{B})$. For any $v \in V$ define

$$v^* = \begin{cases} (O, I, f^*) & \text{if } v = (I, O, f) \text{ with } f \neq \emptyset, \\ (O, \emptyset, \emptyset) & \text{if } v = (\emptyset, O, \emptyset), \\ (\emptyset, I, \emptyset) & \text{if } v = (I, \emptyset, \emptyset). \end{cases} \tag{3.6}$$

The dual circuit (with respect to $\langle|\rangle_i$) of C, denoted by C^*, is the arithmetic circuit over $(R^{op}, \mathcal{B}^*)$

$$C^* = (V^*, E^{-1}, \leqslant', \leqslant_i', \leqslant_o'),$$

where $V^* = \{v^* | v \in V\}$ and the orderings are defined as follows:

$$v \leqslant v' \;\Leftrightarrow\; v'^* \leqslant' v^*, \tag{3.7}$$

$$v \leqslant_o v' \;\Leftrightarrow\; v^* \leqslant_o' v'^*, \tag{3.8}$$

$$v \leqslant_i v' \;\Leftrightarrow\; v^* \leqslant_i' v'^*. \tag{3.9}$$

In particular, this makes (V^*, E^{-1}) the reverse graph of (V, E) in a graph-theoretic sense. Figure 3.2 shows two circuits that are each other's dual. We can now state the transposition theorem.

Theorem 3.13 (Transposition theorem) *Let C be a circuit that computes a morphism f, then C^* computes the dual morphism f^*.*

In order to maintain this chapter at the level of elementary linear algebra, we give here a quite tedious proof: it amounts, in a hidden way, to write down the matrices of f and f^* and check that they are the same. In Appendix A we will give a conceptually simpler proof, using category theory.

Definition 3.14 (Evaluation of a path) Let x and y be nodes of C and let $p = (e_1, \ldots, e_k)$ be a path from x to y.

If $k = 1$, we define eval_p as the identity map $R \to R$. If $k > 1$, for any $1 \leqslant i < k$ let the node

$$\cdots \xrightarrow{e_i} v \xrightarrow{e_{i+1}} \cdots$$

have n_i inputs and m_i outputs, and let $\iota : R \to R^{n_i}$ be the injection corresponding to the position of e_i in $\mathrm{in}(v)$ and $\pi : R^{m_i} \to R$ the projection corresponding to the position of e_{i+1} in $\mathrm{out}(v)$, then we define

$$f_i = \pi \circ \beta(v) \circ \iota. \tag{3.10}$$

Finally, we define eval_p as

$$f_{k-1} \circ \cdots \circ f_1. \tag{3.11}$$

Lemma 3.15 (The electrical network lemma) *Let C be an arithmetic circuit, let $x_1 \leqslant_i \cdots \leqslant_i x_n$ be its input nodes and $y_1 \leqslant_o \cdots \leqslant_o y_m$ its output nodes. We have the following identity*

$$\pi_j \circ \mathrm{eval}_C \circ \iota_i = \sum_{p \in x_i \rightsquigarrow y_j} \mathrm{eval}_p \quad \textit{for any } 1 \leqslant i \leqslant n,\ 1 \leqslant j \leqslant m, \tag{3.12}$$

where the sum ranges over all the distinct paths from x_i to y_j.

Proof. We start by proving that for any node v and any edge $v \xrightarrow{e} v'$

$$\mathrm{eval}_e = \sum_{i=1}^{n} \sum_{p \in x_i \rightsquigarrow v \xrightarrow{e} v'} \mathrm{eval}_p \circ \pi_i, \tag{3.13}$$

where $\pi_i : R^n \to R$ is the i-th projection, and the second sum ranges over all the distinct paths from x_i to v' passing through e. We do this by induction on the length of the longest path to v.

If the longest path to v has length 0, then v is one of the input nodes, say x_i. Then by definition $\mathrm{eval}_e = \pi_i$, and Eq. (3.13) is verified.

Then, let v be an evaluation node, let $e_1 \leqslant_v \ldots \leqslant_v e_k$ be the edges incident to v, and let $v_1, \ldots, v_k$ be the corresponding nodes. Then, by definition

$$\mathrm{eval}_e = \pi_e \circ \beta(v) \circ (\mathrm{eval}_{e_1}, \ldots, \mathrm{eval}_{e_k}) = \sum_{j=1}^{k} \pi_e \circ \beta(v) \circ \iota_j \circ \mathrm{eval}_{e_j}, \tag{3.14}$$

where π_e is the projection corresponding to the position of e in $\mathrm{out}(v)$. By induction, this is equivalent to

$$\sum_{j=1}^{k} \sum_{i=1}^{n} \sum_{p \in x_i \rightsquigarrow v_j \xrightarrow{e_j} v} \pi_e \circ \beta(v) \circ \iota_j \circ \mathrm{eval}_p \circ \pi_i = \sum_{i=1}^{n} \sum_{p \in x_i \rightsquigarrow v \xrightarrow{e} v'} \mathrm{eval}_p \circ \pi_i, \tag{3.15}$$

where the equality comes from Eq. (3.10).

Now, by the definition of eval_C we have

$$\pi_j \circ \mathrm{eval}_C = \mathrm{eval}_{y_j} = \sum_{i=1} \sum_{p \in x_i \rightsquigarrow y_j} \mathrm{eval}_p \circ \pi_i, \tag{3.16}$$

and composing on both sides with ι_i gives Eq. (3.12). ∎

Proof of the transposition theorem. The proof of the transposition theorem is now straightforward. By linearity it is enough to prove that

$$\pi_j \circ \mathrm{eval}_C \circ \iota_i = (\pi_i \circ \mathrm{eval}_{C^*} \circ \iota_j)^* \quad \text{for any } 1 \leqslant i \leqslant n,\ 1 \leqslant j \leqslant m, \tag{3.17}$$

but this is evident by Eqs. (3.12), (3.11) and (3.10). 

Corollary 3.16 *A linear function $f : R^n \to R^m$ and its transpose can be computed by arithmetic circuits on $(R, \mathcal{L})$ of same sizes and depths. In particular if C computes f and C^* computes f^*,*

$$\begin{aligned} \mathrm{size}_{\{+\}}(C) = \mathrm{size}_{\{\&\}}(C^*), &\qquad \mathrm{size}_{\{\&\}}(C) = \mathrm{size}_{\{+\}}(C^*), \\ \mathrm{size}_{\{*_a\}}(C) = \mathrm{size}_{\{*_a\}}(C^*) &\quad \textit{for any } a \in R, \\ \mathrm{size}_{\{0\}}(C) = \mathrm{size}_{\{\omega\}}(C^*), &\qquad \mathrm{size}_{\{\omega\}}(C) = \mathrm{size}_{\{0\}}(C^*). \end{aligned}$$

Remark 3.17. The name "transposition theorem" is somehow misleading. In fact, the theorem says that if eval_C is the map $x \mapsto xM$ for some matrix M, then eval_{C^*} is the map $y \mapsto My$. If R is commutative, this is equivalent to transpose M, but in the non-commutative case this is not true anymore. For this reason, we shall always prefer the formulation in terms of bilinear forms instead of the one in terms of matrices.

It is also worth noticing that the transposition theorem stays true if instead of a bilinear form we had used a sesquilinear form (in this case $(*_a)^* = *_{\bar{a}}$).

Note. The name "The electrical network lemma" is ours, but it could well have been original. The transposition theorem was first discovered in electrical network theory by Bordewijk [Bor57]; he only showed the case $R = \mathbb{C}$. Some attribute the discovery to Tellegen [BCS97b, BLS03], Bordweijk's advisor, but this is debated [Ber].

The first complete algebraic proof, treating the case of an arbitrary non-commutative ring, is due to Fiduccia [Fid73]. There have been many rediscoveries of the theorem (see [Ber]), but Fiduccia's statement stays the most general.

Canny, Kaltofen and Yagati [CKY89, Kal00] pointed out that the transposition theorem can be proven as a special case of Baur and Strassen's differentiation of circuits [BS83]. We shall come back to this question in Section 3.4.

3.1.4 Circuit families

A circuit is limited to compute one specific function with inputs and outputs of fixed size (in terms of elements of R). However complexity theory is interested in algorithms that compute on inputs of variable size. This leads to study families of circuits.

Definition 3.18 (Circuit family) Let $\mathcal{B}$ a basis over R and $\mathcal{P}$ a set. A *circuit family* over $(R, \mathcal{B}, \mathcal{P})$ is a family of circuits over $(R, \mathcal{B})$ indexed by $\mathcal{P}$. $\mathcal{P}$ is called the *parameter space* of the family. When the mapping from $\mathcal{P}$ to the circuits is Turing-computable, the family is called *uniform*.

Algebraic complexity textbooks usually take $\mathcal{P} = \mathbb{N}$ and force C_n to have n inputs. Our construction is more general and lacks some of the interesting complexity theoretic properties of circuit families, but its interest will be clear in the next sections.

Definition 3.19 (Size and depth functions) Let $\mathcal{C} = (C_j)_{j\in\mathcal{P}}$ be a circuit family, we define the size and depth function as

$$\begin{aligned} \mathrm{size}^{\mathcal{C}} : \mathcal{P} &\to \mathbb{N} & \qquad \mathrm{depth}^{\mathcal{C}} : \mathcal{P} &\to \mathbb{N} \\ j &\mapsto \mathrm{size}(C_j) & j &\mapsto \mathrm{depth}(C_j) \end{aligned}$$

respectively.

As in definition 3.5, for $X \subset \mathcal{B}$ we also define

$$\begin{aligned} \mathrm{size}^{\mathcal{C}}_X : \mathcal{P} &\to \mathbb{N} \\ j &\mapsto \mathrm{size}_X(C_j)\ . \end{aligned}$$

We are mainly interested in uniform circuit families since they are equivalent to computable functions, the transposition theorem easily generalizes to them. We will not study uniform circuit families more in depth; what we shall do instead, is directly work on computer programs implicitly representing circuit families and automatically deduce the transposed family without actually using the circuit model. More details on uniform circuit families can be found in [Vol99].

3.2 Multilinearity

In this section we develop an extension of the transposition theorem to the multilinear case. The subject has already been treated by Hopcroft and Musinski [HM73] and Fiduccia [Fid73]; this section restates their results in terms of arithmetic circuits[1] and generalizes them to the case where a path may contain more than one multiplication by an element of a non-commutative ring.

3.2.1 Multilinear circuits

We shall consider *multilinear circuits*, i.e. circuits that, besides the operators of $\mathcal{L}$, also contain binary multiplication nodes. The definitions of the previous section must be adapted to deal with operators that are not module morphisms, but this generalization is straightforward (see also Appendix A for a clean way of defining arithmetic circuits that supports both linear and arbitrary circuits).

Multilinear circuits are constructed using the *standard multilinear basis* $\mathcal{S}$

$$\begin{aligned} + : R \times R &\to R, & * : R \times R &\to R, & \& : R &\to R \times R, \\ (a,b) &\mapsto a+b, & (a,b) &\mapsto ab, & a &\mapsto (a,a), \\ \eta_a : \{\bot\} &\to R, & & & \omega : R &\to \{\bot\}, \\ \bot &\mapsto a, & & & a &\mapsto \bot\ . \end{aligned} \tag{$\mathcal{S}$}$$

We are going to define a transformation process that transforms a circuit over $(R, \mathcal{S})$ into a (uniform) circuit family over $(R, \mathcal{L})$; the idea is to make any bilinear multiplication node linear by *fixing* one of its inputs (see Figure 3.4). We call this process *linearization*, a special case of it has been used in [GG05, Ser08] to transpose circuits that compute differentials.

[1] Note that [Fid73] already used a model very close to ours.

Definition 3.20 (Zero edge, null edge) Let C be a circuit over $(R, \mathcal{S})$, a *zero edge* is any edge e in C such that one of the following conditions holds:

- e stems from a node v with $\beta(v) = \eta_0$, such an edge is also called a *normal* zero edge;
- e stems from a node v with $\beta(v) = +$ and whose incident edges are both zero;
- e stems from a node v with $\beta(v) = *$ and such that at least one of the incident edges of v is zero;
- e stems from a node v with $\beta(v) = \&$ and whose input edge is zero.

A *null edge* is any edge e such that one of the following conditions holds:

- e is incident to a node v with $\beta(v) = \omega$, such an edge is also called a *normal* null edge;
- e is incident to a node v with $\beta(v) = \&$ and whose stemming edges are both null;
- e is incident to a node v with $\beta(v) \in \{+, *\}$ such that its stemming edge is null.

An output node whose incident edge is zero is called a *zero output*, an input node whose stemming edge is null is called a *null input*. A *normal* circuit is a circuit whose zero and null edges are all normal.

Notice that the evaluation of a zero edge or output is the zero function, the converse is not true. There is an obvious normalization technique that takes a generic circuit and transforms it in a normal circuit having the same evaluation; clearly, the normalization does not increase the size and the depth of the circuit (it generally increases $\text{size}_{\{\eta_0,\omega\}}$, though). When necessary, we will restrict ourselves to normal circuits.

Definition 3.21 (Linearization) Let $C = (V, E)$ be a circuit over $(R, \mathcal{S})$. Let $0 = \{v \in V | \beta(v) = \eta_0\}$, a linearization of C is a subset $\ell \subset \text{in}(C) \cup 0$ such that:

- for every $v \in V$ with $\beta(v) = +$ either none of its incident edges is reachable from ℓ, or both are;
- for every $v \in V$ with $\beta(v) = *$ at most one of the edges incident to v is reachable from ℓ; if R is non-commutative, such edge is always the right (left) edge and the linearization is called a *left (right) linearization*.

If $\ell = \emptyset$, the linearization is called *trivial*.

The elements of $\ell \cap \text{in}(C)$ and $s = \text{in}(C) - \ell$ are respectively called the *linear* and *scalar* inputs. An edge reachable from ℓ is called *linear*, *scalar* otherwise.

Definition 3.22 (Linearized circuit, scalar part) Let $C = (V, E, \leqslant, \leqslant_i, \leqslant_o)$ be a normal circuit over $(R, \mathcal{S})$ and let ℓ be a left (resp. right) linearization; without loss of generality, we suppose that all the scalar inputs precede the linear inputs in the order $\leqslant_i$ (one can always permute inputs).

Let n be the number of scalars for the linearization ℓ and let $x_1, \dots, x_n$ be distinct indeterminates over R. The *linearized circuit*

$$C_\ell = (V_\ell, E_\ell, \leqslant_\ell, \leqslant_{i,\ell}, \leqslant_{o,\ell})$$

is the circuit over $(R[x_1, \dots, x_n], \mathcal{L})$ (resp. $(R^{op}[x_1, \dots, x_n], \mathcal{L}^*)$) obtained from C as follows:

- E_ℓ is the subset of E containing the linear edges,
- V_ℓ are the nodes of V adjacent to E_ℓ, where $\beta(v)$ has incurred the following substitutions:
  - η_0 becomes 0; $+, \&, \omega$ are preserved;
  - if $\beta(v) = *$, let e be the only non-linear edge incident to v and let $a = \mathrm{eval}_e(x_1, \dots, x_n, \bullet, \dots, \bullet)$, then $*$ becomes $*_a$;
  - The orders $\leqslant_\ell, \leqslant_{i,\ell}, \leqslant_{o,\ell}$ are the restriction to V of the original ones.

The sets $V - V_\ell$ and $E - E_\ell$ are called the *scalar part* of C.

Observe that non-linear edges do not depend on linear inputs, thus the substitution for $*$ is well defined. The trivial linearization gives the trivial linear circuit with no nodes, hence, its evaluation is the trivial map $0 \to 0$. Figure 3.4 shows an example of linearized circuit (in the case R is commutative), we gray out the scalar part of the circuit.

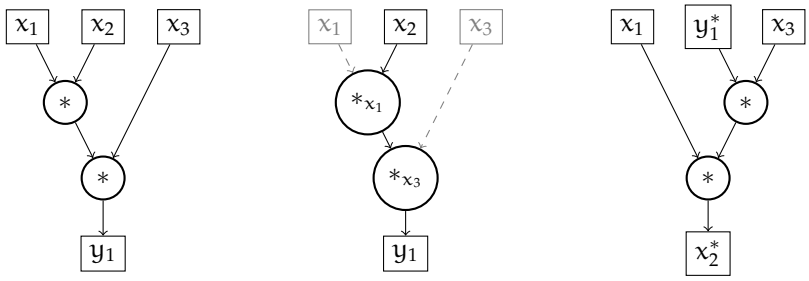


FIGURE 3.4: A circuit over a commutative ring, its linearization for $\ell = \{x_2\}$ (scalar edges are grayed out), and its ℓ-dual.

Any linearized circuit obviously defines an uniform circuit family over $(R, \mathcal{L})$ (resp. $(R^{op}, \mathcal{L}^*)$), thus we can apply the transposition theorem to the family. But there is more: from a linearized circuit we can deduce a new circuit over $(R, \mathcal{S})$ that computes the same function as the transposed family.

DEFINITION 3.23 (ℓ-dual) Let C be a normalized circuit over $(R, \mathcal{S})$ and let ℓ be a linearization. The ℓ-dual of C is the circuit over $(R, \mathcal{S})$ obtained by dualizing the linearized circuit C_ℓ, then connecting back the edges of the scalar part to the corresponding nodes in C_ℓ: in doing this, nodes with $\beta(v) = *_a$ are changed back to $\beta(v) = *$. The order on the nodes of the ℓ-dual is arbitrary.

By abuse of notation, the ℓ-dual will also be denoted by C_ℓ^*. Figure 3.4 shows an example of ℓ-dual; notice that C_ℓ^* is only defined up to reordering of the nodes, we will adopt the convention of preserving the ordering of the linearized circuit, while we take the freedom to permute the scalar part as it will be more convenient.

Proposition 3.24 *The size of the ℓ-dual is the same as that of C, more precisely*

$$\begin{aligned} \mathrm{size}_{\{+,\&\}}(C) &= \mathrm{size}_{\{+,\&\}}(C^*_\ell), & \mathrm{size}_{\{*\}}(C) &= \mathrm{size}_{\{*\}}(C^*_\ell), \\ \mathrm{size}_{\{\eta_0,\omega\}}(C) &= \mathrm{size}_{\{\eta_0,\omega\}}(C^*_\ell), & \mathrm{size}_{\{\eta_a \mid a\neq 0\}}(C) &= \mathrm{size}_{\{\eta_a \mid a\neq 0\}}(C^*_\ell). \end{aligned}$$

Its depth is at most twice that of C.

Proof. The statement on the size follows immediately from Corollary 3.16 and the fact that the scalar part is left unchanged. For the depth, observe that any path in C^*_ℓ cannot exit the linearized circuit once it has entered it, thus it is at most composed of a path p in the scalar part of C and a reverse path p' in C_ℓ; since both p and the reverse of p' are paths of C, the sum of their lengths is at most twice the depth of C. 

3.2.2 Bilinear chains

The case of bilinear circuits has received particular interest because it allows to give lower bounds on the complexity of matrix multiplication [Fid73]. In this section we just point out how the results of Hopcroft and Musinski [HM73] and Fiduccia [Fid73] reduce to ours.

Definition 3.25 (Linear chain) Let R be non-commutative and let $S \subset R$ be a subring of its center. A circuit C over $(R, \mathcal{L})$ such that no directed path in C contains two nodes $v \neq v'$ with $\beta(v) = *_a$ and $\beta(v') = *_{a'}$ where $a, a' \notin S$ is called an *S-linear chain*.

We have seen in Remark 3.17 that in the non commutative case the transposition principle *does not transpose matrices*. It is however possible to transpose linear chains.

Definition 3.26 (Opposite circuit) Let $C = (V, E)$ be a circuit over $(R, \mathcal{L})$, the *opposite circuit* of C, denoted by C^{op}, is the arithmetic circuit over $(R^{\mathrm{op}}, \mathcal{L})$ where any $\beta(v) = *_a$ has been changed to $*_{a^{\mathrm{op}}}$.

Proposition 3.27 *Let C be a linear chain and let $\mathrm{eval}_C(x) = x^\top M$ for some matrix M, then $\mathrm{eval}_{C^{\mathrm{op}}}(x) = M^\top x$.*

Proof. This is a consequence of the electrical network lemma. The matrix M associated to eval_C is given by

$$m_{ij} = \pi_j \circ \mathrm{eval}_C \circ \iota_i(1) = \sum_{p \in x_i \rightsquigarrow y_j} \mathrm{eval}_p(1) = \sum_{p \in x_i \rightsquigarrow y_j} p_1 p_2 \cdots p_{n_p}, \tag{3.18}$$

where $*_{p_1}, \ldots, *_{p_{n_p}}$ are the scalar multiplication nodes on the path p.

Now, by the definition of linear chain, on any path there is at most one element not in the center of R, thus

$$(p_1 p_2 \cdots p_{n_p})^{\mathrm{op}} = p_1^{\mathrm{op}} p_2^{\mathrm{op}} \cdots p_{n_p}^{\mathrm{op}}, \tag{3.19}$$

and the claim follows.

Thus, to any linear chain one can associate the four circuits $C, C^*, C^{\mathrm{op}}, C^{\mathrm{op}*}$. Hopcroft and Musinski [HM73], consider bilinear chains, i.e. bilinear circuits

whose only two non-trivial linearizations are linear chains. They start from a formula to compute p bilinear forms given by $m \times n$ matrices with coefficients in the center S of R. Call this a (m, n, p)-formula. They show that any such formula can be transformed in (m, p, n), (n, m, p), (n, p, m), (p, m, n), (p, n, m)-formulas using the same number of multiplications in S and in $R \setminus S$.

Their result can be derived from this section by observing that their formula is a bilinear chain with linearizations ℓ_1, ℓ_2, and that their five transformed formulas are given by the bilinear chains

$$C^{op}, C^*_{\ell_1}, C^{op*}_{\ell_1}, C^*_{\ell_2}, C^{op*}_{\ell_2};$$

where the opposite circuit of a multilinear circuit is defined by swapping every multiplication node.

Their bounds on the number of multiplications follow by considering the sets $D = \{*_a | a \in S\}$ and $M = \{*_a | a \notin S\}$ and applying Proposition 3.24.

A consequence of their result is that any formula to multiply an $m \times n$ matrix by an $n \times p$ can be transformed in five formulas of the same complexity to multiply matrices of sizes permuted as above.

3.3 Straight Line Programs

One can view arithmetic circuits as algorithms, where each node is an elementary step. Then, the size of a circuit is a measure of complexity in terms of number of elementary (algebraic) operations. However, arithmetic circuits do not carry any information about space complexity.

Straight line programs (SLP) allow to reason about both space and time complexity: they can be seen as evaluation strategies for arithmetic circuits, carrying information about registers to store intermediate results. Informally speaking, they are programs that are only made of a sequence of assignments (no branchings, no loops). See [BCS97b] for formal definitions and proofs.

We work in the algebraic RAM model of [Kal88]; this is to the classic RAM model what the BSS model [BSS89] is to the Turing machine. In a slightly simplified way, an R-algebraic RAM (Random Access Machine) has a memory made of an infinite set of registers that can contain an arbitrary element of R, and a CPU that can perform arithmetic operations on elements of R and store the result in a register.

An SLP can be seen as a program for an algebraic RAM: its inputs are initially stored in some registers, its instructions are executed in order, and its outputs are to be read in some other registers. For example, the program

$$\begin{aligned} R_3 &\leftarrow R_1 + R_2 \\ R_3 &\leftarrow R_3 * 3 \\ R_2 &\leftarrow R_2 * R_3 \end{aligned} \tag{3.20}$$

expects its inputs in the registers R_1 and R_2, performs an addition, a scalar multiplication and a multiplication, and stores its output in the registers R_2 and R_3 (in our context, it is somehow arbitrary to decide which are the input and output registers). In this model, the time complexity of an SLP is given by the number of instructions, the space complexity by the number of different registers used.

3.3.1 The BLS model

In this section we study a particular family of *linear straight line programs* introduced by Bostan, Lecerf and Schost [BLS03] to study the transposition principle. They consider straight line programs consisting uniquely of the two operations

$$R_i \leftarrow R_i + R_j, \qquad \text{also written } R_i \xleftarrow{+} R_j, \tag{3.21}$$

$$R_i \leftarrow R_i * a, \qquad \text{also written } R_i \xleftarrow{*} a, \tag{3.22}$$

where R_i, R_j are registers and $a \in R$.

Such SLP's can compute the same morphisms as linear circuits over the basis BLS:

$$\mathrm{XOR}_1 : \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}, \quad \mathrm{XOR}_2 : \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \quad *_a : (a) \text{ for } a \in R. \tag{BLS}$$

In fact, consider a circuit C over (R, BLS) and let $x_1, \ldots, x_n$ be its inputs. Allocate n registers $R_1, \ldots, R_n$ and initialize them to the values of $x_1, \ldots, x_n$. Then, walk through C in any topological order and for any $*_a$ acting on R_i issue the instruction

$$R_i \xleftarrow{*} a, \tag{3.23}$$

for any XOR_1 acting on R_i and R_j issue the instruction

$$R_i \xleftarrow{+} R_j, \tag{3.24}$$

and for any XOR_2 acting on R_i and R_j issue the instruction

$$R_j \xleftarrow{+} R_i. \tag{3.25}$$

Observe that all the operators in BLS have the same input and output arities, then circuits over BLS necessarily have the same number of inputs and outputs. Hence, if a circuit has n inputs (and outputs), any topological order yields a straight line program using n registers by this evaluation strategy. Inversely, it is clear that any straight line program using only instructions (3.21) and (3.22) can be represented by a circuit over BLS having n inputs (and outputs).

Thus, we identify circuits over BLS with such SLP's (note that the identification is not one-to-one as different topological orders yield different SLP's), and define the space complexity in the algebraic RAM model of a circuit over BLS as its number of inputs (and outputs).

Finally, observing that XOR_1 is the dual of XOR_2, we deduce that any circuit C on (R, BLS) has a dual circuit with the same space and time complexities in the algebraic RAM model.

Remark 3.28. Let $(L_1, \ldots, L_k)$ be a SLP on n registers, where L_i is one of the instructions (3.21) or (3.22). By what we just said we can take as its dual the sequence $(L_k^*, \ldots, L_1^*)$, where L_i^* is defined as

$$(R_i \xleftarrow{+} R_j)^* = R_j \xleftarrow{+} R_i, \tag{3.26}$$

$$(R_i \xleftarrow{*} a)^* = R_i \xleftarrow{*} a. \tag{3.27}$$

3.3.2 Linear straight line programs

The step from the SLP's we just defined to classic linear SLP's is very small. In fact, all one has to do is simulate the instructions

$$R_i \leftarrow R_j * a \quad \text{with } i \neq j, \tag{3.28}$$
$$R_i \leftarrow R_j + R_k \quad \text{with } i \neq j, k. \tag{3.29}$$

The first one can be simulated by the sequence

$$\begin{aligned} &R_i \overset{*}{\leftarrow} 0, \\ &R_i \overset{+}{\leftarrow} R_j, \\ &R_i \overset{*}{\leftarrow} a; \end{aligned} \tag{3.30}$$

and the second one by

$$\begin{aligned} &R_i \overset{*}{\leftarrow} 0, \\ &R_i \overset{+}{\leftarrow} R_j, \\ &R_i \overset{+}{\leftarrow} R_k. \end{aligned} \tag{3.31}$$

It is reasonable not to count multiplications by 0, as these just require to free some memory, than one sees that transposing linear SLP's preserves the space complexity and looses a factor of at most two on time complexity. However this is clumsy: one can do much better by transposing directly the instructions (3.28) and (3.29).

Definition 3.29 (Double use) We say that a register R_i is *doubly used* in a sequence of instructions $(L_1, \dots, L_n)$ if it appears on the right hand side of two instructions L_i and L_j, and no instruction L_k for $i < k < j$ is of the form (3.28) or (3.29).

The matrix of the instruction $R_i \leftarrow R_j * a$ is $\left(\begin{smallmatrix} 1 & 0 \\ a & 0 \end{smallmatrix}\right)$ in general, but simply $\left(\begin{smallmatrix} 0 & 0 \\ a & 0 \end{smallmatrix}\right)$ if R_j is not doubly used; in the second case, the transposition is

$$\begin{aligned} &R_j \leftarrow R_i * a, \\ &R_i \overset{*}{\leftarrow} 0. \end{aligned} \tag{3.32}$$

Similarly, the matrix of $R_i \leftarrow R_j + R_k$ is

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix} \tag{3.33}$$

if R_j and R_k are not doubly used; this transposes to

$$\begin{aligned} &R_j \leftarrow R_i, \\ &R_k \leftarrow R_i, \\ &R_i \overset{*}{\leftarrow} 0 \end{aligned} \tag{3.34}$$

(notice that a double use is introduced by this transposition).

By comparing this to equations (3.30) and (3.31), one sees that each double use of a register introduces a $\overset{+}{\leftarrow}$ in the transposed code, and each addition introduces a double use in the transposed code. This corresponds well to the duality between $+$ and $\&$.

In conclusion, one sees that the sum of additions and double uses stays unchanged when transposing generic straight line programs. Again, it is reasonable not to count multiplications by 0 (in fact, they can be merged to the next assignment to the register). Copies of registers like in (3.34) are still a problem in the algebraic RAM model, but at a higher level of abstraction they can be handled using references (or one can simplify the code by hand, if his code has to run on an algebraic CPU!). Thus we can state the following version of the transposition theorem for straight line programs.

Theorem 3.30 (Transposition theorem) *Any linear straight line program S computing a function f can be transformed in a new straight line program S^* computing f^*. S and S^* use the same number of registers. The sum of the algebraic time complexity and the number of double uses of registers is the same for S and S^*.*

3.3.3 R-algebraic transforms

One rarely programs with straight line programs: to make transposition really useful, we must transpose families of SLP's. Bostan, Lecerf and Schost consider SLP's parameterized by integers and booleans.

Definition 3.31 (algebraic transform) Let R be a ring, an R-*algebraic transform* is a program in the algebraic RAM model composed by the following constructs:

- linear algebraic assignments of the forms (3.21), (3.22), (3.28), (3.29);
- for loops with iterator ranging over a list of non-algebraic registers;
- conditionals with tests over non-algebraic registers;
- function calls, recursive function calls.

If the program is recursive, it must terminate on any valid input.

By extension, we shall also call R-algebraic transform any algorithm that can be expressed in this model. This is equivalent to consider circuit families; for example, Algorithm 3.1 corresponds to a circuit family with parameter space $\mathbb{N}$, computing the map

$$\begin{aligned} &\mathbb{N} \to \hom(R^2, R^2), \\ &n \mapsto \begin{cases} \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right) & \text{if } n = 0, \\ \left(\begin{smallmatrix} f_n & f_{n+1} \\ f_{n-1} & f_n \end{smallmatrix}\right) & \text{if } n > 0 \text{ is odd}, \\ \left(\begin{smallmatrix} f_{n-1} & f_n \\ f_n & f_{n+1} \end{smallmatrix}\right) & \text{if } n > 0 \text{ is even}, \end{cases} \end{aligned} \tag{3.35}$$

where f_n is the n-th Fibonacci number.

It is clear that, for any value of the non-algebraic parameters, an R-algebraic transform corresponds to a SLP, then the transposition theorem can be applied to it. In practice, one leaves conditionals untouched and reverses for loops; function calls (recursive or not) are substituted by their transpose. For example, Algorithm 3.1

ALGORITHM 3.1: R-algebraic transform

```
INPUT : a, b ∈ R; n ∈ ℕ.
  FOR i ∈ [1, ..., n] DO
    IF i is odd THEN
      a ←+ b;
    ELSE
      b ←+ a;
  return a, b;
```

ALGORITHM 3.2: Transposition of Algorithm 3.1

```
INPUT : a, b ∈ R; n ∈ ℕ.
  FOR i ∈ [n, ..., 1] DO
    IF i is odd THEN
      b ←+ a;
    ELSE
      a ←+ b;
  return a, b;
```

becomes Algorithm 3.2; we let to the reader the verification that the transposed algorithm computes the transpose of maps (3.35) for any n.

Putting together the results of this section and the previous ones, we can now state the transposition theorem for algebraic transforms.

THEOREM 3.32 (Transposition theorem) *Any* R*-algebraic transform* T *computing a linear function* f *can be transformed in an* R*-algebraic transform* T^* *computing* f^*. T *and* T^* *use the same number of registers. The sum of the algebraic time complexity and the number of double uses of registers is the same for* T *and* T^*.

Note. Observe that some care must be taken when counting double uses in for loops: a single assignment in a for loop counts as $n-1$ double uses, where n is the number of times the loop is repeated.

3.3.4 R-*algebraic algorithms*

The transposition theorem for algebraic transforms is an important result that we shall use in the following chapters. However, if we want to transpose the multiplication or the Euclidean division of Section 2.2.8, we need to consider SLP's parameterized by algebraic elements.

With a little of hand waving, the transposition theorem is applied to algorithms parameterized by algebraic elements in [BLS03]:

> "Last, we will consider algorithms mixing linear and non-linear pre-computations; the transposition principle leaves the latter unchanged."

We shall call R*-algebraic* any algorithm that can be expressed in the algebraic RAM model, and that terminates on any input. Formally, the way an R-algebraic algorithm can be transposed is by *partial evaluation* [CD93, RV04, CKS09].

DEFINITION 3.33 (Partial evaluation) Let M, N be free R-modules (non necessarily finite). Let $A : M \times \mathcal{P} \to N$ be an R-algebraic algorithm and let $p \in \mathcal{P}$. The *partial*

evaluation of A on p is the algorithm

$$\begin{aligned} A_p : L &\to N, \\ x &\mapsto A(x;p). \end{aligned} \tag{3.36}$$

If for any $p \in \mathcal{P}$ the partial evaluation A_p is a straight line program, then we can apply the transposition theorem to A_p. For example, if $M : R[X] \times R[X] \times \mathbb{N} \to R[X]$ is a polynomial multiplication algorithm, then for any $b \in R[X]$, the *transposed multiplication* is $M^*_{b,\deg b}$, as in Section 2.2.8.

There are many strategies to compute partial evaluations, we shall see one in Section 4.2. The simplest one is to evaluate all the expressions that depend on p and store them in memory so that they can be used in A_p as constants; this is similar to the linearization of arithmetic circuits we saw in Section 3.2. Obviously, the cost in time complexity is bounded by the time complexity of A; however, the cost in space complexity is also bounded by the time complexity of A.

Principle 3.34 (Transposition principle) *Let $\mathcal{P}$ be an arbitrary set. Any* R*-algebraic algorithm A computing a family of linear functions $(f_p : M \to N)_{p\in\mathcal{P}}$ can be transformed in an* R*-algebraic algorithm A^* computing the* dual family $(f_p^* : N^* \to M^*)_{p\in\mathcal{P}}$*. The algebraic time and space complexities of A^* are bounded by the time complexity of A.*

Notice, however, that in many practical instances of transposition, the partially evaluated values constitute a negligible amount of the space used by the algorithm. Thus, in practice, transposed algorithms often have the same space *and* time complexities as the original ones; this is the case for all the transposed algorithms that appear in this document.[2]

Note. Is this the end of the story? Probably not. Umans and Kedlaya [KU08] have recently shown an example of non-algebraic algorithm that can be transposed with no loss in space and time complexity. This makes one wonder what the true limits of the transposition principle are.

3.4 Automatic differentiation

Automatic differentiation (AD) studies the following question: given a program to evaluate a function $f : R^n \to R$ at a point of R^n, how much does it cost to evaluate the gradient ∇f at a point of R^n. More generally, one can consider functions $R^n \to R^m$ and ask for the evaluation of the Jacobian matrix J_f. The two main techniques in AD are the *forward* and the *reverse mode*; Griewank [GW08] traces back their origins to [BKSF59] and [OWB71] respectively.

3.4.1 From automatic differentiation to transposition

Transposition of linear straight line programs reduces to automatic differentiation, in fact, if one has a program computing a linear function $f : R^n \to R^m$ and an $\ell \in (R^m)^*$,

$$\nabla(\ell \circ f) = \left(\frac{\partial \ell \circ f}{\partial x_1}, \dots, \frac{\partial \ell \circ f}{\partial x_n} \right) = \left(\langle f^*(\ell) | \boldsymbol{e}_1 \rangle, \dots, \langle f^*(\ell) | \boldsymbol{e}_n \rangle \right), \tag{3.37}$$

[2] Actually, the author is not aware of *any* application of the transposition principle where space complexity is not preserved, although it is easy to artificially create examples that behave badly.

where $(e_1, \ldots, e_n)$ is the standard basis of $\mathbb{R}^n$. Thus, differentiating the program for $\ell \circ f$ yields the coordinates of $f^*(\ell)$ in the standard basis of $(\mathbb{R}^n)^*$ as requested.

Automatic differentiation is widely used in numerical computations and this is why there is an extensive literature on it. In particular the reverse mode with the *checkpoint method* of Griewank [Gri92] implies that automatic differentiation of functions $\mathbb{R}^n \to \mathbb{R}$ can be done with a constant factor penalty in algebraic time complexity and an $O(\log(n))$ penalty in algebraic space complexity. However, this is still far from the bounds of Theorem 3.32.

Using the method of the adjoint code with optimizations for linear instructions [GLVM91], it is possible to save even more and ultimately reduce to the bounds of the transposition principle. This is not surprising as the adjoint code on linear programs is exactly the same thing as the transposition of linear straight line programs we saw in the previous section.

However, automatic differentiation uses a lot of machinery that has been tailored for non-linear programs. Using it for transposition is just overkill. Even worse, it is clumsy because automatic differentiation is built on top of the transposition principle as it was pointed out by [GG05]. To see this we shall briefly recall how automatic differentiation works on arithmetic circuits.

3.4.2 *Differentiation of arithmetic circuits*

One of the most influential results on the automatic differentiation of arithmetic circuits is due to Baur and Strassen [BS83]. They show that a non-linear arithmetic circuit that computes a function $f : \mathbb{R}^n \to \mathbb{R}$ can be transformed in a circuit to compute f and ∇f with a three-fold increase in size. Gashkov and Gashkov [GG05] interpret their method as a transformation that yields a circuit to compute f and the differential $\mathrm{d}f$ at a point x and n differential inputs $\mathrm{d}x_1, \ldots, \mathrm{d}x_n$; then, an application of the transposition principle on linearized circuits as in Section 3.2.1, yields the original result of Baur and Strassen.

Here we describe the transformation of [GG05] in a simplified manner. A complete description can be found in [GG05, Ser08].

For simplicity, we consider an arithmetic circuit C over a non-linear basis $\mathcal{B}$ over $\mathbb{R}$ made exclusively of everywhere continuously differentiable functions (w.r.t the standard metric of the Euclidean space $\mathbb{R}^n$). We describe a technique to compute the differential of eval_C at a point $a \in \mathbb{R}^n$.

Definition 3.35 (Differential of a circuit) Let $C = (V, E, \leqslant, \leqslant_i, \leqslant_o)$ be a circuit over $(\mathbb{R}, \mathcal{B})$ with n inputs and m outputs and let $a \in \mathbb{R}^n$. For any function $f \in \mathcal{B}$, we denote by J_f its Jacobian.

The *differential* of C at a, denoted by $\mathrm{d}_a C$, is obtained by substituting each $\beta(v)$ with

$$\beta'(v) = J_f\left(\mathrm{eval}_{e_1}(a), \ldots, \mathrm{eval}_{e_k}(a)\right) \tag{3.38}$$

for any $v \in V$, where $e_1, \ldots, e_k$ are the edges incident to v.

Proposition 3.36 *The differential of a circuit satisfies* $\mathrm{eval}_{\mathrm{d}_a C} = J_{\mathrm{eval}_C}(a)$ *for any* $a \in \mathbb{R}^n$.

Proof. Let e be an edge of C, we shall denote by eval_e its evaluation as in Definition 3.6, and by eval'_e the evaluation of the corresponding edge in $\mathrm{d}_a C$. We prove

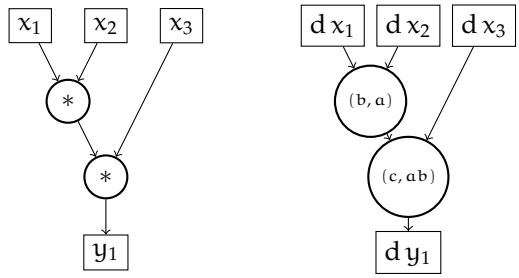


Figure 3.5: A circuit and its derivative at the point (a, b, c). We have replaced multiplication nodes with linear applications represented by 1×2 matrices.

that for any $a \in \mathbb{R}^n$ and for any edge e of C, the differential of eval_e at a is eval'_e. The proof is by induction and follows by the chain rule.

Let $x_1 \leqslant_i \cdots \leqslant_i x_n$ be the inputs of C, we write $f(\boldsymbol{x})$ for $f(x_1, \ldots, x_n)$. Let v be a node, let $e_1 \leqslant_v \cdots \leqslant_v e_k$ be its input edges and let e be the i-th output edge. Set $f = \beta(v)$, then by definition

$$\begin{aligned} \mathrm{eval}_v &= f \circ (\mathrm{eval}_{e_1}, \ldots, \mathrm{eval}_{e_k}), \\ \mathrm{eval}_e &= \pi_i \circ \mathrm{eval}_v, \end{aligned} \tag{3.39}$$

where π_i is the i-th projection. Then, the Jacobian matrix of eval_v is

$$\mathrm{J}_{\mathrm{eval}_v}(\boldsymbol{x}) = \mathrm{J}_f(\mathrm{eval}_{e_1}, \ldots, \mathrm{eval}_{e_k})\mathrm{J}_{(\mathrm{eval}_{e_1}, \ldots, \mathrm{eval}_{e_k})}(\boldsymbol{x}). \tag{3.40}$$

Hence, $\mathrm{d}_a \mathrm{eval}_v$ is equal to

$$\mathrm{J}_{\mathrm{eval}_v}(a) \begin{pmatrix} \mathrm{d}_a x_1 \\ \vdots \\ \mathrm{d}_a x_n \end{pmatrix} = \beta'(v) \begin{pmatrix} \mathrm{d}_a \mathrm{eval}_{e_1} \\ \vdots \\ \mathrm{d}_a \mathrm{eval}_{e_n} \end{pmatrix} = \beta'(v) \circ (\mathrm{eval}'_{e_1}, \ldots, \mathrm{eval}'_{e_n}), \tag{3.41}$$

where the last equality follows by induction. By definition, this is the evaluation of v in $\mathrm{d}_a C$, and the claim follows by composing with π_i.

It is also clear that $\mathrm{d}_a C$ is defined over a linear basis over $\mathbb{R}$, thus the transposition theorem applies to it. In other words we have defined a transformation from circuits computing differentiable functions to linear circuits.

Also notice that when C is a linear circuit, then simply $C = \mathrm{d}_a C$ for any a. In this case, Eq. (3.37) amounts to plug the form ℓ at the bottom of C, so that transposing $\mathrm{d}_a C = C$ and evaluating at 1 gives the desired coefficients of $f^*(\ell)$.

Note. The construction of [GG05] is more powerful: they start from a non-linear circuit C with input nodes $x_1, \ldots, x_n$, and they augment it to obtain a non linear circuit C' with input nodes $x_1, \ldots, x_n, \mathrm{d}\,x_1, \ldots, \mathrm{d}\,x_n$; this circuit admits a linearization $\ell = \{\mathrm{d}\,x_1, \ldots, \mathrm{d}\,x_n\}$, in the sense of Definition 3.21, and they prove that $C'_\ell = \mathrm{d}_{\boldsymbol{x}} C$. Then, Baur and Strassen's theorem follows by considering C'^*_ℓ.

3.4.3 *From transposition to automatic differentiation*

The circuit $\mathrm{d}_a C$ is an important intermediate step to compute the gradient: any automatic differentiator computes it, either explicitly or implicitly.

Now $\mathrm{d}_a C$ can be queried by black-box algorithms to obtain information about the Jacobian $J_{\mathrm{eval}_C}(a)$. The simplest application is to compute the directional derivative in a along a direction u: for this task it suffices to evaluate the circuit once, since $\mathrm{eval}_{\mathrm{d}_a C}(u)$ is the desired value. Computing the derivative along n linearly independent directions yields the whole Jacobian matrix and this roughly corresponds to the forward mode in automatic differentiation.

Remark 3.37. To be more precise, forward mode automatic differentiation constructs $\mathrm{d}_a C$ and evaluates the n directions in parallel, thus avoiding the need to store the whole circuit in memory. This is a great advantage for iterative code, where C can be represented compactly by a for loop, but $\mathrm{d}_a C$ needs the loop to be unrolled.

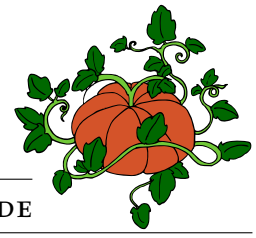
When the circuit has many inputs but only one output, there is a more convenient way to get $\nabla \mathrm{eval}_C$ with only one black-box query: $\mathrm{d}_a C$ computes a linear form whose coefficients are exactly the coefficients of the gradient, thus the dual circuit $(\mathrm{d}_a C)^*$ computes the transposed form. The single query $\mathrm{eval}_{(\mathrm{d}_a C)^*}(1)$ yields this vector. This is exactly what is called "reverse mode" in automatic differentiation.

Remark 3.38. Unlike the forward mode, reverse mode cannot compute $(\mathrm{d}_a C)^*$ and evaluate on a direction in parallel. One solution is to store the whole $\mathrm{d}_a C$ in memory, but this object may be too large. The checkpoint method of Griewank [Gri92] computes $\mathrm{d}_a C$ by *slices* of logarithmic size and transposes them one by one. Another way to gain space is to observe that the Jacobian of a linear operations (e.g., sums) does not depend on a, thus it does not need to be precomputed; this technique is suggested in [GLVM91], although it has seldom been implemented.

Whatever one does to save space, the key observation is the following: the generation of code to compute $\mathrm{d}_a f$ and its transposition can be done in two separate phases. Thus, automatic differentiation can concentrate of finding techniques to save space in the computation of $\mathrm{d}_a f$, while *automatic transposition* can concentrate on *reversing* code.

Note. One is not limited to forward or reverse mode: any black-box algorithm can be combined with the differential circuit to obtain information on the original function. For example, suppose that C is a circuit computing a function $f : \mathbb{R}^n \to \mathbb{R}^n$, and take $a \in \mathbb{R}^n$; one can apply Wiedemann's algorithm [Wie86] to $\mathrm{d}_a C$ to determine if f is invertible around a, and to compute the directional derivatives of f^{-1} in a neighborhood of $f(a)$.

4 Chapter



Automatic transposition of code

In this chapter we present a joint work with Schost [DFS10]. We study the *automatic transposition* of generic code (i.e. not limited to straight line programs). Section 3.4 has shown that this has applications in automatic differentiation, and we will see other applications in the next chapters.

By looking at a specific subproblem of automatic differentiation, our goal is to be more efficient and more general. In particular, compared to the existing implementations of AD tools, we want to:

- avoid unnecessary space overhead;
- handle algebraic, rather than just numerical code;
- handle advanced programming constructs, including recursion and algebraic data types;
- transpose code parameterized by arbitrary algebraic variables.

In this chapter we shall abandon the algebraic RAM model we used in Section 3.3 and work on source code transformation. Implementation details such as knowing what the cost of copying variables is, shall be ignored: one can assume that a good compiler will optimize most of those details. Hence, we shall assume that Theorem 3.32 really reflects the behavior of the code we generate.

4.1 Inferring linearity

By looking at Section 2.2.8 one sees that often we want to transpose families of R-algebraic algorithms parameterized by algebraic elements (e.g., we want to transpose the code that for any $a \in R$ evaluates the map $b \mapsto ab$). This is also necessary in automatic differentiation, when the code for $d_x f$ not only depends on $d x_1, \ldots, d x_n$, but also on $\mathbf{x}$.

The next section will address the question of how to transpose such code. This section, instead, asks the question: can a compiler guess by itself which inputs to a function are parameters, and which are linear arguments?

The answer is yes. We show how the type system of common statically typed functional languages can be extended to automatically infer all the possible *linearizations* of a computer program. We first present the non-commutative case, which can be fully expressed inside the Haskell type system, then we discuss how to extend to the commutative case.

Linears, scalars. Suppose we have defined some data type `R` representing elements of a ring R together with the usual constants (say `zeroR`, `oneR`, etc.), arithmetic operations (say `plus`, `times`, etc.), tests and so on. To simplify, we assume –as usual in algebraic complexity theory– that the type `R` is isomorphic to R, i.e. the elements of R can be represented exactly, the operations do not introduce any rounding error, etc.

For any term involving elements of type `R` we would like the type system to tell us whether its outputs are linear in its inputs. For example the term

```
\x y -> plus x y
```

has type `R -> R -> R`, but we would like the type checker to also output something like $\ell \to \ell \to \ell$ (ℓ for *linear*) telling that the term is a (curryfied) left module homomorphism from R^2 to R. For consistency, we want to view constants as mappings from R^0 to R, thus for the term `zeroR` we want the type checker to compute something like $0 \to \ell$, that we simply write as ℓ.

Now, what do we expect about `oneR` or `times`? The former is the mapping $\perp \mapsto 1$, which is not a module homomorphism; then, by analogy with Definition 3.21, we want the type checker to output something like $0 \to s$, or simply s (s for *scalar*). The second can be made into a linear mapping by *fixing* its second argument (remember that for the moment we are restricting to left modules) as we did in Section 3.2; thus we expect the type checker to output $\ell \to s \to \ell$, meaning that

```
\x -> times x y
```

is a left module homomorphism $R \to R$ for any `y::R`.

Finally consider the following term

```
z x n = if n <= 0 then zeroR else plus x (z x (n-1))
```

as before we expect something like $\ell \to \mathbb{N} \to \ell$, meaning that

```
\x -> z x n
```

is a homomorphism $R \to R$ for any integer `n`.

Observe that in order to make a correct inference about a term such as

```
\x y -> times x (plus y y)
```

we must also admit for any of the previous cases the possibility where everything is a scalar, so that from the hypothesis that `plus` has type $s \to s \to s$ we can deduce the correct type $\ell \to s \to \ell$ for the term above. Summarizing, we would like to have two types `L` and `S` such that the following equations hold

```
plus :: L -> L -> L
plus :: S -> S -> S
times :: L -> S -> L
times :: S -> S -> S
zeroR :: L
zeroR :: S
oneR :: S
```

If we define `L` and `S` as wrappers around `R`

```
newtype L = Lin R
newtype S = Sca R
```

then, using Haskell type classes [WB89], we can conveniently express all the equations above as

```
class Ring r where
  zero :: r
  (<+>) :: r -> r -> r
  neg :: r -> r
  (<*>) :: r -> S -> r
```

together with the obvious `instance` definitions (see the example in Appendix B).

For our inference to work, it is important that `L` be an abstract data type with only the above functions in its interface. On the other hand, any other function acting on `R` can be wrapped inside a function acting on `S` as, for example,

```
one = Sca oneR
(Sca a) == (Sca b) = a == b
```

or, simply, using a `deriving` clause in the declaration of `S`

```
newtype S = Sca R deriving (Eq)
```

If we restrict to terms that do not use the type constructor `Lin`, then we can show that the semantic of a term with type

```
L->...->L->L
```

is a left module homomorphism. The proof for the full language would be too long, thus we restrict to a simply typed λ-calculus with constants. Its terms are defined by the following grammar

$$t ::= c \mid x \mid t_0 t_1 \mid \lambda x.t\ , \tag{4.1}$$

where x are identifiers and c are constants; its types are defined by the grammar

$$\tau ::= \ell \mid s \mid \beta \mid \tau \to \tau\ , \tag{4.2}$$

where β are the usual base types (integers, booleans, etc.). If Γ is a type environment, by $\Gamma \vdash t :: \tau$ we mean that the term t has type τ in Γ. The semantic of our calculus is the usual one, based on $\beta\eta$-reduction.

We suppose all the constants above are defined, plus the usual constants for the other base types; observe that our grammar forbids type constructors altogether (including `Lin` and `Sca`). In this context we use the type names ℓ and s in place of the Haskell types `L` and `S` defined above. For simplicity, we shall also assume that lists and tuples are not part of the types of our language; see the end of this section for a discussion about them.

Definition 4.1 (Flipper) Let τ be the type

$$\tau = \alpha_0 \to \alpha_1 \to \cdots \to \alpha_n \tag{4.3}$$

with α_n not a function type. Let $I \subset [0, \ldots, n-1]$ such that $\alpha_i \neq \ell$ if and only if $i \in I$, and let $m = \#I$. The *flipper* for τ, denoted by flip_τ, is the term

$$\mathrm{flip}_\tau = \lambda t.\lambda x_{i_1}.\ldots.\lambda x_{i_m}.\lambda x_{j_1}.\ldots.\lambda x_{j_{n-m}}.t x_0 \cdots x_{n-1}, \tag{4.4}$$

with $i_1, \ldots, i_m \in I$ and $j_1, \ldots, j_{n-m} \in \bar{I}$.

LEMMA 4.2 *Let* $\Gamma \vdash t :: \tau$ *be a term, let* $m, n \geqslant 0$, *and let* $\Gamma \vdash \mathrm{flip}_\tau\, t :: \sigma$ *with*

$$\sigma = \alpha_1 \to \alpha_2 \to \cdots \to \alpha_m \to \underbrace{\ell \to \cdots \to \ell}_{n \text{ times}} \to \beta, \tag{4.5}$$

with $\alpha_i \neq \ell$ *and* β *not a function type. Let* $\Delta_i \vdash s_i :: \alpha_i$ *for* $1 \leqslant i \leqslant m$. *The semantic of*

$$\Gamma, \Delta_1, \ldots, \Delta_m \vdash \mathrm{flip}_\tau\, t s_1 \cdots s_m \tag{4.6}$$

is

1. *a constant function if* $\beta \neq \ell$,
2. *a module homomorphism* $R^n \to R$ *if* $\beta = \ell$,

assuming the free variables in $t, s_1, \ldots, s_m$ *satisfy 1 or 2.*

Proof. We distinguish the following cases.

- $\Gamma \vdash c$. All the constants satisfy either 1 or 2. We just work out 0 and $+$ defined above and leave the others to the reader; in both cases $\mathrm{flip}_\tau\, c = c$ up to $\beta\eta$-conversion.
  - $\Gamma \vdash 0 :: \ell$ is the map $\bot \mapsto 0$, thus a (constant) morphism.
  - $\Gamma \vdash 0 :: s$ is the map $\bot \mapsto 0$, thus a constant (morphism).
  - $\Gamma \vdash + :: \ell \to \ell \to \ell$ is the map $a, b \mapsto a + b$. A morphism.
  - $\Gamma \vdash + :: s \to s \to s$. Take any $a :: s$ and $b :: s$, then $\Gamma \vdash a + b :: s$ is a constant.
- $\Gamma, x :: \alpha \vdash x :: \alpha$. The claim follows because x is free.
- $\Gamma \vdash t_0 t_1 :: \tau$. This is the only real case to prove. We distinguish two cases:
  - $\Gamma \vdash t_1 :: \ell$, then, by induction its semantic is a morphism $0 \to R$ (because it is $\beta\eta$-equivalent to $\mathrm{flip}_\ell\, t_1$).
    Let $\Gamma \vdash \mathrm{flip}_\tau\, t_0 t_1 :: \sigma$, with σ as in Eq. (4.5), and let $\Delta_i \vdash s_i$ for $1 \leqslant i \leqslant m$ be as in the hypothesis. Let $\Gamma \vdash t_0 :: \tau_0$ and $\Gamma \vdash \mathrm{flip}_{\tau_0}\, t_0 :: \sigma_0$, then by induction
    $$\Gamma, \Delta_1, \ldots, \Delta_m \vdash t_0' \stackrel{\text{def}}{\equiv} \mathrm{flip}_{\tau_0}\, t_0 s_1, \ldots, s_m \tag{4.7}$$
    is either a morphism $R^{n+1} \to R$ or a constant function. In the first case $t_0' t_1$ is a morphism $R^{n'} \to R$, in the second case it is a constant function; in both cases
    $$\mathrm{flip}_\tau (t_0 t_1) s_1 \cdots s_m \overset{\beta\eta}{\longleftrightarrow} t_0' t_1 \tag{4.8}$$
    and the claim follows.
  - $\Gamma \vdash t_1 :: \alpha$ with $\alpha \neq \ell$. Then the claim follows directly by induction on t_0 and $\beta\eta$-conversion, by choosing $s_1 = t_1$.
- $\Gamma \vdash \lambda x.t :: \alpha_1 \to \alpha_2$. By induction $\Gamma, x :: \alpha_1 \vdash t :: \alpha_2$ satisfies 1 or 2 (assuming x does). We distinguish two cases

– $\alpha_1 \neq \ell$, then

$$\lambda x.\, \mathrm{flip}_{\alpha_2}\, t \overset{\beta\eta}{\longleftrightarrow} \mathrm{flip}_{\alpha_1 \to \alpha_2}(\lambda x.t); \tag{4.9}$$

– $\alpha_1 = \ell$, then

$$\lambda x.\, \mathrm{flip}_{\alpha_2}\, t s_1 \cdots s_m \overset{\beta\eta}{\longleftrightarrow} \mathrm{flip}_{\alpha_1 \to \alpha_2}(\lambda x.t) s_1 \cdots s_m. \tag{4.10}$$

In both cases, $\lambda x.t$ satisfies 1 or 2 accordingly.



Proposition 4.3 *Let* $t : \tau$ *be a closed term, let* $n \geqslant 0$ *and let*

$$\tau = \underbrace{\ell \to \cdots \to \ell}_{n \text{ times}} \to \beta, \tag{4.11}$$

with β *not a function type. Then, the semantic of* t *is*

1. *a constant function if* $\beta \neq \ell$,
2. *a module homomorphism* $R^n \to R$ *if* $\beta = \ell$.

By the proof, it should be now clear why we forbid the type constructor `Lin`. In fact, introducing a term as `Lin oneR :: L` tricks the proof (the type checker) by making it believe that the function $\bot \mapsto 1$ is a morphism.

The commutative case. In the commutative case we shall add a second multiplication operator allowing multiplication on the left by a scalar

```
class Ring r => CommRing r where
  (>*<) :: S -> r -> r
```

but this would force the user to choose between the two operators any time he multiplies two elements of `R`. To avoid this we need to overload the operator `(<*>)` with both type signatures, a technique sometimes called *ad-hoc* polymorphism [Str00], but this is not possible in the Haskell type system since the two types are contradictory. To make it possible we need to extend the type inference algorithm: our idea is not new, but it has been rarely implemented because it is not practical for solving generic *ad-hoc* polymorphism; it perfectly fits the needs of our special case, though.

First observe that type classes can be translated to ordinary types of the Hindley-Milner type system as explained in [WB89, §4], thus it suffices to modify the classic type inference algorithm [DM82, Car87]. Second, observe that there is some redundancy between the two signatures of `(<*>)` and that a more concise version is

```
(<*>) :: Ring r => r -> S -> r
(<*>) :: S -> L -> L
```

A review of the Hindley-Milner algorithm and its implementation can be found in [Car87]. The idea is to first assign type variables to terms, then solve type equations by unifying them. In our generalization, instead of handling a single unification, we keep a list of possible unifications: when a type equation implies

that a certain unification is not acceptable, the unification is discarded from the list; if the list gets empty the term cannot by typed and an error is returned, otherwise any unification in the list is valid and is returned.

In practice, the only term that makes the list of unification grow is `(<*>)`: any time an equation involving it has to be solved, the list of unifications potentially doubles. This exponential increase is the reason why this solution is not practical to solve generic *ad-hoc* polymorphism; but in our case we really are interested in knowing all the possible types of a term because each of them gives rise to a different linearization and, hence, to a different transposition.

Modules. Finally we remark that by allowing tuples and lists, Lemma 4.2 can be generalized to morphisms $R^m \to R^n$ and even to infinite dimensional modules using lazy lists. Elements of type `L`, `[L]`, `(L,L)`, etc. share a common pattern: they can be viewed as R-modules. It is convenient to summarize their properties in an unique interface[1]

```
class Ring r => Module m r | m -> r where
  zeroM :: m
  (<<*) :: m -> S -> m
  (>>>) :: m -> Integer -> r
  (<<<) :: r -> Integer -> m
  (<++>) :: m -> m -> m
  add :: m -> m -> Integer -> m
  add a b n = foldl (<++>) zeroM
              [((a>>>i) <+> (b>>>i))<<<i | i <- [1..n]]
```

Instances of this class represent free R-modules: `zeroM` is the zero element, `(<<*)` is scalar multiplication, `(<++>)` is addition, `(<<<)` and `(>>>)` are canonical injections and projections.

This interface adds nothing to the linearity inference system, but we will need it in Section 4.2. Also notice the presence of the operator `add` that performs addition up to a truncation order, it is of no great importance in this section, but for efficiency reasons we will eventually prefer it to plain addition.

A fully worked Haskell example of the ideas presented in this section (without the extension to the commutative case) is given in Appendix B where we implement Karatsuba multiplication of polynomials in $\mathbb{Z}[X]$.

4.2 transalpyne

We present here a "Python implementation of a transposable Algebraic Language", in short `transalpyne`.

`transalpyne` is a limited functional language incorporating all the features that we have discussed until now, in particular:

- it supports algebraic code (i.e., not restricted to native types such as integers or floats);
- it has a static type checker that only gives types to algebraic variables and performs linearity inference (see Section 4.1);

[1] We make use of some experimental modules of Haskell: this code needs the flags `-XMultiParamTypeClasses`, `-XFunctionalDependencies` and `-XFlexibleInstances` in order to work.

- it is able to transpose any linearization of an algebraic program (see Section 3.3.4);
- its code can be compiled to Python code, or interpreted inside the Python interpreter.

At the moment we write, the first stable release of `transalpyne` is not ready yet. We plan to start distribute it by the beginning of 2011. It will be available from `http://transalpyne.gforge.inria.fr/`.

4.2.1 Concepts

`transalpyne` has been conceived as a scripting language to be used on top of computer algebra systems. We made an effort to give syntax and semantics as close as possible to the Python programming language.

In `transalpyne` there is no such concept as an executable program: only functions can be defined in `transalpyne`. `transalpyne` programs can be compiled to Python code, so that their functions can be called by Python programs; we plan to support compilation to other languages in the future. `transalpyne` can also be interpreted via the Python interpreter. A `transalpyne` library contained in a file `my-library.yp` can be imported in a Python program via the statement

```
import my-library
```

The Python interpreter recognizes the `.yp` extension and launches the `transalpyne` interpreter on the file; the functions of the library are interpreted and transposed by the `transalpyne` interpreter and their names are exported to the Python namespace.

`transalpyne` is mostly dynamically typed, with the only exception of *algebraic types*. In order to transpose a function, `transalpyne` must know at *transpose time* which variables contain algebraic elements and which variables contain other data (such as booleans, strings, ints, etc.); this can be done by explicitly specifying the type of the input and output parameters of a function, while all the other variables can be left untyped. `transalpyne` supports two sorts of algebraic types: ring elements and module elements; we plan to support more complex algebraic types, such as algebras, in the future. `transalpyne` relies on Python operator overloading to represent ring operations.

`transalpyne`'s type checker also performs a linearity inference as described in Section 4.1. The base ring is supposed to be commutative, we plan to add support for non commutative rings in the future. When a function admits more than one linearization, `transalpyne` computes and transposes all of them; if this leads to an ambiguity in a function call, it raises a compile time error.

4.2.2 Syntax

We only describe `transalpyne` syntax informally. Indentation has a syntactic value (it delimits blocks) and keywords are pretty much the same as in Python. A `transalpyne` file contains a *type declaration* section followed by a *name definition* section.

Type declarations Type declarations let the user declare which are the algebraic types. `transalpyne` supports two type constructors: a ring constructor and a free module constructor.

```
type Ring R
type Module(R) M
```

This example declares `R` as a ring type and `M` as a free module type over the ring `R`. The typechecker ensures that modules are consistently declared.

NAME DECLARATIONS Name declarations take three forms: *imports*, *function definitions* and *aliases*. Imports are declared as in Python and have the same semantics. Note however that the type checker does not enter imported modules to infer linearity: any imported function is considered as a scalar function.

There is no `return` statement in `transalpyne`, function definitions are declared as follows

```
def (a, b)my-function(c, d):
```

where input arguments are given on the right and output arguments on the left.

Inside function definitions, there are four types of statements: `pass` statements (the statement that does nothing), assignments (including augmented assignments), `for` loops and `if`s. The syntax is identical to Python.

On the left hand side of assignments, may only appear variable names and subscripts. On the right hand side of assignments, the following types of expressions may appear:

- String, numeric and boolean constants;
- Binary and unary operators `+`, `-`, `*`, `/`, `%`, `div`, `mod`, `<`, `>`, `<=`, `<=`, `==`, `!=`, `and`, `or`, `not`, `in`;
- Parenthesized expressions;
- Subscripts and slices;
- List constructors, including comprehensions;
- Variable evaluations;
- Function calls.

The syntax for all of these is identical to Python. The only notable exception are function calls where a keyword `trans` is added to let the user call a transposition of a function. In case a function has more than one linearization (and thus more than one transposition), *signature specifiers* enclosed in braces `{`, `}` let the user specify which linearization/transposition is wanted.

Finally, aliases let the user export specific linearization/transpositions of functions with names that can be used inside a Python program.

Figure 4.1 gives a complete `transalpyne` example. It defines a `product` function and two aliases (with transposition and signature specifiers).

```
type Ring R

def (R c)product(R a, R b):
    c = a * b

l_product = trans {linear R}product{linear R, const R}
r_product = trans {linear R}product{const R, linear R}
```

Figure 4.1: A transalpyne program

4.2.3 *Semantics*

We only give here the points where transalpyne semantics differ from Python.

Types transalpyne is statically typed for algebraic types. The type of each input and output parameter of a function must be specified in the definition as in figure 4.1. When the type of an argument is omitted, it is assumed to have non-algebraic type. Variables inside the body of a function cannot be explicitly typed, the type checker deduces their types from the types of the input parameters.

Side effects There is no side effect in transalpyne. In particular, there is no global variable and assignment itself is a let-binding. After having transposed the functions, the transalpyne compiler/interpreter leaves to the target language the task of executing them, thus it cannot enforce the no-side-effect policy at runtime. It is the responsibility of the user to insure that no side effect happens inside a transalpyne function.

Conditionals, loops In order for the type inference to work, we must work around a feature of Python. The following is correct Python code, even if a is not defined before the if statement:

```
if x:
    a = 0
b = a
```

In case the if-block is not executed, Python simply issues a runtime error. If we want to do a type inference, we must enforce a stricter policy.

In transalpyne one cannot use outside of an if-block a variable that has been first assigned in one of the branches but not in all of them. If-blocks have a return type: it is the product of the types of all the variables modified inside one of its branches, and that appear in each branch or were defined before the if; this type must be the same at the issue of any branch. For example, in the following code

```
a = R.zero()
if True:
    a = R.one()
else:
    b = R.zero()
```

the return type of the if statement is the type of `a`, because it has been modified in the if branch, and was defined before the if, and is the same in both branches (namely, `R`). Using the value of `b` after the block generates a compile time error.

For loops also have a return type. To this extent, they are treated as if they were if-blocks with an empty else-block.

ALGEBRAIC VARIABLES Type declarations merely say that some variables belong to a type, but do not specify any particular implementation of the type. The implementation of rings and modules is left to the user and must be given in an external module written in Python (or in whatever the target language is). The user is only required to implement them as objects and to expose a few methods.

Ring objects must:

- Overload + and * with the obvious semantic;
- Implement a method `zero` that returns the zero of the ring;
- Optionally, implement a method `one` that returns the one of the ring;
- Optionally, implement a method `Z` that takes an integer `n` and returns the element $n \cdot 1$ of the ring;
- Optionally, implement methods `div` and `mod` that perform Euclidean division with remainder;
- Optionally, overload `/`, thus making the ring into a field.

Module objects must:

- Overload + and * with the obvious semantic;
- Implement a method `zero` that returns the zero of the module;
- Overload the subscript operator `[]` so that it implements some arbitrary projections on the underlying ring. Most often, a module will be implemented as an array of ring objects and `[i]` will just be projection onto the `i`-th coordinate.
- Overload the assignment-to-subscript operator in the obvious way.

Algebraic output parameters of a function are implicitly initialized to zero via their `zero` method. This insures that non-assigned algebraic output parameters are always linear in the inputs of the function.

Algebraic elements cannot be combined through the use of lists: lists of algebraic objects are non-algebraic objects and extraction from a list always yields a non-algebraic object.

FUNCTION CALLS `transalpyne` does not have tuples; the return type of a function with many output parameters is not a tuple, as a consequence its return value cannot be assigned to a variable: it must be assigned to as many variables as there are output parameters. Another consequence of this is that functions with many outputs cannot be used inside expressions: their outputs can only be assigned to variables.

Function names not declared in the library are simply regarded as external functions. They are assumed to have one return parameter, thus a multi-assignment will return an error. External functions have no algebraic input or output parameters. This is useful to call built-in Python functions from inside a `transalpyne` program (one common example is the function `range`, needed to iterate in for loops).

Recursion and Higher order `transalpyne` allows recursion and even calling its own transpose. It does not allow to pass functions as arguments to functions, although the transposition algorithm internally uses this technique to transpose for loops. A higher order transposable language is theoretically possible and we consider adding this feature to `transalpyne` in the future.

4.2.4 Linearization

After a first type checking to determine which variables are algebraic, `transalpyne`'s compiler/interpreter runs the linearization inference algorithm of Section 4.1.

Since a function may have more than one linearization, `transalpyne` allows the user to annotate the types of the algebraic arguments so that they can be constrained to be linears or scalars (non-algebraic arguments are by default scalars). The two keywords for this are `linear` and `const`, the following code shows an example of use:

```
def (linear R c)product(linear R a, const R b):
    c = a * b
```

The user is free to leave some arguments unspecified. For example, the previous code could have been written

```
def (R c)product(linear R a, R b):
    c = a * b
```

The linearity inference looks for all the linearizations compatible with the specified modifiers, thus in this case both codes yield the same linearization. If more than a linearization is acceptable, `transalpyne` computes all of them.

We call *signature* a list of linear/const modifiers inferred for the arguments of a function. Signature specifiers (see Section 4.2.2) can be used to distinguish between different linearizations when calling the transposed function. They are written as

```
{const R}product{linear R, const R}
```

Thus, in the example we gave in figure 4.1, `l_product` is an alias for the transposed left-linear product, while `r_product` is an alias for the transposed right-linear one. Aliases are extremely useful since they are the only way to export the transposed functions to the namespace of the target language.

4.2.5 Partial evaluation

After the type checking and linearity inference phase, any discovered linearization of any function is partially evaluated and transposed as in Section 3.3.4.

The first step is to translate any for loop into a tail-recursive function, as this simplifies greatly the partial evaluation. The partial evaluation is then done in

```
def (R a, R b)f(R c, R d):
  if d > R.zero():
    x, y = f(c, d - R.one())
    a, b = x * y, y + R.one()
  else:
    a, b = c, d
```

FIGURE 4.2: A transalpyne program that does nothing interesting, but is very hard to transpose (with respect to the signature {linear R, const R}f{linear R, const R}).

two steps: first we evaluate all the statements depending exclusively from the scalars, then we strip those statements off the partially evaluated program. Let us explain this through an example. Consider the program in Figure 4.2, we want to transpose it with respect to the signature $\ell \times s \to \ell \times s$.

First we generate the program that evaluates at d the statements that only depend on d, we call it fS.

```
def (R b)fS(R d):
  if d > R.zero():
    y = fS(d - R.one())
    b = y + R.one()
  else:
    b = d
```

Now we generate the partial evaluation of f at d by stripping off all the values that solely depend on d.

```
def (R a)f(R c):
  if d > R.zero():
    x = f(c, d - R.one())
    a = x * y
  else:
    a = c
```

Notice that this program needs the values of d and y that are computed in fS. Depending on whether the code is compiled or interpreted, we use a different strategy.

In case we generate code, we simply concatenate the bodies of fS and f, eventually performing α-conversion to avoid name clashes (notice that α-conversion is possible because we have eliminated for loops). Thus, the refactored code for f would look like in Figure 4.3.

Notice, however, that this is inefficient because it generates two recursive calls: one in the scalar part and one in the linear part. A better solution would be to evaluate fS(d) only once and save all its stack, so that all the scalar values needed in the partial evaluation can be retrieved from it. When we interpret code we choose an intermediate solution: we save the return value of any call to fS in a memoization table, then retrieve the values from the table when they are needed in the linear part. This is a sort of lazy evaluation of functions in the scalar part.

Remark 4.4. Regardless of whether the code is interpreted or compiled, scalar

```
def (R a, R b)f(R c, R d):
  # Scalar part
  if d > R.zero():
    y = fS(d - R.one())
    b = y + R.one()
  else:
    b = c
  # Linear part
  if d > R.zero():
    x, _ = f(c, d - R.one())
    a = x * y
  else:
    a = c
```

Figure 4.3: Refactoring of the code of Figure 4.2, separating the scalar from the linear part.

```
def (R c, R b)fT(R a, R d):
  # Forward sweep
  if (d > R.zero()):
    y = fS(d - R.one())
    b = y + R.one()
  else:
    b = d
  # Reverse sweep
  if (d > R.zero()):
    x = a * y
    c, _ = trans f(x, d - R.one())
  else:
    c = a
```

Figure 4.4: Transposition of Figure 4.2.

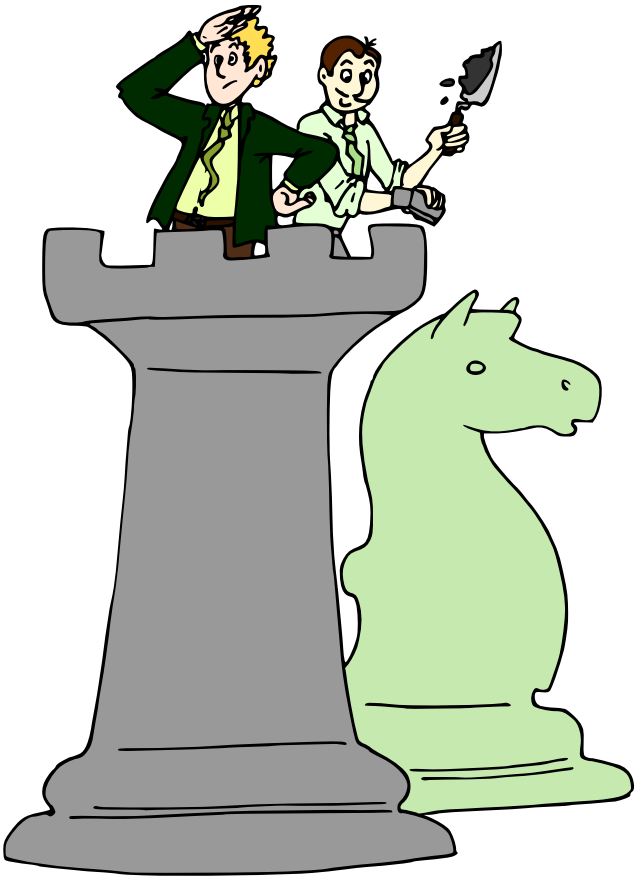
parts of real world algorithms tend to be very short and simple. In particular they seldom contain a recursive call having both scalar and linear return values as the one in Figure 4.2. Thus we can reasonably assume that the generated code is as efficient as the original one.

4.2.6 *Transposition*

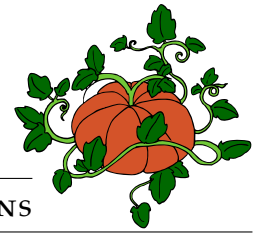
Finally, the linear part of each linearized function is transposed as in Section 3.3. In doing this we read the code from bottom to top and transpose each instruction, i.e. we swap input and output algebraic arguments of any function, and substitute each function call with a call to the transposed function. The example of Figure 4.2 is transposed in Figure 4.4. We borrow the names *forward sweep* and *reverse sweep* from the theory of automatic differentiation [GW08], where a similar technique is applied in the reverse mode.

Part III

Fast arithmetics using univariate representations



5 Chapter



Trace computations

The goal of this part of the document is to present some efficient algorithms to compute in some specific finite dimensional algebras over a field $\mathbb{K}$.

Let $\mathbb{K}$ be a field, and let $x_1, \ldots, x_n$ be indeterminates. We denote by $\mathbb{K}[\mathbf{x}]$ the algebra $\mathbb{K}[x_1, \ldots, x_n]$. Any finite dimensional $\mathbb{K}$-algebra $\mathcal{A}$ is isomorphic to a quotient $\mathbb{K}[\mathbf{x}]/I$ for some 0-dimensional ideal I. Residue classes of $\mathbb{K}[\mathbf{x}]$ modulo an ideal I are indeed a very good representation of the elements of $\mathcal{A}$.

The most popular tools to compute in generic residue class rings are Gröbner bases [Buc65, CLO05a, CLO05b, Fau99, Fau02]. An alternative to Gröbner bases, called *geometric resolution* is presented in [GLS01]. In the bivariate case, resultants [CLO05a, CLO05b] are a classic tool.

Besides the algorithmic tool used to compute in $\mathcal{A}$, the choice of a basis for the ideal I also has a great impact. Consider, for example, the ideal of $\mathbb{Q}[x, y]$

$$(x^2 + x + 1, y^3 - x), \tag{5.1}$$

another set of generators for the same ideal is

$$(y^6 + y^3 + 1, x - y^3). \tag{5.2}$$

Both sets of generators are Gröbner bases of I and identify $\mathbb{Q}[x, y]/I$ to $\mathbb{Q}(\zeta_9)$. However, while (5.1) naturally identifies $\mathbb{Q}[x]/(x^2 + x + 1)$ to the subfield $\mathbb{Q}(\zeta_3) \subset \mathbb{Q}(\zeta_9)$, this information is lost by (5.2), making it harder to test for membership in $\mathbb{Q}(\zeta_3)$ in this case.

Thus, algorithms to change from a set of generators to another are important too. The FGLM algorithm [FGLM93] computes the change from a Gröbner basis to another. There is also a variety of change-of-order algorithms for triangular sets based on resultants [BLMM01], on trace formulas [DTGV01, PS06], on Newton-Hensel lifting [DJMMS08]; while the *rational univariate representation* algorithm [Rou99] allows to go from a Gröbner basis to a geometric resolution.

In this chapter we focus on a generalization of Lagrange interpolation formula, called *trace formulas*, and on its application to the rational univariate representation of a zero-dimensional ideal. Section 5.1 studies the decomposition of a zero dimensional ideal in the algebraic closure of $\mathbb{K}$, then Section 5.2 introduces the trace formulas, and, in Section 5.3, Stickelberger's theorem makes the link between the trace formulas and the trace of the multiplication operator. We present the rational univariate representation algorithm and some algorithmic improvements in Sections 5.4, 5.5 and 5.6; finally, in Section 5.7 we show how it can be applied as a change-of-basis algorithm.

5.1 Decomposition of a zero-dimensional ideal

We let I be a zero-dimensional ideal of $\mathbb{K}[\mathbf{x}]$ and $\mathcal{A} = \mathbb{K}[\mathbf{x}]/I$. To simplify the exposition, from now on we assume that $\mathbb{K}$ is a perfect field and I is radical, this is equivalent to all the points of $V(I) = \{a \in \bar{\mathbb{K}}^n | f(a) = 0, \forall f \in I\}$ being simple. We address the reader interested in the case of arbitrary multiplicity to [EM07].

To better understand the structure of $\mathcal{A}$ it will be important to study the algebraic set $V(I)$. We denote by $\bar{I}$ the ideal of $\bar{\mathbb{K}}[\mathbf{x}]$ generated by I and by $\bar{\mathcal{A}}$ the quotient ring $\bar{\mathbb{K}}[\mathbf{x}]/\bar{I}$.

Lemma 5.1 *$\mathcal{A}$ is naturally identified to a subset of $\bar{\mathcal{A}}$.*

Proof. We want to prove $\bar{I} \cap \mathbb{K}[\mathbf{x}] = I$. The direction $\supset$ is clear. Let $f \in \bar{I} \cap \mathbb{K}[x]$ and let $f_1, \ldots, f_k$ be generators of I, then there exist $g_1, \ldots, g_k \in \bar{\mathbb{K}}[\mathbf{x}]$ such that

$$f = \sum_i g_i f_i. \tag{5.3}$$

Then the coefficients of $g_1, \ldots, g_k$ are the solutions of a linear system with coefficients in $\mathbb{K}$, thus they can be taken in $\mathbb{K}[\mathbf{x}]$.

Hence $f \equiv f' \bmod I$ if and only if $f \equiv f' \bmod \bar{I}$. 

Because of the lemma, we will always implicitly identify elements of $\mathcal{A}$ to their image in $\bar{\mathcal{A}}$.

Lemma 5.2 *The dimension of $\bar{\mathcal{A}}$ as $\bar{\mathbb{K}}$-vector space is the same as the dimension of $\mathcal{A}$ as $\mathbb{K}$-vector space.*

Proof. Observe that $\bar{\mathcal{A}}$ is generated as $\bar{\mathbb{K}}$-vector space by the monomials $M = \{\mathbf{x}^\alpha | \alpha \in \mathbb{N}^n\}$. Since $M \subset \mathcal{A}$, any generating family of $\mathcal{A}$ as $\mathbb{K}$-vector space also generates $\bar{\mathcal{A}}$ as $\bar{\mathbb{K}}$-vector space. Now, if $a_0, \ldots, a_d$ are $\mathbb{K}$-linearly dependent elements of $\mathcal{A}$, they clearly are $\bar{\mathbb{K}}$-linearly dependent in $\bar{\mathcal{A}}$. Thus the dimension of $\bar{\mathcal{A}}$ does not exceed the one of $\mathcal{A}$.

Now suppose that $\bar{\mathcal{A}}$ has dimension d and let $a_0, \ldots, a_d$ be elements of $\mathcal{A}$. Then, if $f_1, \ldots, f_k$ are generators of I, there exist $\lambda_0, \ldots, \lambda_d \in \bar{\mathbb{K}}$ and $g_1, \ldots, g_k \in \bar{\mathbb{K}}[\mathbf{x}]$ such that

$$\sum_i \lambda_i a_i = \sum_j g_j f_j. \tag{5.4}$$

As in the proof of the previous lemma, $\lambda_0, \ldots, \lambda_d$ and the coefficients of $g_1, \ldots, g_k$ are the solutions of a linear system with coefficient in $\mathbb{K}$, thus they can be taken in $\mathbb{K}$. Hence $a_0, \ldots, a_d$ are $\mathbb{K}$-linearly dependent. 

In what follows, we suppose that $V(I)$ has cardinality d and we denote its points by $\zeta_i \in \bar{\mathbb{K}}^n$ for $1 \leqslant i \leqslant d$.

Proposition 5.3 *The number of points of $V(I)$ equals the dimension of $\bar{\mathcal{A}}$ as vector space.*

Proof. Since $\bar{I}$ is radical and zero-dimensional, its primary decomposition is

$$\bar{I} = Q_1 \cap \cdots \cap Q_d, \tag{5.5}$$

where Q_i is the ideal vanishing on ζ_i.

The Q_i's are maximal and pairwise coprime (i.e. $Q_i + Q_j = \bar{\mathbb{K}}[\mathbf{x}]$ whenever $i \neq j$), hence, by the Chinese remainder theorem,

$$\bar{\mathbb{K}}[\mathbf{x}]/Q_1 \cap \cdots \cap Q_d \cong \bigoplus_{i=1}^{d} \bar{\mathbb{K}}[\mathbf{x}]/Q_i. \tag{5.6}$$

But Q_i is maximal, hence $\bar{\mathbb{K}}[\mathbf{x}]/Q_i$ is an algebraic field extension of $\bar{\mathbb{K}}$. Since $\bar{\mathbb{K}}$ is algebraically closed, $\bar{\mathbb{K}}[\mathbf{x}]/Q_i = \bar{\mathbb{K}}$, and $\bar{\mathcal{A}}$ has dimension d as expected.

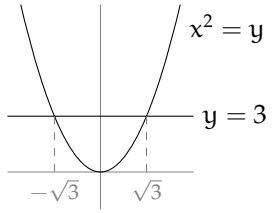


Figure 5.1: Plot in the reals of the ideal $I = (y - 3, x^2 - y)$.

Example 5.4 Consider the ideal $I = (y - 3, x^2 - y)$ of $\mathbb{Q}[x, y]$, a plot is given in Figure 5.1. This ideal is prime and its set of zeros contains no $\mathbb{Q}$-rational points. Since $G = \{y - 3, x^2 - y\}$ is a Gröbner basis for I (for grevlex), elements of $\mathcal{A} = \mathbb{Q}[x, y]/I$ are uniquely represented by their normal form modulo G; for example

$$x^5y + 3xy + 1 \equiv 36x + 1 \mod I.$$

By analyzing the leading monomials of G, it is straightforward to realize that all normal forms modulo G have degree at most 1 in x and degree 0 in y, thus $\mathcal{A}$ has dimension 2 as vector space.

Indeed, the algebraic set $V(I)$ consists of two points:

$$V(I) = \left\{(\sqrt{3}, 3), (-\sqrt{3}, 3)\right\} \subset \bar{\mathbb{Q}}^2.$$

Hence, $\bar{I} = (x - \sqrt{3}, y - 3) \cap (x + \sqrt{3}, y - 3)$ and

$$\bar{\mathcal{A}} \cong \bar{\mathbb{Q}}/(x - \sqrt{3}, y - 3) \oplus \bar{\mathbb{Q}}/(x + \sqrt{3}, y - 3).$$

In particular, the element $36x + 1$ of $\bar{\mathcal{A}}$ is mapped to

$$(1 + 36\sqrt{3}, 1 - 36\sqrt{3})$$

by this isomorphism. The reader will have noticed that $\mathcal{A}$ is isomorphic to $\mathbb{Q}(\sqrt{3})$ as a ring.

We set

$$\bar{\mathcal{A}}_i \stackrel{\text{def}}{=} \bar{\mathbb{K}}[\mathbf{x}]/Q_i, \tag{5.7}$$

then by Eq. (5.6)

$$\bar{\mathcal{A}} = \bigoplus_{i=1}^{d} \bar{\mathcal{A}}_i. \tag{5.8}$$

Now, the $\bar{\mathcal{A}}_i$'s are subalgebras of $\bar{\mathcal{A}}$ isomorphic to $\bar{\mathbb{K}}$. We denote by $\boldsymbol{e}_i$ the unit element of $\bar{\mathcal{A}}_i$, then

$$\begin{aligned} \boldsymbol{e}_i^2 &= \boldsymbol{e}_i, \\ \boldsymbol{e}_i \boldsymbol{e}_j &= 0. \end{aligned} \tag{5.9}$$

Hence $(\boldsymbol{e}_1, \dots, \boldsymbol{e}_d)$ is a basis of $\bar{\mathcal{A}}$ made of orthogonal idempotents.

Example 5.5 Continuing the previous example,

$$\bar{\mathcal{A}}_1 = \bar{\mathbb{Q}}/(x - \sqrt{3}, y - 3) \quad \text{and} \quad \bar{\mathcal{A}}_2 = \bar{\mathbb{Q}}/(x + \sqrt{3}, y - 3).$$

The idempotents are given by

$$\boldsymbol{e}_1 = (3 + \sqrt{3}x)/6 \quad \text{and} \quad \boldsymbol{e}_2 = (3 - \sqrt{3}x)/6.$$

The verification of Eq. (5.9) is straightforward. In particular

$$36x + 1 = (1 + 36\sqrt{3})\boldsymbol{e}_1 + (1 - 36\sqrt{3})\boldsymbol{e}_2.$$

For any $f \in \bar{\mathbb{K}}[\boldsymbol{x}]$, we denote by $f(\zeta_i)$ the evaluation of f at $\zeta_i \in V(I)$. $f(\zeta_i)$ only depends on the class of f in $\bar{\mathcal{A}}$, thus for $a \in \bar{\mathcal{A}}$, we define $a(\zeta_i)$ as the evaluation at ζ_i of an arbitrary representative of the class a.

For any $a \in \bar{\mathcal{A}}$, its class in $\bar{\mathcal{A}}_i$ is $a(\zeta_i)$, by Eq. (5.7). Hence

$$a = \sum_{i=1}^{d} a(\zeta_i)\boldsymbol{e}_i. \tag{5.10}$$

The basis $(\boldsymbol{e}_1, \dots, \boldsymbol{e}_d)$ is a very practical one to represent elements of $\bar{\mathcal{A}}$. Unfortunately, in the general case the idempotents $\boldsymbol{e}_i$ may not be elements of $\mathcal{A}$, as the previous example shows; thus, using such a basis comes at the cost of lifting coefficients in $\bar{\mathbb{K}}$. In order to find a basis better suited to represent elements of $\mathcal{A}$, we shall study the dual of the algebra $\bar{\mathcal{A}}$.

5.2 Trace formulas

We shall denote by $\mathcal{A}^*$ the dual space of $\mathcal{A}$, that is the space of $\mathbb{K}$-linear forms on $\mathcal{A}$. Similarly, we shall denote by $\bar{\mathcal{A}}^*$ the dual space of $\bar{\mathcal{A}}$.

The map

$$\begin{aligned} \mathbf{1}_{\zeta_i} : \bar{\mathcal{A}} &\to \bar{\mathbb{K}} \\ a &\mapsto a(\zeta_i) \end{aligned} \tag{5.11}$$

is linear; in particular

$$\mathbf{1}_{\zeta_i}(\boldsymbol{e}_j) = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases} \tag{5.12}$$

Hence $(\mathbf{1}_{\zeta_1}, \dots, \mathbf{1}_{\zeta_d})$ is the basis of $\bar{\mathcal{A}}^*$ dual to $(\boldsymbol{e}_1, \dots, \boldsymbol{e}_d)$.

The space $\mathcal{A}^*$ has a natural $\mathcal{A}$-module structure under the law $\cdot : \mathcal{A} \times \mathcal{A}^* \to \mathcal{A}^*$ defined by

$$\begin{aligned} a \cdot \ell : \mathcal{A} &\to \mathbb{K} \\ b &\mapsto \ell(ab). \end{aligned} \tag{5.13}$$

Similarly $\bar{\mathcal{A}}^*$ has an $\bar{\mathcal{A}}$-module structure under an analogous law.

Proposition 5.6 *$\bar{A}^*$ and $\bar{A}$ are isomorphic as $\bar{A}$-modules under the mapping $\rho : \boldsymbol{e}_i \mapsto \mathbf{1}_{\zeta_i}$ for $1 \leqslant i \leqslant d$.*

Proof. The mapping is clearly a vector space isomorphism, we only need to prove that it is a morphism of $\bar{A}$-modules. We want to prove that for any $a, b \in \bar{A}$

$$\rho(ab) = a \cdot \rho(b).$$

It suffices to prove this on the bases $(\boldsymbol{e}_1, \ldots, \boldsymbol{e}_d)$ and $(\mathbf{1}_{\zeta_1}, \ldots, \mathbf{1}_{\zeta_i})$.

On the one hand

$$\rho(\boldsymbol{e}_i \boldsymbol{e}_j) = \begin{cases} \rho(0) = 0 & \text{if } i \neq j, \\ \rho(\boldsymbol{e}_i) = \mathbf{1}_{\zeta_i} & \text{if } i = j. \end{cases} \tag{5.14}$$

On the other hand, $\boldsymbol{e}_i \cdot \mathbf{1}_{\zeta_j}$ is the form that associates to any $c \in \bar{A}$ the element

$$(\boldsymbol{e}_i c)(\zeta_j) = \boldsymbol{e}_i(\zeta_j) c(\zeta_j) = \begin{cases} 0 & \text{if } i \neq j, \\ c(\zeta_j) & \text{if } i = j, \end{cases} \tag{5.15}$$

where the last equality comes from (5.12). Hence

$$\boldsymbol{e}_i \cdot \mathbf{1}_{\zeta_j} = \begin{cases} 0 & \text{if } i \neq j, \\ \mathbf{1}_{\zeta_i} & \text{if } i = j. \end{cases} \tag{5.16}$$



Note. We have thus identified $\bar{A}^*$ to $\bar{A}$ as $\bar{A}$-modules, this implies that $\bar{A}$ is a Gorenstein algebra [EM07, Chapter 8]. The theory of Gorenstein algebras is much deeper than the exposition we give here, and giving a complete account of it would be beyond the scope of this document. Nevertheless, we will eventually point out the relationships between the results proven here and the general theory.

Since 1 generates $\bar{A}$ as an $\bar{A}$-module, the form

$$\mathrm{Tr} \overset{\text{def}}{=} \rho(1) = \sum_i \mathbf{1}_{\zeta_i} \tag{5.17}$$

generates $\bar{A}^*$ as an $\bar{A}$-module. $\rho(1)$ will play an important role in the sequel; it is called the *trace form*, the reason for this will be clear in the next section.

The bilinear form on $\bar{A}^* \times \bar{A}$ defined by

$$\langle \ell | a \rangle = \ell(a) \tag{5.18}$$

is non-degenerate by definition (see Section 1.1.3). By means of the isomorphism ρ, we can transport this to a bilinear form on $\bar{A} \times \bar{A}$: we define

$$\langle a | b \rangle = \rho(a)(b). \tag{5.19}$$

By Proposition 5.6, by Eq. (5.13) and by the equality

$$\rho(a)(b) = \sum_i a(\zeta_i) b(\zeta_i), \tag{5.20}$$

we deduce that

$$\langle a|b\rangle = \rho(a)(b) = ab \cdot \mathrm{Tr}(1) = a \cdot \mathrm{Tr}(b) = \mathrm{Tr}(ab) = \langle b|a\rangle . \tag{5.21}$$

is a non-degenerate form on $\bar{A} \times \bar{A}$ that identifies $\bar{A}$ to its dual.

In particular, from Eqs. (5.21) and (5.10) we deduce the *trace formulas* or *interpolation formulas*:

$$a = \sum_{i=1}^{d} \langle a|\boldsymbol{e}_i\rangle\, \boldsymbol{e}_i = \sum_{i=1}^{d} a(\zeta_i)\boldsymbol{e}_i = \sum_{i=1}^{d} a\boldsymbol{e}_i \tag{5.22}$$

Note. In the Gorenstein setting, the forms $\mathbf{1}_{\zeta_i}$ are called the *local residues* at ζ_i and the form $\sum \mathbf{1}_{\zeta_i}$ is called a *global residue*. The non-degeneracy of the global residue implies the $\bar{A}$-isomorphism between $\bar{A}$ and $\bar{A}^*$. The name "residue" comes from complex analysis, because in $\mathbb{C}[\mathbf{x}]/I$ this concept coincides with the classical analytic residue. See [BKL98, EM07].

5.3 Sitckelberger's theorem

Let $a \in \bar{A}$ and consider the linear map

$$M_a : a \mapsto ab. \tag{5.23}$$

Theorem 5.7 (Stickelberger) *The element $\boldsymbol{e}_i$ is an eigenvector of M_a associated to the eigenvalue $a(\zeta_i)$. The characteristic polynomial of M_a is*

$$\prod_{i=1}^{d} (X - a(\zeta_i)).$$

Proof. Using Eqs. (5.22) and (5.9), we have

$$M_a(\boldsymbol{e}_i) = a\boldsymbol{e}_i = \langle a|\boldsymbol{e}_i\rangle\, \boldsymbol{e}_i = a(\zeta_i)\boldsymbol{e}_i. \tag{5.24}$$

Since the $\boldsymbol{e}_i$'s form a basis of $\bar{A}$ as a vector space, M_a is diagonalizable and its eigenvalues are the $a(\zeta_i)$'s, each counted once. 

We define the *trace* and the *norm* of an element of $\bar{A}$ in the same way as they are defined for elements of extension fields.

Definition 5.8 (Trace, norm) We define the *trace* of a as

$$\mathrm{Tr}(a) = \mathrm{Tr}(M_a)$$

and its *norm* as

$$N(a) = \det(M_a).$$

Then, the following corollary is easily derived.

Corollary 5.9 *One has*

$$\mathrm{Tr}(a) = \sum_{i=1}^{d} a(\zeta_i), \tag{5.25}$$

$$N(a) = \prod_{i=1}^{d} a(\zeta_i). \tag{5.26}$$

By Eqs. (5.25) and (5.17), it is clear that $\mathrm{Tr}(a) = \rho(1)(a)$, which justifies the notation we employed in the last section.

Theorem 5.10 *$\mathcal{A}^*$ is isomorphic to $\mathcal{A}$ as $\mathcal{A}$-module under the restriction of ρ to $\mathcal{A}$.*

Proof. When $a, b \in \mathcal{A}$, the characteristic polynomial of M_{ab} has coefficients in $\mathbb{K}$. Thus $\langle a|b\rangle = \mathrm{Tr}(ab)$ is in $\mathbb{K}$, and the restriction of $\rho(a)$ to $\mathcal{A}$ is in $\mathcal{A}^*$.

Consider now the quadratic form on $\bar{\mathcal{A}}$

$$q(a) = \mathrm{Tr}(a^2). \tag{5.27}$$

Its matrix in the basis $(\boldsymbol{e}_1, \ldots, \boldsymbol{e}_d)$ is the identity matrix, thus it has rank d. Now let $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_d)$ be a basis of $\mathcal{A}$, then it is a basis of $\bar{\mathcal{A}}$ too. The matrix of q on $\mathbf{B}$ has coefficients in $\mathbb{K}$ and rank d, thus it also is the matrix of the restriction of q to $\mathcal{A}$.

Hence, q is non degenerate on $\mathcal{A}$, and so is $\langle a|b\rangle$. We deduce that $a \in \mathcal{A}$ equals 0 if and only if $\rho(a) \in \mathcal{A}^*$ equals 0. To conclude it suffices to observe that $\mathcal{A}^*$ and $\mathcal{A}$ have the same dimension as $\mathbb{K}$-vector spaces. ∎

5.4 Rational Univariate Representation

In many circumstances it is useful to switch from a multivariate representation of the elements of $\mathcal{A}$ to an univariate one. A *rational univariate representation* (RUR) [Rou99], sometimes also called *geometric resolution* [GLS01], of $\mathbb{K}[x_1, \ldots, x_n]/I$ consists in expressing $V(I)$ as the solution of the system

$$\begin{aligned} f(t) &= 0, \\ x_1 &= \frac{g_1(t)}{g(t)}, \\ &\vdots \\ x_n &= \frac{g_n(t)}{g(t)}, \end{aligned} \tag{5.28}$$

where t is a new variable and $f, g, g_1, \ldots, g_n$ are univariate polynomials with coefficients in $\mathbb{K}$.

Lemma 5.11 *Let $t \in \mathcal{A}$ and let Q be its characteristic polynomial. Let T be a fresh variable, then*

$$\sum_{i \geqslant 0} \frac{\langle 1|t^i\rangle}{T^{i+1}} = \frac{Q'(T)}{Q(T)}. \tag{5.29}$$

Proof. By the trace formulas (5.22)

$$t^i = \sum_{j=1}^{d} \langle t^i|\boldsymbol{e}_j\rangle\, \boldsymbol{e}_j, \tag{5.30}$$

hence

$$\sum_{i \geqslant 0} \frac{\langle 1|t^i\rangle}{T^{i+1}} = \sum_{i \geqslant 0} \sum_{j=1}^{d} \frac{\langle 1|\boldsymbol{e}_j\rangle \langle t^i|\boldsymbol{e}_j\rangle}{T^{i+1}} = \sum_{i \geqslant 0} \sum_{j=1}^{d} \frac{t(\zeta_j)^i}{T^{i+1}}. \tag{5.31}$$

Swapping the sums, this equals

$$\sum_{j=1}^{d} \frac{1}{T - t(\zeta_j)} = \frac{\sum_{j=1}^{d} \prod_{j' \neq j} (T - t(\zeta_j))}{\prod_{j=1}^{d} (T - t(\zeta_j))} = \frac{Q'(T)}{Q(T)}, \tag{5.32}$$

where the last equality comes from Theorem 5.7. 

Remark 5.12. If $\mathbb{K}$ has characteristic 0, the polynomial Q can be recovered from its logarithmic derivative via the formula

$$Q = \exp\left(\int \frac{Q'}{Q}\right). \tag{5.33}$$

When the degree d is known in advance, this suggests an efficient algorithm to compute Q, provided the characteristic of $\mathbb{K}$ is 0 or larger than d.

We know that $\mathrm{Tr}(1) = d$, hence

$$\frac{Q'}{Q} = \frac{d}{T} + \sum_{i \geqslant 1} \frac{\langle 1 | t^i \rangle}{T^{i+1}}. \tag{5.34}$$

We deduce

$$Q = \exp\left(d \log T + \int \sum_{i \geqslant 1} \frac{\langle 1 | t^i \rangle}{T^{i+1}}\right) = T^d \exp\left(-\sum_{i \geqslant 1} \frac{\langle 1 | t^i \rangle}{i T^i}\right), \tag{5.35}$$

then the power series on the right hand side can be exponentiated using a Newton iteration. But Q is a polynomial of degree d, hence we can truncate the exponent power series to the order $O(T^{-d-1})$.

In conclusion, it is sufficient to know

$$\mathrm{Tr}(t), \ldots, \mathrm{Tr}(t^d) \tag{5.36}$$

in order to compute Q.

Example 5.13 Continuing Example 5.5, we want to compute the characteristic polynomial of $t = 36x + 1$. We know that

$$t = 36x + 1 = (1 + 36\sqrt{3})e_1 + (1 - 36\sqrt{3})e_2,$$

hence its traces are easily computed :

$$\mathrm{Tr}(t) = (1 + 36\sqrt{3}) + (1 - 36\sqrt{3}) = 2,$$
$$\mathrm{Tr}(t^2) = (1 + 36\sqrt{3})^2 + (1 - 36\sqrt{3})^2 = 7778.$$

We compute the exponential:

$$\exp\left(-\frac{2}{T} - \frac{3889}{T^2} + O(T^{-3})\right) = \left(1 - \frac{2}{T} + \frac{4}{2!T^2} + O(T^{-3})\right)\left(1 - \frac{3889}{T^2} + O(T^{-3})\right) = \left(1 - \frac{2}{T} - \frac{3887}{T^2} + O(T^{-3})\right),$$

hence the characteristic polynomial is

$$T^2 - 2T - 3887.$$

Being able to compute characteristic polynomials is not enough to find a rational univariate representation. Indeed, the element t may not generate $\mathcal{A}$ as a $\mathbb{K}$-algebra, and thus not every element of $\mathcal{A}$ could be represented as a rational function of t. We now give a criterion to find elements that generate $\mathcal{A}$.

Definition 5.14 (Separating element) An element $t \in \mathcal{A}$ is said to be *separating* if for any $\zeta, \zeta' \in V(I)$

$$t(\zeta) \neq 0 \qquad \text{and} \qquad \zeta \neq \zeta' \Rightarrow t(\zeta) \neq t(\zeta').$$

Separating elements always exist, provided $\mathcal{A}$ is large enough. We do not give here any proof of this fact because in the applications we have in mind a separating element is always at hand.

Proposition 5.15 *Let t be a separating element, then $1, t, \dots, t^{d-1}$ are $\bar{\mathbb{K}}$-linearly independent.*

Proof. Let $\sum_{i=0}^{d-1} a_i t^i = 0$, then the polynomial $\sum_{i=0}^{d-1} a_i T^i$ has d roots in $\bar{\mathbb{K}}$, namely $t(\zeta_i)$ for $1 \leqslant i \leqslant d$, hence it is identically null.

Thanks to this proposition and to Lemma 5.11, we have a way to find the first line of the representation in Eq. (5.28), provided that we know a separating element t. We now have to express $x_1, \dots, x_n$ as functions of the roots of the minimal polynomial of t.

Theorem 5.16 *Let t be a separating element of $\mathcal{A}$ and let Q be its minimal polynomial. Let $a \in \mathcal{A}$ and set*

$$A(T) = Q(T) \sum_{i \geqslant 0} \frac{\langle a | t^i \rangle}{T^{i+1}}. \tag{5.37}$$

Then $A(T)$ is a polynomial of degree less than d, and

$$a = \frac{A(t)}{Q'(t)}. \tag{5.38}$$

Proof. We develop the series as in the proof of Lemma 5.11:

$$\sum_{i \geqslant 0} \frac{\langle a | t^i \rangle}{T^{i+1}} = \sum_{j=1}^{d} a(\zeta_j) \sum_{i \geqslant 0} \frac{t(\zeta_j)^i}{T^{i+1}} = \frac{\sum_{j=1}^{d} a(\zeta_j) \prod_{j' \neq j} (T - t(\zeta_j))}{Q(T)}. \tag{5.39}$$

Hence $A(T)$ is a polynomial of degree less than d.

Now we use the trace formulas to decompose $A(t)$ and $Q'(t)$:

$$\langle A(t) | e_i \rangle = \sum_{j=1}^{d} a(\zeta_j) \prod_{j' \neq j} (\langle t | e_i \rangle - t(\zeta_j)) = a(\zeta_i) \prod_{j \neq i} (t(\zeta_i) - t(\zeta_j)), \tag{5.40}$$

$$\langle Q'(t) | e_i \rangle = \prod_{j \neq i} (t(\zeta_i) - t(\zeta_j)). \tag{5.41}$$

Because t is separating, $\langle Q'(t) | e_i \rangle \neq 0$ for any i, hence $Q'(t)$ is a unit of $\mathcal{A}$. We deduce that

$$\left\langle \frac{A(t)}{Q'(t)} \middle| e_i \right\rangle = a(\zeta_i) \tag{5.42}$$

for any i. Hence, by the trace formulas

$$\frac{A(t)}{Q'(t)} = \sum_i \left\langle \frac{A(t)}{Q'(t)} \middle| e_i \right\rangle e_i = \sum_i a(\zeta_i) e_i = a. \tag{5.43}$$



By taking $a = x_i$, the theorem can be used to find a rational univariate representation: it suffices to know

$$\mathrm{Tr}(x_i), \mathrm{Tr}(x_i t), \ldots, \mathrm{Tr}(x_i t^{d-1}) \tag{5.44}$$

in order to deduce $g_i(T)$ as in the representation (5.28).

Example 5.17 We conclude the previous example. We want to find a parameterization of x and y with respect to $t = 36x + 1$. We already know the minimal polynomial of t:

$$Q(T) = T^2 - 2T - 3887,$$

thus

$$Q'(T) = 2T - 2.$$

Now

$$x = \sqrt{3}e_1 - \sqrt{3}e_2 \qquad y = 3e_1 + 3e_2,$$

hence

$$\begin{aligned} &\mathrm{Tr}(x) = 0, && \mathrm{Tr}(xt) = 216, \\ &\mathrm{Tr}(y) = 6, && \mathrm{Tr}(yt) = 6. \end{aligned}$$

We deduce that

$$x = \frac{216}{2t-2}, \qquad y = \frac{6t-6}{2t-2}.$$

Note. Theorem 5.16 was introduced in [ABRW96]. It was used by Rouiller [Rou99] to give an explicit algorithm to compute a rational univariate representation of an arbitrary zero-dimensional ideal. This algorithm requires to have a monomial basis for the vector space $\mathcal{A}$, thus in practice it computes a rational univariate representation starting from a Gröbner basis.

A completely different approach based on Noether's normalization theorem, called geometric resolution [GLS01], gives a Gröbner-basis-free alternative for computing rational univariate representations.

5.5 The univariate case

In this section we shall see that, in the particular case of ideals of the univariate polynomial ring $\mathbb{K}[x]$, trace formulas reduce to Lagrange interpolation and Lemma 5.11 reduces to Newton's identities.

Any ideal I of $\mathbb{K}[x]$ is principal. Let f be a monic generator of I and let $\zeta_1, \ldots, \zeta_d$ be its roots in $\bar{\mathbb{K}}$, then

$$\bar{\mathbb{K}}[x]/(f) = \bigoplus_{i=1}^{d} \bar{\mathbb{K}}[x]/(x - \zeta_i). \tag{5.45}$$

Hence, the orthogonal idempotents are given by

$$e_i = \prod_{j \neq i} \frac{x - \zeta_j}{\zeta_i - \zeta_j}. \tag{5.46}$$

By definition

$$\langle a | e_i \rangle = a(\zeta_i), \tag{5.47}$$

hence the trace formulas rewrite

$$a = \sum_{i=1}^{d} a(\zeta_i) \prod_{j \neq i} \frac{x - \zeta_i}{\zeta_i - \zeta_j}, \tag{5.48}$$

which is exactly the formula of Lagrange interpolation (see Eq. (2.24)).

Now we want to compute the Newton sums of f, i.e. the values

$$p_i = \zeta_1^i + \cdots + \zeta_d^i. \tag{5.49}$$

x is a root of f in $\mathcal{A}$ and it clearly separates $V(I)$, then by Lemma 5.11

$$\frac{f'}{f} = \sum_{i \geqslant 0} \frac{\langle 1 | x^i \rangle}{T^{i+1}} = \sum_{i \geqslant 0} \frac{\sum_{j=1}^{d} \zeta_j^i}{T^{i+1}} = \sum_{i \geqslant 0} \frac{p_i}{T^{i+1}}. \tag{5.50}$$

Hence the Newton sums can be recovered as coefficients of the power series. Inversely, the coefficients of f can be computed from its Newton sums using Newton's identities; notice, however, that it is more efficient to use Remark 5.12 for this.

5.6 Shoup's algorithm

We have seen that at the heart of the rational univariate representation is the computation of the coefficients of the power series

$$\sum_{i \geqslant 0} \frac{\langle a | t^i \rangle}{T^{i+1}} \tag{5.51}$$

up to a certain precision. In this section we shall find an efficient way to compute such truncated series.

Consider the univariate polynomial ring $\mathbb{K}[T]$ and identify its dual space $\mathbb{K}[T]^*$ to $\mathbb{K}[[1/T]]$ via the bilinear form

$$\langle \alpha | f \rangle = [\alpha f]_0, \tag{5.52}$$

where $\alpha \in \mathbb{K}[[1/T]]$, $f \in \mathbb{K}[T]$ and $[\beta]_i$ is the coefficient of T^i in β. So that

$$\left\langle \sum_{i \geqslant 0} \frac{\alpha_i}{T^i} \middle| \sum_{j=0}^{n} f_j T^j \right\rangle = \sum_{i \geqslant 0} \alpha_i f_i. \tag{5.53}$$

For a $t \in \mathcal{A}$, consider the evaluation map at t

$$\begin{aligned} \mathrm{ev}_t : \mathbb{K}[T] &\to \mathcal{A}, \\ g &\mapsto g(t). \end{aligned} \tag{5.54}$$

Lemma 5.18 *For any $t \in \mathcal{A}$, the dual map ev_t^* with respect to the bilinear forms defined in Eqs.* (5.19) *and* (5.54) *is such that*

$$\mathrm{ev}_t^*(a) = \sum_{i \geqslant 0} \frac{\langle a | t^i \rangle}{T^i}. \tag{5.55}$$

Proof. ev_t and ev_t^* are clearly linear maps, thus it suffices to show the identity on the basis $\{1, T, T^2, \ldots\}$ of $\mathbb{K}[T]$.

$$\langle a | \mathrm{ev}_t(T^j) \rangle = \langle a | t^j \rangle = \left\langle \sum_{k \geqslant 0} \frac{\langle a | t^k \rangle}{T^k} \middle| T^j \right\rangle = \langle \mathrm{ev}_t^*(a) | T^j \rangle . \tag{5.56}$$



Thus, applying the techniques of Section 3.3, from any $\mathbb{K}$-algebraic transform to compute ev_t for a fixed t we can deduce a transform to compute ev_t^* that has the same time and space complexity. Furthermore, from a generic $\mathbb{K}$-algebraic algorithm to evaluate polynomials in $\mathbb{K}[T]$ at points of $\mathcal{A}$, we can deduce an algorithm to compute ev_t^* for any t, having the same time complexity and possibly a penalty in space complexity.

However, for a given basis of $\mathcal{A}$, it may be difficult to find the corresponding dual basis with respect to $\langle | \rangle$. In order to give an explicit algorithm, we work with the form

$$\langle \ell | a \rangle = \ell(a) \tag{5.57}$$

on $\mathcal{A}^* \times \mathcal{A}$, instead.

Lemma 5.19 *Let $a \in \mathcal{A}$ and consider the map $M_a : b \mapsto ab$. The dual map M_a^* with respect to the form* (5.57) *is such that*

$$M_a^*(\ell) = a \cdot \ell. \tag{5.58}$$

Proof. The verification is straightforward:

$$\langle \ell | M_a(b) \rangle = \langle \ell | ab \rangle = \ell(ab) = \langle a \cdot \ell | b \rangle . \tag{5.59}$$



Hence, using Principle 3.34, any multiplication algorithm for a given basis $\mathbf{B}$ of $\mathcal{A}$ can be transposed to compute $a \cdot \ell$ given a on $\mathbf{B}$ and ℓ on $\mathbf{B}^*$. Transposed multiplication is a classic problem, it can be solved without significant losses in space complexity on most bases, see Section 2.2.8 and [Sho95, Sho99, BLS03, HQZ04, PS06].

We now consider again the dual map of ev_t, this time with respect to (5.57); we denote it by proj_t to avoid confusion with (5.55).

Lemma 5.20 *For any $t \in \mathcal{A}$, the map proj_t is such that*

$$\mathrm{proj}_t(\ell) = \sum_{i \geqslant 0} \frac{\ell(t^i)}{T^i}. \tag{5.60}$$

Algorithm 5.1: RUR

Input : A basis $\mathbf{B}$ of $\mathcal{A}$, $a_1, \ldots, a_n \in \mathcal{A}^{\mathbf{B}}$, t separating, $\mathrm{Tr} \in (\mathcal{A}^*)^{\mathbf{B}^*}$.
Output : A rational univariate representation of $a_1, \ldots, a_n, t$.

1: Compute $\frac{Q'}{Q} = \frac{1}{T}\,\mathrm{proj}_t(\mathrm{Tr})$;
2: Compute $Q = \exp\left(\int \frac{Q'}{Q}\right)$;
3: for all a_i do
4: Compute $A_i(T) = \frac{Q(T)}{T} \cdot (\mathrm{proj}_t \circ M^*_{a_i}(\mathrm{Tr})) \bmod T^d$;
5: Output $Q(T)$ and $\frac{A_1(T)}{Q'(T)}, \ldots, \frac{A_n(T)}{Q'(T)}$.

Proof. This is just a consequence of Lemma 5.18, since there exists an $a \in \mathcal{A}$ such that $\ell = \rho^{-1}(a)$. 

The problem of computing proj_t is often known as *power projection*. Using again Principle 3.34, any algorithm to evaluate polynomials in $\mathbb{K}[T]$ on t can be transposed to a power projection algorithm on t. We will see some instances of power projection algorithms later on.

Then, the proof of the following theorem is evident.

Theorem 5.21 *Let $a, t \in \mathcal{A}$, let $\mathrm{Tr} \in \mathcal{A}^*$ be the trace form, then*

$$\sum_{i \geqslant 0} \frac{\langle a | t^i \rangle}{T^i} = \mathrm{proj}_t \circ M^*_a(\mathrm{Tr}). \tag{5.61}$$

From it, we can derive an algorithm to compute a rational univariate representation, provided we know the coordinates of the linear form Tr.

In order to apply this algorithm, prior knowledge of the expression of Tr in the basis $\mathbf{B}^*$, i.e. $\{\mathrm{Tr}(b) | b \in \mathbf{B}\}$, is needed. When B is a polynomial basis $\{1, b, \ldots, b^d\}$, and the minimal polynomial of b is known in advance, this can be obtained using Lemma 5.11.

Remark 5.22. In the univariate case $\mathcal{A} = \mathbb{K}[x]/(f)$, $t \in \mathcal{A}$ is expressed in the basis $(1, x, \ldots, x^{d-1})$ as a polynomial in x modulo f. Then,

$$\mathrm{ev}_t(g) = g(t) = g \circ t \bmod f,$$

where $g \circ t$ is polynomial composition. The problem of computing $g \circ t \bmod f$ for $g, t, f \in \mathbb{K}[x]$ is modular composition; as in Section 2.1, its complexity is denoted by $C(d)$, where $d = \deg f$. A naive algorithm gives $C(d) \in O(d^2)$.

As we saw in Section 2.2.4, the most efficient algorithms for modular composition are Brent and Kung's [BK78], having complexity $O(M(d)\sqrt{d} + d^{(\omega+1)/2})$, and Kedlaya and Umans' [Uma08, KU08], having quasi-linear complexity. Each of these has a dual algorithm solving power projection with the same complexity [Sho94, KU08], thus, at least in the univariate case, power projection can be solved in subquadratic time. Some extensions to the bivariate and multivariate cases also exist [Sho99, KU08].

Remark 5.23. Algorithm 5.1 first appeared in [BSS03], which combined the ideas of [Rou99] and [Sho94, Sho95, Sho99]. Lemma 5.20 was first used in [Sho94] to compute minimal polynomials of elements of a residue class field $\mathbb{K}[x]/(f)$, based on a transposed modular composition algorithm. The method was extended to

the bivariate case in [Sho99]. A review of the methods of [Sho94, Sho95, Sho99] can be found in [Kal00, §6].

In [Sho94, Sho99], a generic linear form ℓ is taken, so that by Theorem 5.16

$$\sum_{i \geqslant 0} \frac{\ell(t^i)}{T^{i+1}} = \frac{A(T)}{Q(T)}$$

for some $A(T)$. Since ℓ is arbitrary, $A(T)$ is *a priori* unknown, thus one cannot use Remark 5.12 to recover Q. Instead, a rational fraction reconstruction algorithm (see Section 2.2.6) is used to recover both $A(T)$ and $Q(T)$. However, in comparison to Remark 5.12, this requires to compute twice as many "power projections".

The intuition for the method, and for the name "power projection", comes from Wiedemann's method to solve sparse linear systems [Wie86]. The idea is that the minimal polynomial of a black-box matrix A is the same as the one of the linear recurrent sequence

$$1, A, A^2, \ldots$$

Then, for any linear form ℓ, the minimal polynomial of the sequence

$$\ell(1), \ell(A), \ell(A^2), \ldots$$

divides the minimal polynomial of A. The algorithm takes a random form ℓ and uses it to "project" the first $2d$ powers of A onto $\mathbb{K}$, then recovers its minimal polynomial using the Berlekamp-Massey algorithm [Mas03]. On the equivalence between the Berlekamp-Massey algorithm and the rational fraction reconstruction, see [Dor87].

5.7 From univariate to bivariate and back again

Algorithm RUR can be used as an efficient change of basis algorithm. Let $\mathbf{B}$ be any basis for $\mathcal{A}$, let t be a separating element, and let the coordinates of Tr on $\mathbf{B}^*$ be known.

Proposition 5.24 *Let $M_{\mathbf{B}}$ be the cost of multiplication in the basis $\mathbf{B}$ and let $E_{\mathbf{B}}$ be the cost of the change of basis from*

$$\mathbf{T} = (1, t, \ldots, t^{d-1}) \tag{5.62}$$

to $\mathbf{B}$. Then, the cost of the change of basis from $\mathbf{B}$ to $\mathbf{T}$ is

$$M_{\mathbf{B}} + E_{\mathbf{B}} + O(M(d)), \tag{5.63}$$

plus a precomputation cost of

$$E_{\mathbf{B}} + O(M(d) \log d), \tag{5.64}$$

where $M(d)$ is the cost of polynomial multiplication, as usual.

The algorithm follows immediately from the observation that the change of basis map from $\mathbf{T}$ to $\mathbf{B}$ is equivalent to the map

$$\begin{aligned} ev_t : \mathbb{K}[T] &\to \mathcal{A}^{\mathbf{B}}, \\ g &\mapsto g(t). \end{aligned} \tag{5.65}$$

Then algorithm RUR on inputs $a \in \mathcal{A}^{\mathbf{B}}$ and Tr, outputs the expression of a in the basis $\mathbf{T}$. We now prove the complexity estimates (5.63) and (5.64).

Proof of Proposition 5.24. By Principle 3.34, the call to proj_t at step 1 of RUR has the same cost as $E_{\mathbf{B}}$. Then the cost of step 2 is $O(M(d))$ using a Newton iteration. Both steps can be done just once, thus they contribute to (5.64).

Again by Principle 3.34, step 4 costs

$$M_{\mathbf{B}} + E_{\mathbf{B}} + M(d), \tag{5.66}$$

where the first term comes from transposed multiplication, the second one from the power projection and the third one from multiplication by Q.

Finally, in step 5 instead of expressing a as a rational fraction, we need to invert Q' modulo Q and multiply A by the result. The inversion costs $O(M(d)\log d)$ by extended GCD, but can be done just once, thus it contributes to (5.64); the multiplication costs $O(M(d))$ by polynomial multiplication and Newton inversion.



Chapter 6

Artin-Schreier towers

In this chapter we give fast algorithms for arithmetic operations in Artin-Schreier towers. Prior results for this task are due to Cantor [Can89] and Couveignes [Cou00]. However, the algorithms of [Cou00] need as a prerequisite a fast multiplication algorithm in some towers of a special kind, called "Cantor towers" in [Cou00]. Such an algorithm is unfortunately not in the literature, making the results of [Cou00] non practical. This chapter fills the gap.

To our knowledge, no previous work provided the missing ingredients to put Couveignes' algorithms to practice. Part of Cantor's results were independently discovered by Wang and Zhu [WZ88] and have been extended in another direction (fast polynomial multiplication over arbitrary finite fields) by von zur Gathen and Gerhard [vzGG02] and Mateer [Mat08].

Technically, the main algorithmic contribution of this chapter is a fast change-of-basis algorithm based on the techniques of Chapter 5. Building on this, it is possible to obtain fast multiplication routines, and by extension completely explicit versions of all algorithms of [Cou00]. Along the way, we also extend constructions of Cantor to the case of a general finite base field $\mathbb{U}_0$, where Cantor had $\mathbb{U}_0 = \mathbb{F}_p$.

The algorithms presented in this chapter have been integrated in a `C++` library called `FAAST`, based on Shoup's `NTL` [Sho03] library. This chapter is joint work with Schost [DFS09].

6.1 Introduction

If $\mathbb{U}$ is a field of characteristic p, polynomials of the form

$$X^p - X - \alpha, \tag{6.1}$$

with $\alpha \in \mathbb{U}$ are called *Artin-Schreier polynomials*; a field extension $\mathbb{U}'/\mathbb{U}$ is *Artin-Schreier* if it is of the form $\mathbb{U}' = \mathbb{U}[X]/P$, with P an Artin-Schreier polynomial.

An *Artin-Schreier tower* of height k is a sequence of Artin-Schreier extensions $\mathbb{U}_i/\mathbb{U}_{i-1}$, for $1 \leqslant i \leqslant k$; it is denoted by $(\mathbb{U}_0, \ldots, \mathbb{U}_k)$. In what follows, we only consider extensions of finite degree over $\mathbb{F}_p$. Thus, $\mathbb{U}_i$ is of degree p^i over $\mathbb{U}_0$, and of degree $p^i d$ over $\mathbb{F}_p$, with $d = [\mathbb{U}_0 : \mathbb{F}_p]$.

The importance of this concept comes from the fact that all Galois extensions of degree p are Artin-Schreier (see [Lan02, VI, §6]). As such, they arise frequently, e.g., in number theory (for instance, when computing p^k-torsion groups of Abelian varieties over $\mathbb{F}_p$). The need for fast arithmetic in these towers is motivated in particular by applications to isogeny computation and point-counting in cryptography, as we will see in Chapter 8.

We count all time complexities in number of operations in $\mathbb{F}_p$. Then, notation being as before, optimal algorithms in $\mathbb{U}_k$ would have complexity $O(p^k d)$; most of our results are (up to logarithmic factors) of the form $O(p^{k+\alpha} d^{1+\beta})$, for small constants α, β such as $0, 1, 2$ or 3.

For several operations, different algorithms will be available, and their relative efficiencies can depend on the values of p, d and k. In these situations, we always give details for the case where p is small, since cases such as $p = 2$ or $p = 3$ are especially useful in practice.

In the following section we give the relevant definitions and preliminaries. In Section 6.3, we define a specific Artin-Schreier tower, where arithmetic operations will be fast. Our key change-of-basis algorithm for this tower is in Section 6.4. In Sections 6.5 and 6.6, we revisit Couveignes' algorithm for isomorphisms between Artin-Schreier towers [Cou00] in our context, which yields fast arithmetics for *any* Artin-Schreier tower. Finally, Section 6.7 presents our implementation of the `FAAST` library and gives experimental results.

6.2 Preliminaries

As a general rule, variables and polynomials are in upper case; elements algebraic over $\mathbb{F}_p$ (or some other field, that will be clear from the context) are in lower case.

6.2.1 Element representation

We let Q_0 be in $\mathbb{F}_p[X_0]$ and $\mathbb{U}_0 = \mathbb{F}_p[X_0]/Q_0$.

Definition 6.1 Let $(G_i)_{0 \leqslant i < k}$ be a sequence of polynomials over $\mathbb{F}_p$, with G_i in $\mathbb{F}_p[X_0, \ldots, X_i]$. The sequence $(G_i)_{0 \leqslant i < k}$ is said to *define the tower* $(\mathbb{U}_0, \ldots, \mathbb{U}_k)$ if for $i \geqslant 0$, $\mathbb{U}_i = \mathbb{F}_p[X_0, \ldots, X_i]/K_i$, where K_i is the ideal generated by

$$\left| \begin{array}{l} P_i = X_i^p - X_i - G_{i-1}(X_0, \ldots, X_{i-1}) \\ \quad \vdots \\ P_1 = X_1^p - X_1 - G_0(X_0) \\ Q_0(X_0) \end{array} \right. \tag{6.2}$$

in $\mathbb{F}_p[X_0, \ldots, X_i]$, and if $\mathbb{U}_i$ is a field.

The residue class of X_i (resp. G_{i-1}) in $\mathbb{U}_i$, and thus in $\mathbb{U}_{i+1}, \ldots$, is written x_i (resp. γ_{i-1}), so that we have $x_i^p - x_i = \gamma_{i-1}$.

Finding a suitable $\mathbb{F}_p$-basis to represent elements of a tower $(\mathbb{U}_0, \ldots, \mathbb{U}_k)$ is a crucial question.

Definition 6.2 (Multivariate basis) If $d = \deg(Q_0)$, the *multivariate basis* $\mathbf{B}_i$ of $\mathbb{U}_i$ is

$$\mathbf{B}_i = \{x_0^{e_0} \cdots x_i^{e_i} \mid 0 \leqslant e_0 < d,\ 0 \leqslant e_j < p \text{ for } j > 0\}. \tag{6.3}$$

However, in this basis, we do not have very efficient arithmetic operations, starting from multiplication. Indeed, the natural approach to multiplication in $\mathbf{B}_i$ consists in a polynomial multiplication, followed by reduction modulo $(Q_0, P_1, \ldots, P_i)$; however, the initial product gives a polynomial of partial degrees $(2d-2, 2p-2, \ldots, 2p-2)$, so the number of monomials appearing is not linear in $[\mathbb{U}_i : \mathbb{F}_p] = p^i d$. See [LMMS07] for details.

As a workaround, we introduce the notion of a *primitive tower*.

Definition 6.3 (Primitive tower) With the same notation of Definition 6.1, a tower $(\mathbb{U}_0, \ldots, \mathbb{U}_k)$ defined by $(G_i)_{0 \leqslant i < k}$ is said to be *primitive* if

$$\mathbb{U}_i = \mathbb{F}_p[x_i] \tag{6.4}$$

for all i. In this case, we let $Q_i \in \mathbb{F}_p[X]$ be the minimal polynomial of x_i, of degree $p^i d$.

Definition 6.4 (Univariate basis) Let $(\mathbb{U}_0, \ldots, \mathbb{U}_k)$ be a primitive tower, the *univariate basis* $\mathbf{C}_i$ of $\mathbb{U}_i$ is the $\mathbb{F}_p$-basis

$$\mathbf{C}_i = \{x_i^a \mid 0 \leqslant a < p^i d\}. \tag{6.5}$$

To stress the fact that $v \in \mathbb{U}_i$ is represented in the univariate basis, we write $v \in \mathbb{U}_i$ using a bold "belongs to" sign.

In a primitive tower, unless otherwise stated, we represent the elements of $\mathbb{U}_i$ in the univariate basis. In this basis, assuming Q_i is known, additions and subtractions are done in $p^i d$ operations, multiplications in $O(M(p^i d))$ operations and inversions in $O(M(p^i d)\log(p^i d))$ operations (see Section 2.2).

Note that having fast arithmetic operations in $\mathbb{U}_i$ enables us to write fast algorithms for polynomial arithmetic in $\mathbb{U}_i[Y]$, where Y is a new variable. Extending the previous notation, let us write $A \in \mathbb{U}_i[Y]$ to indicate that a polynomial $A \in \mathbb{U}_i[Y]$ is written in the basis $(x_i^\alpha Y^\beta)_{0 \leqslant \alpha < p^i d, 0 \leqslant \beta}$ of $\mathbb{U}_i[Y]$. Then, given $A, B \in \mathbb{U}_i[Y]$, both of degrees less than n, one can compute $AB \in \mathbb{U}_i[Y]$ in $O(M(p^i dn))$ operations using Kronecker's substitution (see Section 2.2.7).

One can extend the fast Euclidean division algorithm to this context, as Newton iteration reduces Euclidean division to polynomial multiplication (see Section 2.2.3). This implies that Euclidean division of a degree n polynomial $A \in \mathbb{U}_i[Y]$ by a monic degree m polynomial $B \in \mathbb{U}_i[Y]$, with $m \leqslant n$, can be done in $O(M(p^i dn))$ operations.

Finally, fast GCD techniques carry over as well, as they are based on multiplication and division. As we saw in Section 2.2.6, the extended GCD of two monic polynomials $A, B \in \mathbb{U}_i[Y]$ of degree at most n can be computed in $O(M(p^i dn\log(n)))$ operations.

6.2.2 Trace and pseudotrace

We continue with a few useful facts on traces.

Proposition 6.5 *We have the following well-known properties:*

$$\mathrm{Tr}_{\mathbb{F}_{q^n}/\mathbb{F}_q} : a \mapsto \textstyle\sum_{\ell=0}^{n-1} a^{q^\ell}, \tag{$\mathbf{P}_1$}$$

$$\mathrm{Tr}_{\mathbb{F}_{q^{mn}}/\mathbb{F}_q} = \mathrm{Tr}_{\mathbb{F}_{q^m}/\mathbb{F}_q} \circ \mathrm{Tr}_{\mathbb{F}_{q^{mn}}/\mathbb{F}_{q^m}}. \tag{$\mathbf{P}_2$}$$

Proof. This is a direct consequence of Proposition 1.4. 

Proposition 6.6 *If $\mathbb{U}'/\mathbb{U}$ is an Artin-Schreier extension generated by a polynomial Q and x is a root of Q in $\mathbb{U}'$, then*

$$\mathrm{Tr}_{\mathbb{U}'/\mathbb{U}}(x^j) = 0 \text{ for } j < p-1; \quad \mathrm{Tr}_{\mathbb{U}'/\mathbb{U}}(x^{p-1}) = -1. \tag{$\mathbf{P}_3$}$$

Proof. This is a consequence of Lemma 5.11. In fact

$$\sum_{i\geqslant 0}\frac{\mathrm{Tr}(x^i)}{T^{i+1}}=\frac{Q'(T)}{Q(T)}=-\frac{1}{T^p-T-\alpha}, \tag{6.6}$$

from which we deduce

$$\mathrm{Tr}(x^i)=0 \qquad \text{for } 0\leqslant i<p-1, \tag{6.7}$$

$$\mathrm{Tr}(x^{p-1})-\mathrm{Tr}(1)=\mathrm{Tr}(x^{p-1})=-1, \tag{6.8}$$

$$\mathrm{Tr}(x^{i+p})-\mathrm{Tr}(x^{i+1})-\alpha\,\mathrm{Tr}(x^i)=0 \qquad \text{for } i\geqslant 0. \tag{6.9}$$



Proposition 6.7 *The Artin-Schreier polynomial $X^p-X-\alpha$ is irreducible in $\mathbb{F}_q$ if and only if*

$$\mathrm{Tr}_{\mathbb{F}_q/\mathbb{F}_p}(\alpha)\neq 0. \tag{6.10}$$

If it is reducible, then it is split and its roots are

$$\eta,\eta+1,\dots,\eta+p-1. \tag{6.11}$$

Proof. This proof is from [LN96, Chapter 2].

The map X^p-X is linear and its kernel is $\mathbb{F}_p$, thus if η is a root of $X^p-X-\alpha$ Eq. (6.11) follows immediately, implying that all the roots lie in the same extension of $\mathbb{F}_q$.

Suppose that $X^p-X-\alpha$ is split and let η be one of its roots. Then

$$\mathrm{Tr}_{\mathbb{F}_q/\mathbb{F}_p}(\alpha)=\mathrm{Tr}_{\mathbb{F}_q/\mathbb{F}_p}(\eta^p)-\mathrm{Tr}_{\mathbb{F}_q/\mathbb{F}_p}(\eta)=0, \tag{6.12}$$

where the last equality comes from 1.4.

Suppose now that $\mathrm{Tr}_{\mathbb{F}_q/\mathbb{F}_p}(\alpha)=0$, and let η be a root of $X^p-X-\alpha$ in its splitting field. Let $m=[\mathbb{F}_q:\mathbb{F}_p]$, then by $(\mathbf{P}_1)$

$$0=\mathrm{Tr}_{\mathbb{F}_q/\mathbb{F}_p}(\alpha)=\sum_{i=0}^{m-1}\alpha^{p^i}=\sum_{i=0}^{m-1}(\eta^p-\eta)^{p^i}=\eta^q-\eta, \tag{6.13}$$

thus $\eta\in\mathbb{F}_q$. 

Following [vzGS92, Cou00], we also use a generalization of the trace, as already introduced in Section 2.2.4.

Definition 6.8 (Pseudotrace) The n-th *pseudotrace* of order m is the $\mathbb{F}_{p^m}$-linear operator

$$T_{(n,m)}:a\mapsto\sum_{\ell=0}^{n-1}a^{p^{m\ell}};$$

for $m=1$, we call it the n-th pseudotrace and write T_n.

Note. In our context, for $n=[\mathbb{U}_i:\mathbb{U}_j]=p^{i-j}$ and $m=[\mathbb{U}_j:\mathbb{F}_p]=p^j d$, $T_{(n,m)}(\nu)$ coincides with $\mathrm{Tr}_{\mathbb{U}_i/\mathbb{U}_j}(\nu)$ for ν in $\mathbb{U}_i$; however $T_{(n,m)}(\nu)$ remains defined for ν in a field extension of $\mathbb{U}_i$, whereas $\mathrm{Tr}_{\mathbb{U}_i/\mathbb{U}_j}(\nu)$ is not.

6.3 A primitive tower

Our first task in this section is to describe a specific Artin-Schreier tower where arithmetic will be fast; then, we explain how to construct this tower.

6.3.1 Definition

The following theorem extends results by Cantor [Can89, Theorem 1.2], who dealt with the case $\mathbb{U}_0 = \mathbb{F}_p$.

Theorem 6.9 *Let $\mathbb{U}_0 = \mathbb{F}_p[X_0]/Q_0$, with Q_0 irreducible of degree d, let $x_0 = X_0 \bmod Q_0$ and assume that $\mathrm{Tr}_{\mathbb{U}_0/\mathbb{F}_p}(x_0) \neq 0$. Let $(G_i)_{0\leqslant i<k}$ be defined by*

$$\begin{cases} G_0 = X_0 & \\ G_1 = X_1 & \text{if } p = 2 \text{ and } d \text{ is odd,} \\ G_i = X_i^{2p-1} & \text{in any other case.} \end{cases}$$

Then, $(G_i)_{0\leqslant i<k}$ defines a primitive tower $(\mathbb{U}_0, \ldots, \mathbb{U}_k)$.

As before, for $i \geqslant 1$, let $P_i = X_i^p - X_i - G_{i-1}$ and for $i \geqslant 0$, let K_i be the ideal $\langle Q_0, P_1, \ldots, P_i\rangle$ in $\mathbb{F}_p[X_0, \ldots, X_i]$. Then the theorem says that for $i \geqslant 0$, $\mathbb{U}_i = \mathbb{F}_p[X_0, \ldots, X_i]/K_i$ is a field, and that $x_i = X_i \bmod K_i$ generates it over $\mathbb{F}_p$. We prove it as a consequence of a more general statement.

Lemma 6.10 *Let $\mathbb{U}$ be the finite field with p^n elements, and let $\mathbb{U}'/\mathbb{U}$ be an extension field with $[\mathbb{U}' : \mathbb{U}] = p^i$. Let $\alpha \in \mathbb{U}'$ be such that*

$$\mathrm{Tr}_{\mathbb{U}'/\mathbb{U}}(\alpha) = \beta \neq 0, \tag{6.14}$$

then $\mathbb{F}_p[\beta] \subset \mathbb{F}_p[\alpha]$ and p^i divides $[\mathbb{F}_p[\alpha] : \mathbb{F}_p[\beta]]$.

Proof. Equation (6.14) can be written as $\beta = \sum_j \alpha^{p^{jn}}$, thus $\mathbb{F}_p[\beta] \subset \mathbb{F}_p[\alpha]$. The rest of the proof follows by induction on i. If $[\mathbb{U}' : \mathbb{U}] = 1$, then $\alpha = \beta$ and there is nothing to prove. If $i \geqslant 1$, let $\mathbb{U}''$ be the intermediate extension such that $[\mathbb{U}' : \mathbb{U}''] = p$ and let $\alpha' = \mathrm{Tr}_{\mathbb{U}'/\mathbb{U}''}(\alpha)$, then, by composition of traces (Eq. $\mathbf{P}_2$), $\mathrm{Tr}_{\mathbb{U}''/\mathbb{U}}(\alpha') = \beta$ and by induction hypothesis p^{i-1} divides $[\mathbb{F}_p[\alpha'] : \mathbb{F}_p[\beta]]$.

Now, suppose that p does not divide $[\mathbb{F}_p[\alpha] : \mathbb{F}_p[\alpha']]$. Since $\mathbb{F}_p[\alpha'] \subset \mathbb{U}''$, this implies that p does not divide $[\mathbb{U}''[\alpha] : \mathbb{U}'']$; but $\alpha \in \mathbb{U}'$ and $[\mathbb{U}' : \mathbb{U}''] = p$ by construction, so necessarily $[\mathbb{U}''[\alpha] : \mathbb{U}''] = 1$ and $\alpha \in \mathbb{U}''$. This implies $\mathrm{Tr}_{\mathbb{U}'/\mathbb{U}''}(\alpha) = p\alpha = 0$ and, by $\mathbf{P}_2$, $\beta = 0$. Thus, we have a contradiction and p must divide $[\mathbb{F}_p[\alpha] : \mathbb{F}_p[\alpha']]$. The claim follows.

Corollary 6.11 *With the same notation as above, if $\mathrm{Tr}_{\mathbb{U}'/\mathbb{U}}(\alpha)$ generates $\mathbb{U}$ over $\mathbb{F}_p$, then $\mathbb{F}_p[\alpha] = \mathbb{U}'$.*

Hereafter, recall that we write $\gamma_i = G_i \bmod K_i$. We prove that the γ_i's meet the conditions of the corollary.

Lemma 6.12 *If $p \neq 2$, for $i \geqslant 0$, $\mathbb{U}_i$ is a field and, for $i \geqslant 1$,*

$$\mathrm{Tr}_{\mathbb{U}_i/\mathbb{U}_{i-1}}(\gamma_i) = -\gamma_{i-1}. \tag{6.15}$$

Proof. Induction on i: for $i = 0$, this is true by hypothesis. For $i \geqslant 1$, by induction hypothesis $\mathbb{U}_0, \dots, \mathbb{U}_{i-1}$ are fields; we then set $i' = i - 1$ and prove by nested induction that $\mathrm{Tr}_{\mathbb{U}_{i'}/\mathbb{F}_p}(\gamma_{i'}) \neq 0$ under the hypothesis that $\mathbb{U}_0, \dots, \mathbb{U}_{i'}$ are fields. This, by Proposition 6.7, implies that $X_i^p - X_i - \gamma_{i-1}$ is irreducible in $\mathbb{U}_{i-1}[X_{i+1}]$ and $\mathbb{U}_i$ is a field.

For $i' = 0$, $\mathrm{Tr}_{\mathbb{U}_0/\mathbb{F}_p}(\gamma_0) = \mathrm{Tr}_{\mathbb{U}_0/\mathbb{F}_p}(x_0)$ is non-zero and we are done. For $i' \geqslant 1$, we know that $\gamma_{i'} = x_{i'}^{2p-1} = x_{i'}^p x_{i'}^{p-1}$, which rewrites

$$(x_{i'} + \gamma_{i'-1})x_{i'}^{p-1} = x_{i'}^p + \gamma_{i'-1}x_{i'}^{p-1} = \gamma_{i'-1} + x_{i'} + \gamma_{i'-1}x_{i'}^{p-1}. \tag{6.16}$$

By $\mathbf{P}_3$ we get Eq. (6.15), and by $\mathbf{P}_2$ we deduce the equality

$$\mathrm{Tr}_{\mathbb{U}_{i'}/\mathbb{F}_p}(\gamma_{i'}) = -\mathrm{Tr}_{\mathbb{U}_{i'-1}/\mathbb{F}_p}(\gamma_{i'-1}). \tag{6.17}$$

The induction assumption implies that this is non-zero, and the claim follows.

Lemma 6.13 *If $p = 2$, for $i \geqslant 0$, $\mathbb{U}_i$ is a field. For $i = 1$*

$$\mathrm{Tr}_{\mathbb{U}_1/\mathbb{U}_0}(\gamma_1) = \begin{cases} 1 + \gamma_0 & \text{if } d \text{ even}, \\ 1 & \text{if } d \text{ odd}, \end{cases} \tag{6.18}$$

and for $i \geqslant 2$

$$\mathrm{Tr}_{\mathbb{U}_i/\mathbb{U}_{i-1}}(\gamma_i) = 1 + \gamma_{i-1}. \tag{6.19}$$

Proof. The proof closely follows the previous one. For $i' = 0$, $\mathrm{Tr}_{\mathbb{U}_0/\mathbb{F}_p}(\gamma_0) = \mathrm{Tr}_{\mathbb{U}_0/\mathbb{F}_p}(x_0)$ is non-zero. For $i' = 1$ and d odd,

$$\mathrm{Tr}_{\mathbb{U}_1/\mathbb{U}_0}(\gamma_1) = \mathrm{Tr}_{\mathbb{U}_1/\mathbb{U}_0}(x_1) = 1 \tag{6.20}$$

by $\mathbf{P}_3$, and

$$\mathrm{Tr}_{\mathbb{U}_0/\mathbb{F}_p}(1) = d \bmod 2 \neq 0. \tag{6.21}$$

For all the other cases

$$\gamma_{i'} = x_{i'}^2 x_{i'} = \gamma_{i'-1} + (1 + \gamma_{i'-1})x_{i'}, \tag{6.22}$$

thus

$$\mathrm{Tr}_{\mathbb{U}_{i'}/\mathbb{U}_{i'-1}}(\gamma_{i'}) = 1 + \gamma_{i'-1} \tag{6.23}$$

by $\mathbf{P}_3$ and $\mathrm{Tr}_{\mathbb{U}_{i'-1}/\mathbb{F}_p}(1) = 0$. In any case, using the induction hypothesis and $\mathbf{P}_2$, we deduce $\mathrm{Tr}_{\mathbb{U}_{i'}/\mathbb{F}_p}(\gamma_{i'}) = 1$ and this concludes the proof.

Proof of Theorem 6.9. We prove that $\mathbb{U}_i = \mathbb{F}_p[\gamma_i]$, then the theorem follows because, clearly, $\mathbb{F}_p[\gamma_i] \subset \mathbb{F}_p[x_i]$.

If $p \neq 2$, by Lemma 6.12 and $\mathbf{P}_2$,

$$\mathrm{Tr}_{\mathbb{U}_i/\mathbb{U}_0}(\gamma_i) = (-1)^i \gamma_0, \tag{6.24}$$

thus $\mathbb{U}_i = \mathbb{F}_p[\gamma_i]$ by Corollary 6.11 and the fact that $\gamma_0 = x_0$ generates $\mathbb{U}_0$ over $\mathbb{F}_p$.

If $p = 2$, we first prove that $\mathbb{U}_1 = \mathbb{F}_p[\gamma_1]$. If d is odd, $\gamma_1^p + \gamma_1 = x_0$ implies $\mathbb{U}_0 \subset \mathbb{F}_p[\gamma_1]$, but $\gamma_1 \notin \mathbb{U}_0$, thus necessarily $\mathbb{U}_1 = \mathbb{F}_p[\gamma_1]$. If d is even, $\mathrm{Tr}_{\mathbb{U}_1/\mathbb{U}_0}(\gamma_1) = 1 + \gamma_0$ clearly generates $\mathbb{U}_0$ over $\mathbb{F}_p$, thus $\mathbb{U}_1 = \mathbb{F}_p[\gamma_1]$ by Corollary 6.11.

Now we proceed like in the $p \neq 2$ case by observing that $\mathrm{Tr}_{\mathbb{U}_i/\mathbb{U}_1}(\gamma_i) = 1 + \gamma_1$ generates $\mathbb{U}_1$ over $\mathbb{F}_p$.

Remark 6.14. The choice of the tower of Theorem 6.9 is in some sense *optimal* between the choices given by Corollary 6.11. In fact, each of the G_i's is the "simplest" polynomial in $\mathbb{F}_p[X_i]$ such that $\mathrm{Tr}_{\mathbb{U}_i/\mathbb{F}_p}(\gamma_i) \neq 0$, in terms of lowest degree and least number of monomials, as shown by Proposition 6.6 and Eq. (6.10).

We also remark that the construction we made in this section gives us a family of normal elements for free. In fact, recall the following proposition from [Hac97, Section 5].

Proposition 6.15 *Let $\mathbb{U}'/\mathbb{U}$ be an extension of finite fields with $[\mathbb{U}' : \mathbb{U}] = kp^i$ where k is prime to p and let $\mathbb{U}''$ be the intermediate field of degree k over $\mathbb{U}$. Then $x \in \mathbb{U}'$ is normal over $\mathbb{U}$ if and only if $\mathrm{Tr}_{\mathbb{U}'/\mathbb{U}''}(x)$ is normal over $\mathbb{U}$. In particular, if $[\mathbb{U}' : \mathbb{U}] = p^i$, then $x \in \mathbb{U}'$ is normal over $\mathbb{U}$ if and only if $\mathrm{Tr}_{\mathbb{U}'/\mathbb{U}}(x) \neq 0$.*

Then we easily deduce the following corollary.

Corollary 6.16 *Let $(\mathbb{U}_0, \ldots, \mathbb{U}_k)$ be an Artin-Schreier tower defined by some $(G_i)_{0 \leqslant i < k}$. Then, every γ_i is normal over $\mathbb{U}_0$; furthermore γ_i is normal over $\mathbb{F}_p$ if and only if $\mathrm{Tr}_{\mathbb{U}_i/\mathbb{U}_0}(\gamma_i)$ is normal over $\mathbb{F}_p$.*

In the construction of Theorem 6.9, if we furthermore suppose that γ_0 is normal over $\mathbb{F}_p$, using Lemma 6.12 we easily see that the conditions of the corollary are met for $p \neq 2$. For $p = 2$, this is the case only if $[\mathbb{U}_0 : \mathbb{F}_p]$ is even (we omit the proofs that if γ_0 is normal then so are $-\gamma_0$ and $1 + \gamma_0$).

Remark 6.17. Observe however that this does not imply the normality of the x_i's. In fact, they can *never* be normal because $\mathrm{Tr}_{\mathbb{U}_i/\mathbb{U}_{i-1}}(x_i) = 0$ by $\mathbf{P}_3$. Granted that γ_0 is normal over $\mathbb{F}_p$, it would be interesting to have an efficient algorithm to switch representations from the univariate $\mathbb{F}_p$-basis in x_i to the $\mathbb{F}_p$-normal basis generated by γ_i.

In particular, having such a change of representations would allow efficient computations of Frobenius automorphisms. However, in Section 6.5, we give a quasi-optimal algorithm to compute Frobenius automorphisms, making no use of this remark.

6.3.2 Building the tower

This subsection introduces the basic algorithms required to build the primitive tower of Theorem 6.9, that is, compute the required minimal polynomials Q_i.

Composition. We give first an algorithm for sparse polynomial composition, to be used in the construction of the tower defined before. Given P and R in $\mathbb{F}_p[X]$, we want to compute $P(R)$. For the cost analysis, it will be useful later on to consider both the degree k and the number of terms ℓ of R.

Compose is a recursive process that cuts P into $c + 1$ "slices" of degree less than p^n, recursively composes them with R, and concludes using Horner's scheme and the linearity of the p-power; a similar recursive step was used in [Ber98] to compose power series in small characteristic. At the leaves of the recursion tree, we use the algorithm NaiveCompose.

Lemma 6.18 NaiveCompose *has cost* $O(\deg(P)^2 k\ell)$.

Proof. At step i, ρ and S have degree at most ik. Computing the sum $S + p_i\rho$ takes $O(ik)$ operations and computing the product ρR takes $O(ik\ell)$ operations,

Algorithm 6.1: NaiveCompose

Input : $P, R \in \mathbb{F}_p[X]$.
Output : $P(R)$.

1: write $P = \sum_{i=0}^{\deg(P)} p_i X^i$, with $p_i \in \mathbb{F}_p$;
2: let $S = 0$, $\rho = 1$;
3: for $i \in [0, \dots, \deg(P)]$, let $S = S + p_i\rho$ and $\rho = \rho R$;
4: return S.

Algorithm 6.2: Compose

Input : $P, R \in \mathbb{F}_p[X]$.
Output : $P(R)$.

1: let $n = \lfloor \log_p(\deg(P)) \rfloor$ and $c = \deg(P)$ div p^n;
2: If $n = 0$, return NaiveCompose(P, R);
3: write $P = \sum_{i=0}^{c} P_i X^{ip^n}$, with $P_i \in \mathbb{F}_p[X], \deg P_i < p^n$;
4: for $i \in [0, \dots, c]$, let $Q_i =$ Compose(P_i, R);
5: let $Q = 0$;
6: for $i \in [c, \dots, 0]$, let $Q = QR(X^{p^n}) + Q_i$;
7: return Q.

since R has ℓ terms. The total cost of step i is thus $O(ik\ell)$, whence a total cost of $O(\deg(P)^2 k\ell)$.

Theorem 6.19 *If R has degree k and ℓ non-zero coefficients and if $\deg(P) = s$, then* Compose(P, R) *outputs* $P(R)$ *in* $O(ps \log_p(s) k\ell)$ *operations.*

Proof. To analyze the cost, we let $K(c, n)$ be the cost of Compose when $\deg(P) \leqslant (c+1)p^n$, with $c < p$. Then $K(c, 0) \in O(c^2 k\ell)$. For $n > 0$, at each pass in the loop at step 6, $\deg(Q) < cp^n k$, so that the multiplication (using the naive algorithm) and addition take $O(cp^n k\ell)$ operations. Thus the cost of the loop is $O(c^2 p^n k\ell)$, and the total cost satisfies

$$K(c, n) \leqslant (c+1)K(p-1, n-1) + O(c^2 p^n k\ell). \tag{6.25}$$

Let then $K'(n) = K(p-1, n)$, so that we have

$$K'(0) \in O(p^2 k\ell), \quad K'(n) \leqslant pK'(n-1) + O(p^{n+2} k\ell). \tag{6.26}$$

We deduce that $K'(n) \in O(p^{n+2} nk\ell)$, and finally

$$K(c, n) \in O(cp^{n+1} nk\ell + c^2 p^n k\ell). \tag{6.27}$$

The values c, n computed at step 1 of the top-level call to Compose satisfy $cp^n \leqslant s$ and $n \leqslant \log_p(s)$; this gives our conclusion.

Note. A binary divide-and-conquer algorithm [vzGG99, Exercise 9.20] has cost

$$O(M(sk)\log(s)).$$

Our algorithm has a slightly better dependency on s, but adds a polynomial cost in p and ℓ. However, we have in mind cases with p small and $\ell = 2$, where the latter solution is advantageous.

Computing the minimal polynomials. Theorem 6.9 shows that we have defined a primitive tower. To be able to work with it, we explain now how to compute the minimal polynomial Q_i of x_i over $\mathbb{F}_p$. This is done by extending Cantor's construction [Can89], which had $\mathbb{U}_0 = \mathbb{F}_p$.

For $i = 0$, we are given $Q_0 \in \mathbb{F}_p[X_0]$ such that $\mathbb{U}_0 = \mathbb{F}_p[X_0]/Q_0(X_0)$, so there is nothing to do; we assume that $\mathrm{Tr}_{\mathbb{U}_0/\mathbb{F}_p}(x_0) \neq 0$ to meet the hypotheses of Theorem 6.9.

Remark 6.20. If $\mathrm{Tr}_{\mathbb{U}_0/\mathbb{F}_p}(x_0) = 0$, assuming $\gcd(d, p) = 1$, we can replace Q_0 by $Q_0(X_0 - 1)$; this is done by taking $R = X_0 - 1$ in algorithm Compose, so by Theorem 6.19 the cost is $O(pd \log_p(d))$.

So, the only case we cannot handle is when p divides d and the trace of x_0 is zero. By Lemma 5.11, this happens if and only if the $(d-1)$-th coefficient of Q is equal to 0. If the polynomial Q is chosen at random, this happens with probability $1/q$.

For $i = 1$, we know that $x_1^p - x_1 = x_0$, so x_1 is a root of $Q_0(X_1^p - X_1)$. Since $Q_0(X_1^p - X_1)$ is monic of degree pd, we deduce that $Q_1 = Q_0(X_1^p - X_1)$. To compute it, we use algorithm Compose with arguments Q_0 and $R = X_1^p - X_1$; the cost is $O(p^2 d \log_p(d))$ by Theorem 6.19. The same arguments hold for $i = 2$ when $p = 2$ and d is odd.

To deal with other indices i, we follow Cantor's construction. Let $\Phi \in \mathbb{F}_p[X]$ be the reduction modulo p of the $(2p-1)$-th cyclotomic polynomial. Cantor implicitly works modulo an irreducible factor of Φ. The following shows that we can avoid factorization, by working modulo Φ.

Lemma 6.21 *Let $\mathcal{A} = \mathbb{F}_p[X]/\Phi$ and let $x = X \bmod \Phi$. For $Q \in \mathbb{F}_p[Y]$, define*

$$Q^\star = \prod_{i=0}^{2p-2} Q(x^i Y). \tag{6.28}$$

Then $Q^\star$ is in $\mathbb{F}_p[Y]$ and there exists $q^\star \in \mathbb{F}_p[Y]$ such that $Q^\star = q^\star(Y^{2p-1})$.

Proof. Let $F_1, \ldots, F_e$ be the irreducible factors of Φ and let f be their common degree. To prove that $Q^\star$ is in $\mathbb{F}_p[Y]$, we prove that for $j \leqslant e$,

$$Q_j^\star = Q^\star \bmod F_j \tag{6.29}$$

is in $\mathbb{F}_p[Y]$ and independent from j; the claim follows by Chinese remaindering.

For $j \leqslant e$, let a_j be a root of F_j in the algebraic closure of $\mathbb{F}_p$, so that

$$Q_j^\star = \prod_{i=0}^{2p-2} Q(a_j^i Y). \tag{6.30}$$

Since $\gcd(p^f, 2p-1) = 1$, $Q_j^\star$ is invariant under $\mathrm{Gal}(\mathbb{F}_{p^f}/\mathbb{F}_p)$, and thus in $\mathbb{F}_p[Y]$. Besides, for $j, j' \leqslant e$, $a_{j'} = a_j^k$ for some k coprime to $2p-1$, so that $Q_j^\star = Q_{j'}^\star$, as needed.

To conclude, note that for $j \leqslant e$,

$$Q_j^\star(a_j Y) = Q_j^\star(Y), \tag{6.31}$$

so that all coefficients of degree not a multiple of $2p-1$ are zero. Thus, $Q_j^\star$ has the form $q_j^\star(Y^{2p-1})$; by Chinese remaindering, this proves the existence of the polynomial $q^\star$. 

Algorithm 6.3: MinimalPolynomial

Input : Q_i, Φ.
Output : Q_{i+1}.
1: $Q^\star = \prod_{i=0}^{2p-2} Q(x^i Y) \bmod \Phi$;
2: $q^\star(Y^{2p-1}) = Q^\star(Y)$;
3: $Q_{i+1} = \mathsf{Compose}(q^\star, Y^p - Y)$.

We conclude as in [Can89]: supposing that we know the minimal polynomial Q_i of x_i over $\mathbb{F}_p$, we compute Q_{i+1} using algorithm MinimalPolynomial.

Theorem 6.22 *Algorithm* MinimalPolynomial *is correct and computes its output in*

$$O(p^{i+2} d \log_p(p^i d) + M(p^{i+2} d) \log(p)) \tag{6.32}$$

operations.

Proof. Since x_i is a root of Q_i, it is a root of $Q_i^\star$ too. So $\gamma_i = x_i^{2p-1}$ is a root of $q_i^\star$ and x_{i+1} is a root of $q_i^\star(Y^p - Y)$. Since the latter polynomial is monic of degree $p^{i+1}d$, it is the minimal polynomial Q_{i+1} of x_{i+1} over $\mathbb{F}_p$.

As for the complexity, the algorithm of [Bre93] computes Φ in $O(p^2)$ operations; then, polynomial multiplications of degree s in $\mathcal{A}[Y]$ can be done in $O(M(sp))$ operations by Kronecker substitution. The overall cost of step 1 is $O(M(p^{i+2}d)\log p)$ using a subproduct tree (see Section 2.2.5). Step 2 is free and step 3 costs $O(p^{i+2} d \log_p(p^i d))$. 

The former cost is linear in $p^{i+2}d$, up to logarithmic factors, for an input of size $p^i d$ and an output of size $p^{i+1}d$.

Some further operations will be performed when we construct the tower: we will precompute quantities that will be of use in the algorithms of the next sections. Details are given in the next sections, when needed.

6.4 Level embedding

We discuss here change-of-basis algorithms for the tower $(\mathbb{U}_0, \ldots, \mathbb{U}_k)$ of the previous section; these algorithms are needed for most further operations. We detail the main case where $P_i = X_i^p - X_i - X_{i-1}^{2p-1}$; the case $P_1 = X_1^p - X_1 - X_0$ (and $P_2 = X_2^2 + X_2 + X_1$ for $p = 2$ and d odd) is easier.

Recall the two families of $\mathbb{F}_p$-bases we have defined so far:

$$\mathbf{B}_i = \{x_0^{e_0} \cdots x_i^{e_i} \mid 0 \leqslant e_0 < d,\ 0 \leqslant e_j < p \text{ for } j > 0\}, \tag{6.33}$$

$$\mathbf{C}_i = \{x_i^a \mid 0 \leqslant a < p^i d\}. \tag{6.34}$$

The first one arises naturally when constructing the tower as a succession of Artin-Schreier extensions, and we expect our inputs to be given in such basis. Furthermore, lifting in $\mathbf{B}_j$ an element written in $\mathbf{B}_i$ for $i < j$ is immediate in this basis, and so is the inverse operation. The basis $\mathbf{C}_i$, on the other hand, is practical for multiplication, inversion, etc., but it is not evident how to lift elements.

We shall thus need algorithms to change between these two bases. Since x_i is clearly a separating element for the variety $V(K_i)$ (see Chapter 5), we will use

Proposition 5.24 to go from $\mathbf{B}_i$ to $\mathbf{C}_i$, but we shall need an algorithm for the inverse map first.

Instead of converting from $\mathbf{C}_i$ to $\mathbf{B}_i$ directly, we will pass through some intermediate bivariate bases to keep the complexity low. By Theorem 6.9, $\mathbb{U}_i$ equals $\mathbb{F}_p[X_{i-1}, X_i]/I$, where the ideal I admits the following Gröbner bases, for respectively the lexicographic orders $X_i > X_{i-1}$ and $X_{i-1} > X_i$:

$$\left| \begin{array}{r} X_i^p - X_i - X_{i-1}^{2p-1} \\ Q_{i-1}(X_{i-1}) \end{array} \right. \quad \text{and} \quad \left| \begin{array}{r} X_{i-1} - R_i(X_i) \\ Q_i(X_i) \end{array} \right. \tag{6.35}$$

with R_i in $\mathbb{F}_p[X_i]$. Both Gröbner bases are triangular and bivariate, one can go from one to the other using the algorithms of Pascal and Schost [PS06], in fact most of the ideas of this section are inspired by their paper.

Since $\deg(Q_{i-1}) = p^{i-1}d$ and $\deg(Q_i) = p^i d$, we associate the following $\mathbb{F}_p$-bases of $\mathbb{U}_i$ to each system:

$$\mathbf{D}_i = \{x_{i-1}^a x_i^b \mid 0 \leqslant a < p^{i-1}d,\ 0 \leqslant b < p\}, \tag{6.36}$$

$$\mathbf{C}_i = \{x_i^a \mid 0 \leqslant a < p^i d\}. \tag{6.37}$$

We describe an algorithm called Push-down which takes v written in the basis $\mathbf{C}_i$ and returns its coordinates in the basis $\mathbf{D}_i$. Then, using Proposition 5.24, we will be able to describe the inverse operation, called Lift-up. In other words, Push-down inputs $v \in \mathbb{U}_i$ and outputs the representation of v as

$$v = v_0 + v_1 x_i + \cdots + v_{p-1} x_i^{p-1}, \quad \text{with all } v_j \in \mathbb{U}_{i-1} \tag{6.38}$$

and Lift-up does the opposite.

Then, the change from $\mathbf{C}_i$ to $\mathbf{B}_i$ is done by repeatedly applying Push-down, and the opposite is obtained by repeatedly applying Lift-up.

Hereafter, we let $L : \mathbb{N} - \{0\} \to \mathbb{N}$ be such that both Push-down and Lift-up can be performed in $L(i)$ operations; to simplify some expressions appearing later on, we add the mild constraints that $p\,L(i) \leqslant L(i+1)$ and $p\,M(p^i d) \in O(L(i))$. To reflect the behavior of the implementation, we also allow precomputations. These precomputations are performed when we build the tower; further details are at the end of this section.

THEOREM 6.23 *One can take* $L(i)$ *in* $O(p^{i+1} d \log_p^2(p^i d) \ + \ p\,M(p^i d))$.

Remark that the input and output have size $p^i d$; using fast multiplication, the cost is linear in $p^{i+1}d$, up to logarithmic factors. The rest of this section is devoted to proving this theorem. Push-down is a divide-and-conquer process, adapted to the shape of our tower; Lift-up is a special case of Proposition 5.24, the power projection will be obtained using the transposed version of Push-down.

As said before, the algorithms of this section (and of the following ones) use precomputed quantities. To keep the pseudo-code simple, we do not explicitly list them in the inputs of the algorithms; we show, later, that the precomputation is fast too.

6.4.1 Modular multiplication

We first discuss a routine for multiplication by $X_i^{p^n}$ in $\mathbb{F}_p[Y, X_i]/(X_i^p - X_i - Y)$, and its transpose. We start by remarking that

$$X_i^{p^n} = X_i + R_n \mod (X_i^p - X_i - Y) \qquad \text{with } R_n = \sum_{j=0}^{n-1} Y^{p^j}. \tag{6.39}$$

Then, precisely, for k in $\mathbb{N}$, we are interested in the operation

$$\mathsf{MulMod}_{k,n} : A \mapsto (X_i + R_n)A \mod (X_i^p - X_i - Y), \tag{MulMod}$$

with $A \in \mathbb{F}_p[Y, X_i]$, $\deg_Y(A) < k$ and $\deg_{X_i}(A) < p$.

Since R_n is sparse, it is advantageous to use the naive algorithm; besides, to make transposition easy, we explicitly give the matrix of MulMod. Let m_0 be the $(k + p^{n-1}) \times k$ matrix having 1's on the diagonal only, and for $\ell \leqslant p^{n-1}$, let m_ℓ be the matrix obtained from m_0 by shifting the diagonal down by ℓ places. Let finally m' be the sum $\Sigma_{j=0}^{n-1} m_{p^j}$. Then one verifies that the matrix of $\mathsf{MulMod}_{k,n}$ is

$$\begin{bmatrix} m' & & & & & m_1 \\ m_0 & m' & & & & m_0 \\ & m_0 & m' & & & 0 \\ & & \ddots & \ddots & & \vdots \\ & & & \ddots & \ddots & 0 \\ & & & & m_0 & m' \end{bmatrix}, \tag{6.40}$$

with columns and rows indexed by

$$(X_i^j, \ldots, Y^{k-1}X_i^j)_{j<p} \quad \text{and} \quad (X_i^j, \ldots, Y^{k+p^{n-1}-1}X_i^j)_{j<p} \tag{6.41}$$

respectively. Since this matrix has $O(pnk)$ non-zero entries, we can compute both MulMod and its dual MulMod^* in $O(pnk)$ operations.

6.4.2 Push-down

The input of Push-down is $v \in \mathbb{U}_i$, that is, given in the basis $\mathbf{C}_i$; we see it as a polynomial $V \in \mathbb{F}_p[X_i]$ of degree less than $p^i d$. The output is the normal form of V modulo $X_i^p - X_i - X_{i-1}^{2p-1}$ and $Q_{i-1}(X_{i-1})$. We first use a divide-and-conquer subroutine to reduce V modulo $X_i^p - X_i - X_{i-1}^{2p-1}$; then, the result is reduced modulo $Q_{i-1}(X_{i-1})$ coefficient-wise.

To reduce V modulo $X_i^p - X_i - X_{i-1}^{2p-1}$, we first compute

$$W = V \bmod (X_i^p - X_i - Y), \tag{6.42}$$

then we replace Y by X_{i-1}^{2p-1} in W. Because our algorithm will be recursive, we let $\deg(V)$ be arbitrary; then, we have the following estimate for W.

Lemma 6.24 *We have* $\deg_Y(W) \leqslant \deg(V)/p$.

ALGORITHM 6.4: Push-down-rec

INPUT : $V \in \mathbb{F}_p[X_i]$ and $c, n \in \mathbb{N}$.
OUTPUT : $W \in \mathbb{F}_p[Y, X_i]$.
1: if $n = 0$ return V;
2: write $V = \sum_{j=0}^{c} V_j X_i^{jp^n}$, with $V_j \in \mathbb{F}_p[X_i], \deg V_j < p^n$;
3: for $j \in [0, \ldots, c]$, let $W_j =$ Push-down-rec$(V_j, p-1, n-1)$;
4: $W = 0$;
5: for $j \in [c, \ldots, 0]$, let $W =$ MulMod$_{(c+1)p^{n-1},n}(W) + W_j$;
6: return W.

ALGORITHM 6.5: Push-down

INPUT : $v \in \mathbb{U}_i$.
OUTPUT : v written as $v_0 + \cdots + v_{p-1} x_i^{p-1}$ with $v_j \in \mathbb{U}_{i-1}$.
1: let V be the canonical preimage of v in $\mathbb{F}_p[X_i]$;
2: let $n = \lfloor \log_p(p^i d - 1) \rfloor$ and $c = (p^i d - 1)$ div p^n;
3: let $W =$ Push-down-rec(V, c, n);
4: let $Z =$ Evaluate$(W, [X_{i-1}^{2p-1}, X_i])$;
5: let $Z = Z \bmod Q_{i-1}$;
6: return the residue class of Z mod $(X_i^p - X_i - X_{i-1}^{2p-1}, Q_{i-1})$.

Proof. Consider the matrix M of multiplication by X_i^p modulo $X_i^p - X_i - Y$; it has entries in $\mathbb{F}_p[Y]$. Due to the sparseness of the modulus, one sees that M has degree at most 1, and so M^k has coefficients of degree at most k. Thus, the remainders of $X_i^{pk}, \ldots, X_i^{pk+p-1}$ modulo $X_i^p - X_i - Y$ have degree at most k in Y. 

We compute W by a recursive subroutine Push-down-rec, similar to Compose. As before, we let c, n be such that $1 \leqslant c < p$ and $\deg(V) < (c+1)p^n$, so that we have

$$V = V_0 + V_1 X_i^{p^n} + \cdots + V_c X_i^{cp^n},$$

with all V_j in $\mathbb{F}_p[X_i]$ of degree less than p^n. First, we recursively reduce $V_0, \ldots, V_c$ modulo $X_i^p - X_i - Y$, to obtain bivariate polynomials $W_0, \ldots, W_c$. Let R_n be the polynomial defined in Equation (6.39). Then, we get W by computing $\Sigma_{j=0}^{c} W_j (X_i + R_n)^j$ modulo $X_i^p - X_i - Y$, using Horner's scheme as in Compose. Multiplications by $X_i + R_n$ modulo $X_i^p - X_i - Y$ are done using MulMod.

PROPOSITION 6.25 *Algorithm* Push-down *is correct and takes*

$$O(p^{i+1} d \log_p^2(p^i d) + p\, M(p^i d)) \tag{6.43}$$

operations.

Proof. Correctness is straightforward; note that at step 5 of Push-down-rec, $\deg_Y(W) < (c+1)p^{n-1}$, so our call to MulMod is justified. By the claim of Subsection 6.4.1 on the cost of MulMod, the total cost of that loop is $O(nc^2 p^n)$. As in Theorem 6.19, we deduce that the cost of Push-down-rec is $O(n^2 c^2 p^n)$.

In Push-down, we have $cp^n < p^i d$ and $n < \log_p(p^i d)$, so the previous cost is seen to be $O(p^{i+1} d \log_p^2(p^i d))$. Reducing one coefficient of Z modulo Q_{i-1} takes $O(M(p^i d))$ operations, so step 5 has cost $O(p\, M(p^i d))$. Step 6 is free, since at this stage Z is already reduced.

Algorithm 6.6: Push-down-rec*

Input : $L \in \mathbb{F}_p[[1/Y, 1/X_i]]$ and $c, n \in \mathbb{N}$.
Output : $M \in \mathbb{F}_p[[1/X_i]]$.

1: If $n = 0$ return L;
2: for all $j \in [c, \dots, 0]$ do
3: let $L_j = L \bmod Y^{1-n}$;
4: let $M_j = \text{Push-down-rec}^*(L_j, p-1, n-1)$;
5: let $L = \text{MulMod}^*_{(c+1)p^{n-1},n}(L)$;
6: return $\sum_{j=0}^{c} \frac{M_j}{X_i^{jp^n}}$.

6.4.3 *Transposed push-down*

Before giving the details for Lift-up, we discuss here the transpose of Push-down. As in Section 5.7, Push-down is equivalent to the map

$$\begin{aligned} \text{ev}_{x_i} : \mathbb{F}_p[T] &\to \mathbb{U}_i^{\mathbf{D}_i}, \\ g &\mapsto g(x_i). \end{aligned} \tag{6.44}$$

So its transpose is the map

$$\begin{aligned} \text{proj}_{x_i} : (\mathbb{U}_i^*)^{\mathbf{D}_i{}^*} &\to \mathbb{F}_p[[1/T]], \\ \ell &\mapsto \sum_{j \geqslant 0} \frac{\ell(x_i^j)}{T^j}. \end{aligned} \tag{6.45}$$

Push-down is an algebraic transform, thus, applying Theorem 3.32, the transposed algorithm is obtained by reversing the initial algorithm step by step, and replacing subroutines by their transposes. The overall cost remains the same; we review here the main transformations.

As usual, we identify the dual of the space $\mathbb{F}_p[Y, X_i]$ to $\mathbb{F}_p[[1/Y, 1/X_i]]$. Thus linear forms given as input to the algorithm are written as series

$$L = \sum_{a,b \geqslant 0} \frac{\ell_{a,b}}{Y^a X_i^b}. \tag{6.46}$$

We do the same for $\mathbb{F}_p[X_i]$ and $\mathbb{F}_p[X_{i-1}, X_i]$.

The initial loop at step 5 is a Horner scheme; the transposed loop is run backward, and its core becomes $L_j = L \bmod Y^{1-n}$ and $L = \text{MulMod}^*_{(c+1)p^{n-1},n}(L)$; a small simplification yields the pseudo-code we give. In Push-down, after calling Push-down-rec, we evaluate W at $[X_{i-1}^{2p-1}, X_i]$: the transposed operation Evaluate* is the map

$$\sum_{a,b} \frac{\ell_{a,b}}{X_{i-1}^a X_i^b} \mapsto \sum_{a,b} \frac{\ell_{(2p-1)a,b}}{Y^a X_i^b}. \tag{6.47}$$

Then, originally, we perform a Euclidean division by Q_{i-1} on Z. The transposed algorithm mod* amounts to compute the values of a sequence linearly generated by the polynomial Q_{i-1} from its first $p^{i-1}d$ values (see Section 2.2.8).

ALGORITHM 6.7: Push-down*

INPUT : $L \in \mathbb{F}_p[[1/X_{i-1}, 1/X_i]]$.
OUTPUT : $M \in \mathbb{F}_p[[1/T]]$.
1: let $n = \lfloor \log_p(p^i d - 1) \rfloor$ and $c = (p^i d - 1) \operatorname{div} p^n$;
2: let $P = \text{mod}^*(L, Q_{i-1})$;
3: let $M = \text{Evaluate}^*(P, [X_{i-1}^{2p-1}, X_i])$;
4: return Push-down-rec*(M, c, n);

ALGORITHM 6.8: Lift-up

INPUT : v written as $v_0 + \cdots + v_{p-1} x_i^{p-1}$ with $v_j \in \mathbb{U}_{i-1}$.
OUTPUT : $v \in \mathbb{U}_i$.
1: let $\ell = \text{TransposedMul}(v, \rho_i(1))$;
2: let $M = \frac{1}{T}\text{Push-down}^*(\ell)$;
3: let $V = Q_i M \bmod T^{p^i d}$;
4: return $v = V(x_i)Q_i(x_i)^{-1} = VQ_i'^{-1} \bmod Q_i$.

6.4.4 *Lift-up*

Let v be given in the basis $\mathbf{D}_i$ and W its canonical preimage in $\mathbb{F}_p[X_{i-1}, X_i]$. The lift-up algorithm finds V in $\mathbb{F}_p[X_i]$ such that

$$W = V \mod (X_i^p - X_i - X_{i-1}^{2p-1}, Q_{i-1}) \tag{6.48}$$

and outputs the residue class of V modulo Q_i. Hereafter, we assume that both $Q_i'^{-1} \bmod Q_i$ and the values $\rho_i(1)$ of the trace $\text{Tr}_{\mathbb{U}_i/\mathbb{F}_p}$ on the basis $\mathbf{D}_i$ are known. See the discussion below.

Lift-up. We use Proposition 5.24 to write v as a polynomial in x_i. To do this we proceed as in steps 4 and 5 of RUR. To compute the power projection we could use transposed bivariate modular composition as in [Sho99]; it is however more efficient to use Push-down*.

PROPOSITION 6.26 *Algorithm* Lift-up *is correct and takes*

$$O(p^{i+1} d \log_p^2(p^i d) + p\, M(p^i d)) \tag{6.49}$$

operations.

Proof. Correctness is a consequence of Theorem 5.16 and of the algorithm given in Section 5.7.

TransposedMul implements the transposed bivariate modular multiplication; an algorithm of cost $O(M(p^i d))$ for this is in [PS06, Corollary 2] (see also Section 2.2.8). The last subsection showed that step 2 has the same cost as Push-down. Then, the costs of steps 3 and 4 are $O(M(p^i d))$. 

Propositions 6.25 and 6.26 prove Theorem 6.23.

Precomputations. The precomputations, that are done at the construction of $\mathbb{U}_i$, are as follows. First, we need the values of the trace on the basis $\mathbf{D}_i$. By $(\mathbf{P}_2)$ we know that

$$\mathrm{Tr}_{\mathbb{U}_i/\mathbb{F}_p}(x_{i-1}^a x_i^b) = \mathrm{Tr}_{\mathbb{U}_{i-1}/\mathbb{F}_p} \circ \mathrm{Tr}_{\mathbb{U}_i/\mathbb{U}_{i-1}}(x_{i-1}^a x_i^b), \tag{6.50}$$

then, by $(\mathbf{P}_3)$

$$\mathrm{Tr}_{\mathbb{U}_i/\mathbb{U}_{i-1}}(x_{i-1}^a x_i^b) = \begin{cases} 0 & \text{for } 0 \leqslant b < p-1, \\ -x_{i-1}^a & \text{for } b = p-1. \end{cases} \tag{6.51}$$

Thus the values of $\mathrm{Tr}_{\mathbb{U}_i/\mathbb{F}_p}$ on the basis $\mathbf{D}_i$ are

$$0, \dots, 0, -\mathrm{Tr}_{\mathbb{U}_{i-1}/\mathbb{F}_p}(1), -\mathrm{Tr}_{\mathbb{U}_{i-1}/\mathbb{F}_p}(x_{i-1}), \dots, -\mathrm{Tr}_{\mathbb{U}_{i-1}/\mathbb{F}_p}(x_{i-1}^{p^{i-1}d-1}). \tag{6.52}$$

They can be computed in $O(M(p^{i-1}d))$ operations using Lemma 5.11.

Then, we need $Q_i'^{-1} \bmod Q_i$; this takes $O(M(p^i d)\log(p^i d))$ operations by fast extended GCD computation. These precomputations save logarithmic factors at best, but are useful in practice.

6.5 Frobenius and pseudotrace

In this section, we describe algorithms computing Frobenius and pseudotrace operators, specific to the tower of Section 6.3; they are the keys to the algorithms of the next section.

The algorithms in this section and the next one closely follow Couveignes' [Cou00]. However, the latter assumed the existence of a quasi-linear time algorithm for multiplication in some specific towers in the multivariate basis $\mathbf{B}_i$ of Subsection 6.2.1. To our knowledge, no such algorithm exists. We use here the univariate basis $\mathbf{C}_i$ introduced previously, which makes multiplication straightforward. However, several push-down and lift-up operations are now required to accommodate the recursive nature of the algorithm.

Our main purpose here is to compute the pseudotrace

$$\mathrm{T}_{p^j d} : x \mapsto \sum_{\ell=0}^{p^j d-1} x^{p^\ell}; \tag{6.53}$$

we already gave an algorithm for this task in Section 2.2.4, but in our context we can do better. We start by describing how to compute values of the iterated Frobenius operator $x \mapsto x^{p^n}$ by a recursive descent in the tower.

We focus on computing the iterated Frobenius for $n < d$ or $n = p^j d$. In both cases, similarly to (6.39), we have:

$$x_i^{p^n} = x_i + \beta_{i-1,n}, \quad \text{with} \quad \beta_{i-1,n} = \mathrm{T}_n(\gamma_{i-1}). \tag{6.54}$$

Assuming $\beta_{i-1,n}$ is known, the recursive step of the Frobenius algorithm follows: starting from $v \in \mathbb{U}_i$, we first write $v = v_0 + \cdots + v_{p-1}x_i^{p-1}$, with $v_h \in \mathbb{U}_{i-1}$; by (6.54) and the linearity of the Frobenius, we deduce that

$$v^{p^n} = \textstyle\sum_{h=0}^{p-1} v_h^{p^n} \left(x_i + \beta_{i-1,n}\right)^h. \tag{6.55}$$

ALGORITHM 6.9: IterFrobenius

INPUT : v, i, n with $v \in \mathbb{U}_i$ and $n < d$ or $n = p^j d$.
OUTPUT : $v^{p^n} \in \mathbb{U}_i$.
1: if $n = p^j d$ and $i \leqslant j$, return v;
2: if $i = 0$, return v^{p^n};
3: let $v_0 + v_1 x_i + \cdots + v_{p-1} x_i^{p-1} = \mathsf{Push\text{-}down}(v)$;
4: for $h \in [0, \ldots, p-1]$, let $t_h = \mathsf{IterFrobenius}(v_h, i-1, n)$;
5: let $F = 0$;
6: for $h \in [p-1, \ldots, 0]$, let $F = t_h + (x_i + \beta_{i-1,n})F$;
7: return $\mathsf{Lift\text{-}up}(F)$.

Then, we compute all $v_h^{p^n}$ recursively; the final sum is computed using Horner's scheme. Note that these equations are not limited to the case where $n < d$ or of the form $p^j d$: an arbitrary n would do as well. However, we impose this limitation since these are the only values we need to compute $T_{p^j d}$.

In the case $n = p^j d$, any $v \in \mathbb{U}_j$ is left invariant by this Frobenius map, thus we stop the recursion when $i = j$, as there is nothing left to do. In the case $n < d$, we stop the recursion when $i = 0$ and apply the algorithm of Section 2.2.4. We summarize the two variants in one unique algorithm IterFrobenius.

As mentioned above, the algorithm requires the values $\beta_{i',n}$ for $i' < i$: we suppose that they are precomputed (the discussion of how we precompute them follows). To analyze costs, we use the function L of Section 6.4.

THEOREM 6.27 *On input $v \in \mathbb{U}_i$ and $n = p^j d$, algorithm* IterFrobenius *correctly computes v^{p^n} and takes*

$$O((i-j)L(i)) \tag{6.56}$$

operations.

Proof. Correctness is clear. We write $F(i,j)$ for the complexity on inputs as in the statement; then $F(0,j) = \cdots = F(j,j) = 0$ because step 1 comes at no cost. For $i > j$, each pass through step 6 involves a multiplication by $x_i + \beta_{i-1,n}$, of cost of $O(pM(p^{i-1}d))$, assuming $\beta_{i-1,n} \in \mathbb{U}_{i-1}$ is known. Altogether, we deduce the recurrence relation

$$F(i,j) \leqslant p\,F(i-1,j) + 2\,L(i) + O(p^2 M(p^{i-1}d)), \tag{6.57}$$

so $F(i,j) \leqslant p\,F(i-1,j) + O(L(i))$, by assumptions on M and L. The conclusion follows, again by assumptions on L. ∎

THEOREM 6.28 *On input $v \in \mathbb{U}_i$ and $n < d$, algorithm* IterFrobenius *correctly computes v^{p^n} and takes*

$$O(p^i C(d)\log(n) + iL(i)) \tag{6.58}$$

operations.

Proof. The analysis is identical to the previous one, except that step 2 is now executed instead of step 1 and this costs $O(C(d)\log(n))$ by the algorithm of Section 2.2.4. The conclusion follows by observing that step 2 is repeated p^i times. ∎

Algorithm 6.10: LittlePseudotrace

Input : ν, i, n with $\nu \in \mathbb{U}_i$ and $0 < n \leqslant d$.
Output : $T_n(\nu) \in \mathbb{U}_i$.
1: if $n = 1$ return ν;
2: let $m = \lfloor n/2 \rfloor$;
3: let $t = \mathsf{LittlePseudotrace}(\nu, i, m)$;
4: let $t = t + \mathsf{IterFrobenius}(t, i, m)$;
5: if n is odd, let $t = t + \mathsf{IterFrobenius}(\nu, i, n)$;
6: return t.

Algorithm 6.11: Pseudotrace

Input : ν, i, j with $\nu \in \mathbb{U}_i$.
Output : $T_{p^j d}(\nu) \in \mathbb{U}_i$.
1: if $j = 0$ return $\mathsf{LittlePseudotrace}(\nu, d)$;
2: $t_0 = \mathsf{Pseudotrace}(\nu, i, j-1)$;
3: for $h \in [1, \ldots, p-1]$, let $t_h = \mathsf{IterFrobenius}(t_{h-1}, i, j-1)$;
4: return $t_0 + t_1 + \cdots + t_{p-1}$.

Next, we compute pseudotraces. We use the following relations, whose verification is straightforward:

$$T_{n+m}(\nu) = T_n(\nu) + T_m(\nu)^{p^n}, \qquad T_{nm}(\nu) = \sum_{h=0}^{m-1} T_n(\nu)^{p^{hn}}. \tag{6.59}$$

We give two *divide-and-conquer* algorithms that do a slightly different *divide* step; each of them is based on one of the previous formulas. The first one, LittlePseudotrace, is meant to compute T_d. It follows a binary divide-and-conquer scheme similar to the algorithm in Section 2.2.4. The second one, Pseudotrace, computes $T_{p^j d}$ for $j > 0$. It uses the previous formula with $n = p^{j-1}d$ and $m = p$, computing Frobenius-es for such n; when $j = 0$, it invokes the first algorithm.

Theorem 6.29 *Algorithm* LittlePseudotrace *is correct and takes*

$$O(p^i C(d) \log^2(n) + iL(i) \log(n)) \tag{6.60}$$

operations.

Proof. Correctness is clear. For the cost analysis, we write $\mathsf{PT}(i, n)$ for the cost on input i and n, so $\mathsf{PT}(i, 1) = O(1)$. For $n > 1$, step 3 costs $\mathsf{PT}(i, \lfloor n/2 \rfloor)$, steps 4 and 5 cost both

$$O(p^i C(d) \log^2(n) + iL(i)) \tag{6.61}$$

by Theorem 6.28. This gives

$$\mathsf{PT}(i, n) = \mathsf{PT}(i, \lfloor n/2 \rfloor) + O(p^i C(d) \log^2(n) + iL(i)), \tag{6.62}$$

and thus

$$\mathsf{PT}(i, n) \in O(p^i C(d) \log^2(n) + iL(i) \log n). \tag{6.63}$$



THEOREM 6.30 *Algorithm* Pseudotrace *is correct and takes*

$$\mathsf{PT}(i) = O((pi + \log(d))iL(i) + p^i C(d)\log^2(d)) \tag{6.64}$$

operations for $j \leqslant i$.

Proof. Correctness is clear. For the cost analysis, we write $\mathsf{PT}(i,j)$ for the cost on input i and j, so theorem 6.29 gives

$$\mathsf{PT}(i,0) = O(p^i C(d)\log^2(d) + iL(i)\log(d)). \tag{6.65}$$

For $j > 0$, step 2 costs $\mathsf{PT}(i,j-1)$, step 3 costs $O(piL(i))$ by Theorem 6.27 and step 4 costs $O(p^{i+1}d)$. This gives

$$\mathsf{PT}(i,j) = \mathsf{PT}(i,j-1) + O(piL(i)), \tag{6.66}$$

and thus

$$\mathsf{PT}(i,j) \in O(pijL(i) + \mathsf{PT}(i,0)). \tag{6.67}$$



The cost is thus $O(p^{i+2}d + p^i C(d))$, up to logarithmic factors, for an input and output size of $p^i d$: this time, due to modular compositions in $\mathbb{U}_0$, the cost is not linear in d.

Finally, let us discuss precomputations. On input ν, i, d, the algorithm LittlePseudotrace makes less than $2\log d$ calls to IterFrobenius(x,i,n) for some value $x \in \mathbb{U}_i$ and for $n \in N$ where the set N only depends on d. When we construct $\mathbb{U}_{i+1}$, we compute (only) all $\beta_{i,n} = T_n(\gamma_i) \in \mathbb{U}_i$, for increasing $n \in N$, using the LittlePseudotrace algorithm. The inner calls to IterFrobenius only use pseudotraces that are already known. Besides, a single call to LittlePseudotrace(γ_i,i,d) actually computes *all* $T_n(\gamma_i)$ in

$$O(p^i C(d)\log^2 d + iL(i)\log d) \tag{6.68}$$

operations. Same goes for the precomputation of all $\beta_{i,p^j d} = T_{p^j d}(\gamma_i) \in \mathbb{U}_i$, for $j \leqslant i$, using the Pseudotrace algorithm: this costs $\mathsf{PT}(i)$. Observe that in total we only store $O(k^2 + k\log d)$ elements of the tower, thus the space requirements are quasi-linear.

Remark 6.31. A dynamic programming version of LittlePseudotrace as in Section 2.2.4 would only precompute $\beta_{i,2^e}$ for $2^e < d$, thus reducing the storage from $2\log d$ to $\lfloor \log d \rfloor$ elements. This would also allow to compute T_n for any $n < d$ without needing any further precomputation. Using this algorithm and a decomposition of $n > d$ as $n = r + \sum_j c_j p^j d$ with $r < d$ and $c_j < p$, one could also compute T_n and x^{p^n} for any n at essentially the same cost. We omit these improvements since they are not essential to the next section.

6.6 Arbitrary towers

Finally, we bring our previous algorithms to an arbitrary tower, using Couveignes' isomorphism algorithm [Cou00]. As in the previous section, we adapt this algorithm to our context, by adding suitable push-down and lift-up operations.

Let Q_0 be irreducible of degree d in $\mathbb{F}_p[X_0]$, such that $\mathrm{Tr}_{\mathbb{U}_0/\mathbb{F}_p}(x_0) \neq 0$, with as before $\mathbb{U}_0 = \mathbb{F}_p[X_0]/Q_0$. We let $(G_i)_{0\leqslant i<k}$ and $(\mathbb{U}_0, \ldots, \mathbb{U}_k)$ be as in Section 6.3.

We also consider another sequence $(G'_i)_{0\leqslant i<k}$, that defines another tower $(\mathbb{U}'_0, \ldots, \mathbb{U}'_k)$. Since $(\mathbb{U}'_0, \ldots, \mathbb{U}'_k)$ is not necessarily primitive, we fall back to the multivariate basis of Subsection 6.2.1: we write elements of $\mathbb{U}'_i$ in the basis

$$\mathbf{B}'_i = \{x_0'^{e_0} \cdots x_i'^{e_i} \mid 0 \leqslant e_0 < d,\ 0 \leqslant e_j < p \text{ for } j > 0\}, \tag{6.69}$$

where $x_0 = x'_0$.

To compute in $\mathbb{U}'_i$, we will use an isomorphism $\mathbb{U}'_i \to \mathbb{U}_i$. Such an isomorphism is determined by the images $\mathbf{s}_i = (s_0, \ldots, s_i)$ of $(x'_0, \ldots, x'_i)$, with $s_i \in \mathbb{U}_i$ (we always take $s_0 = x_0$). This isomorphism, denoted by $\sigma_{\mathbf{s}_i}$, takes as input v written in the basis $\mathbf{B}'_i$ and outputs $\sigma_{\mathbf{s}_i}(v) \in \mathbb{U}_i$.

To analyze costs, we use the functions L and PT introduced in the previous sections. We also let $2 \leqslant \omega \leqslant 3$ be a feasible exponent for linear algebra over $\mathbb{F}_p$ (see Section 2.1).

Theorem 6.32 *Given Q_0 and $(G'_i)_{0\leqslant i<k}$, one can find $\mathbf{s}_k = (s_0, \ldots, s_k)$ in*

$$O(d^\omega k + \mathsf{PT}(k) + M(p^{k+1}d)\log(p)) \tag{6.70}$$

operations. Once they are known, one can apply $\sigma_{\mathbf{s}_k}$ and $\sigma_{\mathbf{s}_k}^{-1}$ using $O(kL(k))$ operations.

Thus, we can compute products, inverses, etc, in $\mathbb{U}'_k$ for the cost of the corresponding operation in $\mathbb{U}_k$, plus $O(kL(k))$.

6.6.1 Solving Artin-Schreier equations

As a preliminary, given $\alpha \in \mathbb{U}_i$, we discuss how to solve the Artin-Schreier equation $X^p - X = \alpha$ in $\mathbb{U}_i$. We assume that $\mathrm{Tr}_{\mathbb{U}_i/\mathbb{F}_p}(\alpha) = 0$, so this equation has solutions in $\mathbb{U}_i$.

Because $X^p - X$ is $\mathbb{F}_p$-linear, the equation can be directly solved by linear algebra, but this is too costly. In [Cou00], Couveignes gives a solution adapted to our setting, that reduces the problem to solving Artin-Schreier equations in $\mathbb{U}_0$. He observed what follows.

Proposition 6.33 *Let $\delta \in \mathbb{U}_i$ be a solution of the equation*

$$X^p - X = \alpha. \tag{6.71}$$

Any solution μ of

$$X^{p^{p^{i-1}d}} - X = \eta, \quad \textit{with} \quad \eta = \mathrm{T}_{p^{i-1}d}(\alpha). \tag{6.72}$$

is of the form $\mu = \delta - \Delta$ with $\Delta \in \mathbb{U}_{i-1}$.

Proof. Let μ be a solution of Eq. (6.72), and let $\Delta = \delta - \mu$, then

$$\Delta^{p^{p^{i-1}d}} - \Delta = \delta^{p^{p^{i-1}d}} - \delta - \eta = \left(\sum_{\ell=0}^{p^{i-1}d-1} (\delta^p - \delta)^{p^\ell}\right) - \eta =$$

$$\mathrm{T}_{p^{i-1}d}(\delta^p - \delta) - \eta = 0, \tag{6.73}$$

where the last equality comes from the definition of the pseudotrace. This implies $\Delta \in \mathbb{U}_{i-1}$. ∎

ALGORITHM 6.12: ApproximateAS

INPUT : $\eta \in \mathbb{U}_i$ such that (6.72) has a solution.
OUTPUT : $\mu \in \mathbb{U}_i$ solution of (6.72).
1: let $\eta_0 + \eta_1 x_i + \cdots + \eta_{p-2} x_i^{p-2} = \text{Push-down}(\eta)$;
2: FOR ALL $j \in [p-1, \ldots, 1]$ DO
3: let $\mu_j = \frac{1}{jT}\left(\eta_{j-1} - \sum_{h=j+1}^{p-1} \binom{h}{j-1} \beta_{i-1,p^{i-1}d}^{h-j+1} \mu_h\right)$;
4: return $\text{Lift-up}(\mu_1 x_i + \ldots + \mu_{p-1} x_i^{p-1})$;

By its definition, $\Delta = \delta - \mu$ is a root of

$$X^p - X - \alpha + \mu^p - \mu. \tag{6.74}$$

This equation has solutions in $\mathbb{U}_{i-1}$ by the previous proposition, hence it can be solved recursively. In a certain sense, μ is a good approximation to δ, and their difference Δ can be computed as the solution of a new, simpler, Artin-Schreier equation.

First we tackle the problem of finding a solution of (6.72). For this purpose, observe that the left hand side of (6.72) is $\mathbb{U}_{i-1}$-linear and its matrix in the basis $(1, \ldots, x_i^{p-1})$ is

$$\begin{bmatrix} 0 & \binom{1}{0}\beta_{i-1,p^{i-1}d} & \binom{2}{0}\beta_{i-1,p^{i-1}d}^2 & \cdots & \binom{p-1}{0}\beta_{i-1,p^{i-1}d}^{p-1} \\ 0 & 0 & \binom{2}{1}\beta_{i-1,p^{i-1}d} & \cdots & \binom{p-1}{1}\beta_{i-1,p^{i-1}d}^{p-2} \\ \vdots & & & & \vdots \\ 0 & \cdots & \cdots & 0 & \binom{p-1}{p-2}\beta_{i-1,p^{i-1}d} \\ 0 & \cdots & \cdots & \cdots & 0 \end{bmatrix} \tag{6.75}$$

(we recall that $\beta_{i-1,n} = T_n(\gamma_{i-1})$). Then, algorithm ApproximateAS finds the required solution.

THEOREM 6.34 *Algorithm* ApproximateAS *is correct and takes* $O(L(i))$ *operations.*

Proof. Correctness is clear by Gaussian elimination. For the cost analysis, note that $\beta_{i-1,p^{i-1}d}$ has already been precomputed as a prerequisite for the iterated Frobenius and pseudotrace algorithms. Step 2 takes $O(p^2)$ additions and scalar operations in $\mathbb{U}_{i-1}$; the overall cost is dominated by the one of the push-down and lift-up steps, by assumptions on L. 

Writing the recursive algorithm is now straightforward. To solve Artin-Schreier equations in $\mathbb{U}_0$, we use a naive algorithm based on linear algebra, written Naive-Solve.

THEOREM 6.35 *Algorithm* Artin-Schreier *is correct and takes*

$$O(d^\omega + PT(i)) \tag{6.76}$$

operations.

Proof. Correctness follows from the previous discussion. For the complexity, let $AS(i)$ be the cost for $\alpha \in \mathbb{U}_i$. The cost $AS(0)$ of the naive algorithm is

Algorithm 6.13: Artin-Schreier

Input : α, i such that $\alpha \in \mathbb{U}_i$ and $\mathrm{Tr}_{\mathbb{U}_i/\mathbb{F}_p}(\alpha) = 0$.
Output : $\delta \in \mathbb{U}_i$ such that $\delta^p - \delta = \alpha$.
1: if $i = 0$, return NaiveSolve$(X^p - X - \alpha)$;
2: let $\eta =$ Pseudotrace$(\alpha, i, i-1)$;
3: let $\mu =$ ApproximateAS(η);
4: let $\alpha_0 =$ Push-down$(\alpha - \mu^p + \mu)$;
5: let $\Delta =$ Artin-Schreier$(\alpha_0, i-1)$;
6: return $\mu +$ Lift-up(Δ).

Algorithm 6.14: ApplyIsomorphism

Input : ν, i with $\nu \in \mathbb{U}'_i$ written in the basis $\mathbf{B}'_i$.
Output : $\sigma_{\mathbf{s}_i}(\nu) \in \mathbb{U}_i$.
1: if $i = 0$ then return ν;
2: write $\nu = \Sigma_{j<p} \nu_j(x'_0, \ldots, x'_{i-1}) x_i'^j$;
3: let $s_{i,0} + \cdots + s_{i,p-1} x_i^{p-1} =$ Push-down(s_i);
4: for $j \in [0, \ldots, p-1]$ let $t_j =$ ApplyIsomorphism$(\nu_j, i-1)$;
5: let $t = 0$;
6: for $j \in [p-1, \ldots, 0]$ let $t = (s_{i,0} + \cdots + s_{i,p-1} x_i^{p-1}) t + t_j$;
7: return Lift-up(t).

$O(M(d)\log(p) + d^\omega)$, where the first term is the cost of computing x_0^p and the second one the cost of linear algebra.

When $i \geqslant 1$, step 2 has cost $PT(i)$, steps 3, 4 and 6 all contribute $O(L(i))$ and step 5 contributes $AS(i-1)$. The most important contribution is at step 2, hence $AS(i) = AS(i-1) + O(PT(i))$. The assumptions on L imply that the sum $PT(1) + \cdots + PT(i)$ is $O(PT(i))$.

6.6.2 *Applying the isomorphism*

We get back to the isomorphism question. We assume that $\mathbf{s}_i = (s_0, \ldots, s_i)$ is known and we give the cost of applying $\sigma_{\mathbf{s}_i}$ and its inverse. We first discuss the forward direction.

As input, $\nu \in \mathbb{U}'_i$ is written in the multivariate basis $\mathbf{B}'_i$ of $\mathbb{U}'_i$; the output is $t = \sigma_{\mathbf{s}_i}(\nu) \in \mathbb{U}_i$. As before, the algorithm is recursive: we write $\nu = \Sigma_{j<p} \nu_j(x'_0, \ldots, x'_{i-1}) x_i'^j$, whence

$$\sigma_{\mathbf{s}_i}(\nu) \;=\; \sum_{j<p} \sigma_{\mathbf{s}_i}(\nu_j) s_i^j \;=\; \sum_{j<p} \sigma_{\mathbf{s}_{i-1}}(\nu_j) s_i^j; \tag{6.77}$$

the sum is computed by Horner's scheme. To speed-up the computation, it is better to perform the latter step in a bivariate basis, that is, through a push-down and a lift-up.

Given $t \in \mathbb{U}_i$, to compute $\nu = \sigma_{\mathbf{s}_i}^{-1}(t)$, we run the previous algorithm backward. We first push-down t, obtaining $t = t_0 + \cdots + t_{p-1} x_i^{p-1}$, with all $t_j \in \mathbb{U}_{i-1}$. Next, we rewrite this as $t = t'_0 + \cdots + t'_{p-1} s_i^{p-1}$, with all $t'_j \in \mathbb{U}_{i-1}$, and it suffices to apply $\sigma_{\mathbf{s}_i}^{-1}$ (or equivalently $\sigma_{\mathbf{s}_{i-1}}^{-1}$) to each t'_i. The non-trivial part is the computation of the t'_j's: this is done by another application of RUR in the extension $\mathbb{U}_i = \mathbb{U}_{i-1}[X_i]/(P_i)$,

ALGORITHM 6.15: ApplyInverse

INPUT : t, i with $t \in \mathbb{U}_i$.
OUTPUT : $\sigma_{s_i}^{-1}(t) \in \mathbb{U}'_i$ written in the basis $\mathbf{B}'_i$.
1: if $i = 0$ then return t;
2: let $t_0 + \cdots + t_{p-1}x_i^{p-1} = \text{Push-down}(t)$;
3: let $s_{i,0} + \cdots + s_{i,p-1}x_i^{p-1} = \text{Push-down}(s_i)$;
4: let $t'_0 + \cdots + t'_{p-1}X^{p-1} = \text{RUR}_{\mathbf{s}_i}(t_0 + \cdots + t_{p-1}x_i^{p-1}, s_{i,0} + \cdots + s_{i,p-1}x_i^{p-1})$;
5: return $\Sigma_{j<p}$ ApplyInverse$(t'_j, i-1)x_i'^j$;

that we shall call $\text{RUR}_{\mathbf{s}_i}$. We shall discuss $\text{RUR}_{\mathbf{s}_i}$ later, for the moment we just state a result about its complexity.

LEMMA 6.36 *Algorithm* $\text{RUR}_{\mathbf{s}_i}$ *computes its output in* $O(pM(p^i d))$ *operations.*

PROPOSITION 6.37 *Algorithms* ApplyIsomorphism *and* ApplyInverse *are correct and both take* $O(iL(i))$ *operations.*

Proof. In both cases, correctness is clear, since the algorithms translate the former discussion. As to complexity, in both cases, we do p recursive calls, $O(1)$ push-downs and lift-ups, and a few extra operations: for ApplyIsomorphism, these are p multiplications / additions in the bivariate basis $\mathbf{D}_i$ of Section 6.4; for ApplyInverse, this is calling the algorithm $\text{RUR}_{\mathbf{s}_i}$. By using Kronecker substitution to multiply elements in the basis $\mathbf{D}_i$, the cost is $O(pM(p^i d))$ in both cases, which is in $O(L(i))$ by assumptions on L. We conclude as in Theorem 6.27. 

Rational univariate representation. We describe now the algorithm to change from the basis $\mathbf{D}_i$ to the basis $\mathbf{D}'_i$, based on the rational univariate representation. Unlike Lift-up, this algorithm will not make use of transposed subroutines.

We consider the algebra $\mathbb{U}_i[X_i]/(P_i)$. We have an element s_i, its minimal polynomial P'_i, and we know that s_i separates $V(P_i)$. We want to express an element $t \in \mathbb{U}_i$ written in the basis $\mathbf{D}_i$ as a rational fraction in s_i.

We first observe that the values of $\text{Tr}_{\mathbb{U}_i/\mathbb{U}_{i-1}}$ over the basis $1, x_i, \ldots, x_i^{p-1}$ are

$$\rho(1) = (0, \ldots, 0, -1) \tag{6.78}$$

by $(\mathbf{P}_3)$. Now, as in step 4 of RUR, we need to compute $t \cdot \rho(1)$. By writing down the multiplication matrix of t, we verify that

$$t \cdot \rho(1) = (-t_{p-1}, -t_{p-2}, \ldots, -t_1, -t_0 - t_{p-1}). \tag{6.79}$$

Also observe that we need the inverse of the derivative of P'_i, as in step 5 of RUR. Since P'_i is Artin-Schreier, this value is just -1. The algorithm $\text{RUR}_{\mathbf{s}_i}$ is given below.

Proof of Lemma 6.36. Step 2 is the most expensive one. We use $p-1$ multiplications in the bivariate basis $\mathbf{D}_i$ to compute $s_i, \ldots, s_i^{p-1}$, then we apply ℓ to each of them. These operations cost respectively $O(pM(p^i d))$ by Kronecker substitution, and $O(p^2 M(p^{i-1} d))$. Then, steps 1 and 3 cost one addition in $\mathbb{U}_{i-1}$ each, because $P'_i(T) = T^p - T - \gamma'_{i-1}$. 

Algorithm 6.16: $\mathrm{RUR}_{\mathbf{s}_i}$

Input : $t_0, \dots, t_{p-1}, s_{i,0}, \dots, s_{i,p-1} \in \mathbb{U}_{i-1}$ such that $s_{i,0} + \cdots + s_{i,p-1} x_i^{p-1} = s_i$.
Output : $t'_0, \dots, t'_{p-1} \in \mathbb{U}_{i-1}$ such that $t_0 + \cdots + t_{p-1} x_i^{p-1} = t'_0 + \cdots + t'_{p-1} s_i^{p-1}$.

1: let $\ell = t \cdot \rho(1) = -\frac{t_{p-1}}{X_i^{p-1}} - \sum_{j=0}^{p-1} \frac{t_{p-1-j}}{X_i^j}$;
2: let $M = \frac{1}{T} \sum_{j=0}^{p-1} \frac{\ell(s_i^j)}{T^j}$;
3: let $V = P'_i M \bmod T^p$;
4: return $-V$.

Note. We could have done better in step 2 by using transposed modular composition, but this would not influence the overall complexity of ApplyInverse.

6.6.3 *Proof of Theorem 6.32*

Finally, assuming that only $(s_0, \dots, s_{i-1})$ are known, we describe how to determine s_i. Several choices are possible: the only constraint is that s_i should be a root of $X_i^p - X_i - \sigma_{\mathbf{s}_i}(\gamma'_{i-1}) = X_i^p - X_i - \sigma_{\mathbf{s}_{i-1}}(\gamma'_{i-1})$ in $\mathbb{U}_i$.

Using Proposition 6.37, we can compute $\alpha = \sigma_{\mathbf{s}_{i-1}}(\gamma'_{i-1}) \in \mathbb{U}_{i-1}$ in $O((i-1)L(i-1)) \subset O(iL(i))$ operations. Applying a lift-up to α, we are then in the conditions of Theorem 6.35, so we can find s_i for an extra $O(d^\omega + PT(i))$ operations.

We can then summarize the cost of all precomputations: to the cost of determining $\mathbf{s}_i$, we add the costs related to the tower $(\mathbb{U}_0, \dots, \mathbb{U}_i)$, given in Sections 6.3, 6.4 and 6.5. After a few simplifications, we obtain the upper bound $O(d^\omega + PT(i) + M(p^{i+1}d)\log(p))$. Summing over i gives the first claim of the theorem. The second is a restatement of Proposition 6.37.

6.7 Experimental results

We discuss here the implementation of the algorithms of this Chapter and some experimental results.

Implementation. We packaged the algorithms of this chapter in a `C++` library called `FAAST` and made it available under the terms of the `GNU GPL` software license from `http://www.lix.polytechnique.fr/Labo/Luca.De-Feo/FAAST/`.

`FAAST` is implemented on top of the `NTL` library [Sho03] which provides the basic univariate polynomial arithmetic needed here. Our library handles three NTL classes of finite fields: `GF2` for $p = 2$, `zz_p` for word-size p and `ZZ_p` for arbitrary p; this choice is made by the user at compile-time through the use of `C++` templates and the resulting code is thus quite efficient. Optionally, `NTL` can be combined with the `gf2x` package [BGTZ08] for better performance in the $p = 2$ case, as we did in our experiments.

All the algorithms of Sections 6.3–6.5 are faithfully implemented in `FAAST`. The algorithms ApplyIsomorphism and ApplyInverse have slightly different implementations `toUnivariate()` and `toBivariate()` that allow more flexibility. Instead of being recursive algorithms doing the change to and from the multivariate basis $\mathbf{B}'_i = \{x'_0{}^{e_0} \cdots x'_i{}^{e_i}\}$, they only implement the change to and from the bivariate basis

$$\mathbf{D}'_i = \{x_{i-1}{}^{e_{i-1}} x'_i{}^{e_i} \mid 0 \leqslant e_{i-1} < p^{i-1}d,\ 0 \leqslant e_i < p\}. \tag{6.80}$$

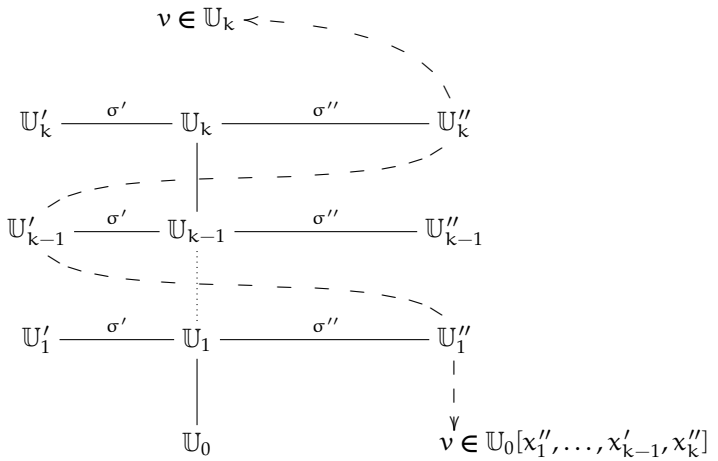


FIGURE 6.1: An example of conversion from the univariate basis to the multivariate basis of the tower $(\mathbb{U}_0, \mathbb{U}''_1, \ldots, \mathbb{U}'_{k-1}, \mathbb{U}''_k)$.

Equivalently, this amounts to switch between the representations

$$\in \mathbb{U}_i \quad \text{and} \quad \in \mathbb{U}_{i-1}[X'_i]/(X'^p_i - X'_i - \gamma'_{i-1}). \tag{6.81}$$

The same result as one call to ApplyIsomorphism or ApplyInverse can be obtained by `i` calls to `toUnivariate()` and `toBivariate()` respectively.

However, more freedom is allowed in the case where several generic Artin-Schreier towers, say $(\mathbb{U}'_0, \ldots, \mathbb{U}'_k)$ and $(\mathbb{U}''_0, \ldots, \mathbb{U}''_k)$, are built using the algorithms of Section 6.6. In fact it is possible, for example, to see $\mathbb{U}''_k$ as an extension field of $\mathbb{U}'_k$ by first converting to the basis $\mathbf{D}''_k$ and then recursively to $\mathbf{D}'_{k-1}, \ldots, \mathbf{D}'_1$. More generally, this allows the user to build a new Artin-Schreier tower $(\mathbb{U}'''_0, \ldots, \mathbb{U}'''_k)$ by taking at each level either $\mathbb{U}'''_i = \mathbb{U}'_i$ or $\mathbb{U}'''_i = \mathbb{U}''_i$. In other words this allows the user to *zig-zag* in the lattice of finite fields as in Figure 6.1.

Besides the algorithms of this Chapter, `FAAST` also implements the algorithms described in Section 8.7 for minimal polynomials, evaluation and interpolation.

Experimental results. We compare our timings with those obtained in Magma 2.16 [BCP97] for similar questions. All results are obtained on an Intel Xeon E5520 (2.26GHz). Our experiments revealed a regression in the performances of Magma 2.16, concerning one algorithm. When such difference is noticeable, we also plot the timings obtained with Magma 2.11 on an equivalent machine (Intel Xeon E5430).

The experiments for the `FAAST` library were only made for the classes `GF2` and `zz_p`. The class `ZZ_p` was left out because all the primes that can be reasonably handled by our library fit in one machine-word. In Magma, there exist several ways to build field extensions:

- `quo<U|P>` builds the quotient of the univariate polynomial ring `U` by `P` $\in$ `U` (written magma(1) hereafter);
- `ext<k|P>` builds the extension of the field `k` by `P` $\in$ `k`[X] (written magma(2));

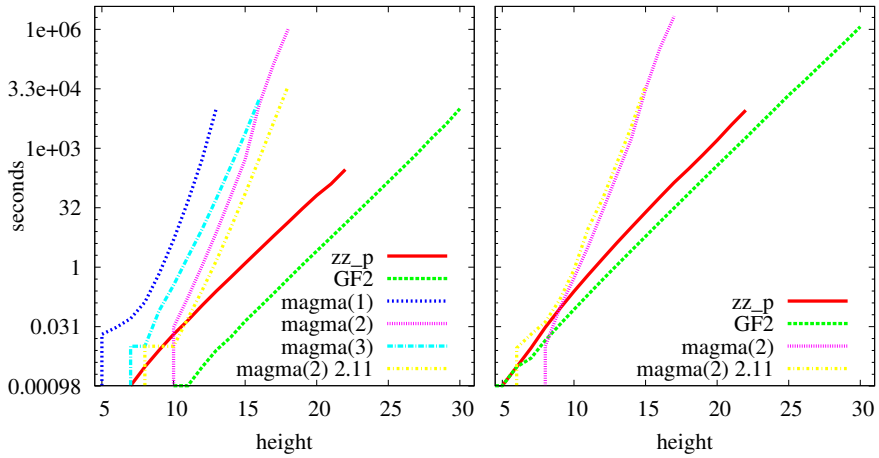


Figure 6.2: Build time (left) and isomorphism time (right) with respect to tower height. Plot is in logarithmic scale.

- `ext<k|p>` builds an extension of degree p of k (written magma(3)).

We made experiments for each of these choices where this makes sense.

The parameters to our algorithms are (p, d, k). Thus, our experiments describe the following situations:

- *Increasing the height* k. Here we take $p = 2$ and $d = 1$ (that is, $\mathbb{U}_0 = \mathbb{F}_2$); the x-coordinate gives the number of levels we construct and the y-coordinate gives timings in seconds, in *logarithmic* scale.

  This is done in Figure 6.2. We let the height of the tower increase and we give timings for (1) building the tower of Section 6.3 and (2) computing an isomorphism with a random arbitrary tower as in Section 6.6. In the latter experiment, only the magma(2) approach was meaningful for Magma.

- *Increasing the degree* d *of* $\mathbb{U}_0$. Here we take $p = 5$ and we construct 2 levels; the x-coordinate gives the degree $d = [\mathbb{U}_0 : \mathbb{F}_p]$ and the y-coordinate gives timings in seconds. This is done in Figure 6.3 (left).

- *Increasing* p. Here we take $d = 1$ (thus $\mathbb{U}_0 = \mathbb{F}_p$) and we construct 2 levels; the x-coordinate gives the characteristic p and the y-coordinate gives timings in seconds. This is done in Figure 6.3 (right).

The timings of our code are significantly better for increasing height or increasing d. Not surprisingly, for increasing p, the magma(1) approach performs better than any other: the quo operation simply creates a residue class ring, regardless of the (ir)reducibility of the modulus, so the timing for building two levels barely depend on p. The most adapted approach for this situation presumably is magma(2); yet we notice that `FAAST` has reasonable performances for characteristics up to about $p = 50$.

In Tables 6.1 and 6.2 we provide some comparative timings for the different arithmetic operations provided by `FAAST`. The column "Primitive" gives the time

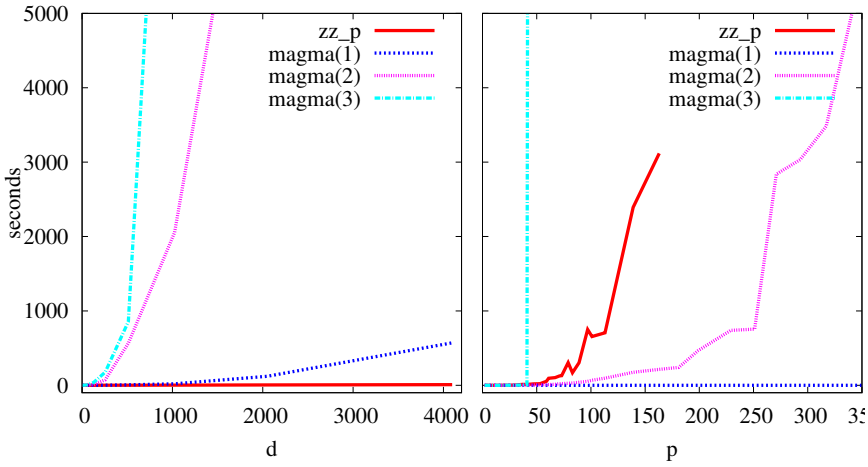


FIGURE 6.3: Build times with respect to d (left) and p (right).

level	Primitive	Push-d.	Lift-up	Product	Reciprocal	apply σ^{-1}	apply σ
19	1.143	0.304	1.265	0.039	0.649	0.652	1.290
20	2.566	0.609	2.796	0.081	1.544	1.314	2.602
21	5.686	1.225	6.147	0.187	3.598	2.409	2.668
22	12.660	2.515	13.746	0.463	8.355	5.565	11.179
23	28.511	5.295	31.200	1.046	19.522	12.323	24.740

TABLE 6.1: Some timings in seconds for arithmetics in a generic tower built over $\mathbb{F}_2$ using `GF2`.

level	Primitive	Push-d.	Lift-up	Product	Reciprocal	apply σ^{-1}	apply σ
18	13.618	0.884	13.712	0.476	10.753	1.337	3.578
19	30.288	1.814	30.432	1.001	23.046	2.850	7.798
20	65.632	3.953	66.889	2.106	51.544	6.564	18.141
21	128.190	8.347	131.271	4.791	121.349	14.396	39.296
22	296.671	11.396	298.541	6.413	249.520	28.851	86.628

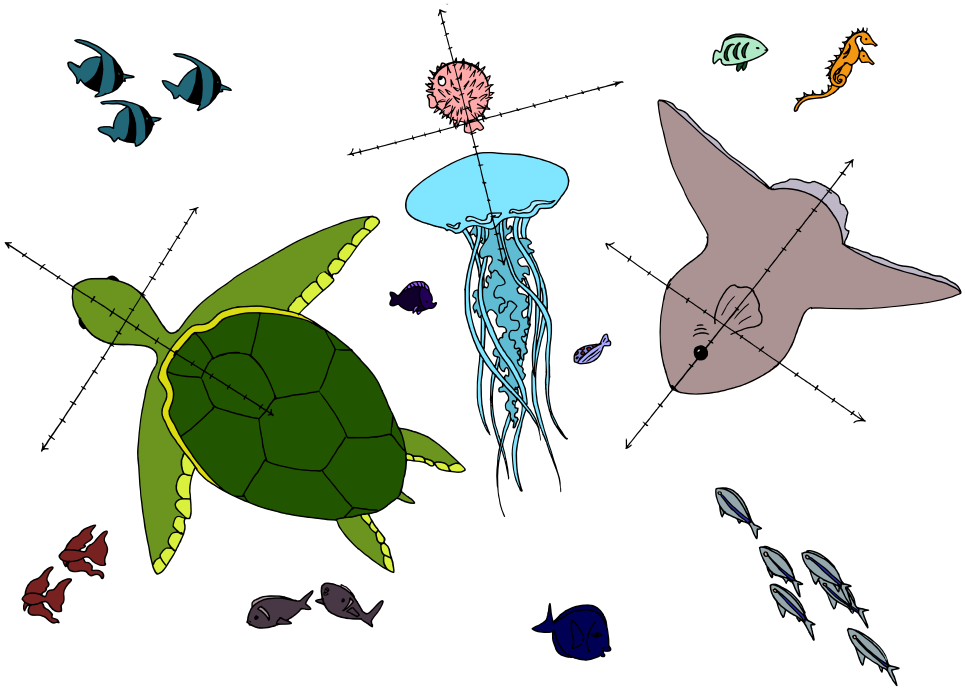
TABLE 6.2: Some timings in seconds for arithmetics in a generic tower built over $\mathbb{F}_2$ using `zz_p`.

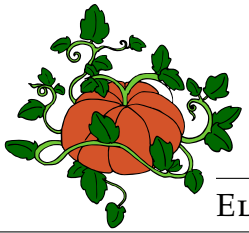
taken to build one level of the primitive tower (this includes the precomputation of the data as described in Subsection 6.4.4); the other entries are self-explanatory. Product and inversion are just wrappers around `NTL` routines: in these operations we did not observe any overhead compared to the native `NTL` code.

Finally, we mention the cost of precomputation. The precomputation of the images of σ as explained in Section 6.6 is quite expensive; most of it is spent computing pseudotraces. Indeed it took one week to precompute the data in Figure 6.2 (right), while all the other data can be computed in a few hours. There is still space for some minor improvement in `FAAST`, mainly tweaking recursion thresholds and implementing better algorithms for small and moderate input sizes. Yet we think that only a major algorithmic improvement could consistently speed up this phase.

Part IV

Applications to isogeny computation and cryptography





Chapter 7

Elliptic curves and isogenies

The techniques we have developed to perform fast computations in some finite-dimensional algebras, find an application in some number-theoretic computations on elliptic curves.

Elliptic curves are a central object of study in number theory and algebraic geometry. They are probably most famous for having being used by Wiles [Wil95, TW95] to prove of Fermat's last theorem. In computer science, they have applications in cryptography [Kob87, Mil86, BSS99], factorization [Len87, AM93b, BBLP08] and primality proving [AM93a, Mor07].

In this section we recall the basic definitions and properties of elliptic curves, the material is drawn from [Sil86, Mil96, Con91].

7.1 Definitions

We let $\mathbb{K}$ be a field and $\bar{\mathbb{K}}$ be its algebraic closure. Elliptic curves are genus 1 curves over $\bar{\mathbb{K}}$ with a distinguished point.

7.1.1 Weierstrass equations

Definition 7.1 (Weierstrass equation) Any elliptic curve E can be viewed as the projective variety associated to the equation

$$E \ : \ Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3, \tag{7.1}$$

with $a_1, \ldots, a_6 \in \bar{\mathbb{K}}$. The distinguished point is $[0 : 1 : 0]$, called the *point at infinity* and denoted by $\mathcal{O}$. When $a_1, \ldots, a_6 \in \mathbb{K}$, the curve is said to be *defined over* $\mathbb{K}$.

Eq. (7.1) is called the *homogeneous Weierstrass form* of the curve E. After the change of variables $x = \frac{X}{Z}, y = \frac{Y}{Z}$, Eq. (7.1) can be rewritten as

$$E \ : \ y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \tag{7.2}$$

called the *affine Weierstrass form*. Then the curve E is the locus of zeros of Eq. (7.2) in $\bar{\mathbb{K}}^2$ (or, more exactly, in the affine plane $\mathbb{A}^2$), plus the extra point at infinity $\mathcal{O}$.

The discriminant of Eq. (7.2) is

$$\Delta_E = -b_2^2b_8 - 8b_4^3 - 27b_6^2 + 9b_2b_4b_6, \tag{7.3}$$

where

$$\begin{aligned} b_2 &= a_1^2 + 4a_2, \\ b_4 &= 2a_4 + a_1 a_3, \\ b_6 &= a_3^2 + 4a_6, \\ b_8 &= a_1^2 a_6 + 4a_2 a_6 - a_1 a_3 a_4 + a_2 a_3^2 - a_4^2. \end{aligned} \tag{7.4}$$

Remark 7.2. The curve defined by (7.2) has an unique singular point if and only if $\Delta_E = 0$. Figure 7.1 shows the two possible shapes of a singular curve over the reals. Then, every line passing through the singular point meets the curve in another unique point. This parameterization makes E birationally equivalent to $\mathbb{P}^1$, thus E has genus 0. The converse is also true, thus Eq. (7.2) defines an elliptic curve if and only if $\Delta_E \neq 0$.

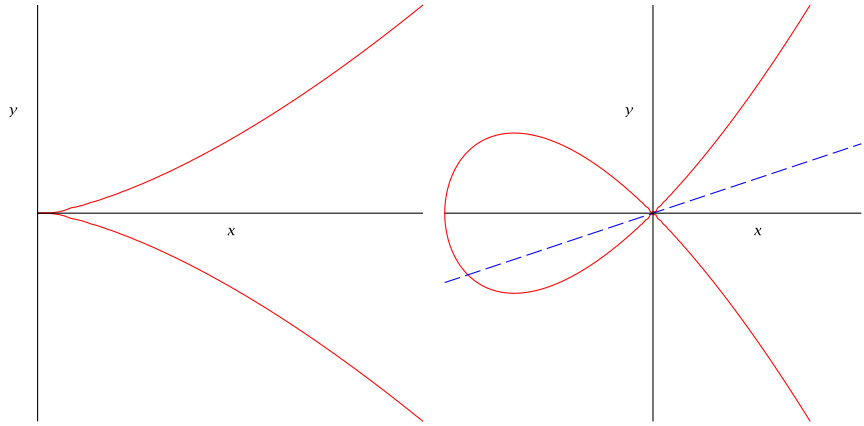


Figure 7.1: Two singular Weierstrass curves. On the left $y^2 = x^3$, on the right $y^2 = x^3 + x^2$. On the right we have shown the parameterization by the line passing through the origin.

Definition 7.3 (j-invariant) We associate to the curve E defined by Eq. (7.2), the j-invariant

$$j_E = \frac{(b_2^2 - 24b_4)^3}{\Delta_E}.$$

Note. Recall that if E is an elliptic curve defined by an affine Weierstrass equation $f(x, y) = 0$ with coefficients in $\mathbb{K}$, its function field $\mathbb{K}(E)$ is the field of fractions of

$$\mathbb{K}[E] \stackrel{\text{def}}{\equiv} \mathbb{K}[x, y]/(f(x, y)). \tag{7.5}$$

7.1.2 Group law

Elliptic curves are endowed with a group structure via the *chord-tangent law*.

Definition 7.4 Let E be an elliptic curve and let $P, Q \in \mathbb{P}^2$ be two points on the curve. Let L be the line passing through P and Q, with multiplicity two if $P = Q$, and let R be the third intersection point with E. Let L' be the line passing through

R and $\mathcal{O}$. Then the point $P + Q$ is defined as the third point of intersection of L' and E.

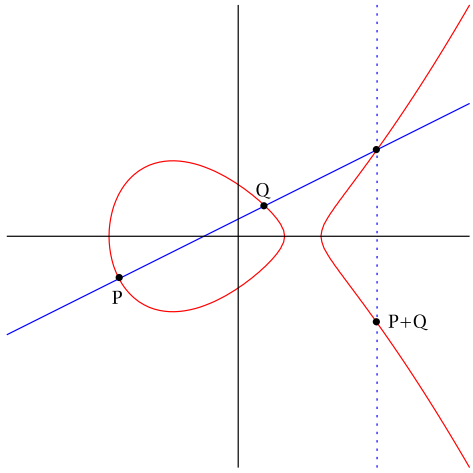


FIGURE 7.2: Chord-tangent law on an elliptic curve defined over the reals.

The definition is pictured in Figure 7.2. For a proof that this defines indeed a group law on the points of E, with $\mathcal{O}$ being the identity element, see [Sil86, II, §2].

DEFINITION 7.5 (Rational points) Let E be an elliptic curve defined over $\mathbb{K}$, the set of $\mathbb{K}$-*rational points* $E(\mathbb{K})$ is

$$E(\mathbb{K}) = E \cap \mathbb{P}^2(\mathbb{K}). \tag{7.6}$$

If P and Q are rational points, the lines L and L' are defined by equations over $\mathbb{K}$, thus $E(\mathbb{K})$ forms a subgroup of E. To allow calculations in $E(\mathbb{K})$ without having to lift the coefficients in an algebraic closure, it is convenient to give explicit formulas for the chord-tangent law.

We shall give the formulas for affine coordinates. x, y are functions in $\mathbb{K}(E)$, thus from now on, if P is a point of E, we denote by $x(P)$ its abscissa and by $y(P)$ its ordinate (in affine coordinates).

PROPOSITION 7.6 *Let E be the elliptic curve defined by Eq. (7.2). Let P, be a point of E different from $\mathcal{O}$, the coordinates of $-P$ are given by*

$$x(-P) = x(P), \qquad y(-P) = -y(P) - a_1 x(P) - a_3. \tag{7.7}$$

Let P, Q be points of E different from $\mathcal{O}$ and let $Q \neq -P$, the coordinates of $P + Q$ are given by

$$\begin{aligned} \lambda &= \begin{cases} \frac{y(Q)-y(P)}{x(Q)-x(P)} & \text{if } P \neq Q, \\ \frac{3x(P)^2+2a_2x(P)+a_4-a_1y(P)}{2y(P)+a_1x(P)+a_3} & \text{if } P = Q, \end{cases} \\ x(P+Q) &= \lambda^2 + a_1\lambda - a_2 - x(P) - x(Q), \\ y(P+Q) &= -(\lambda + a_1)x(P+Q) - y(P) + \lambda x(P) - a_3. \end{aligned} \tag{7.8}$$

For $m \in \mathbb{Z}$, we denote by $[m]P$ the point $\overbrace{P + P + \cdots + P}^{m \text{ times}}$ if $m > 0$, or the point $[-m](-P)$ if $m < 0$, or $\mathcal{O}$ if $m = 0$.

Definition 7.7 (Kummer variety) The *Kummer variety* of an elliptic curve E, denoted by K_E, is the quotient of E by the equivalence relation $P \simeq -P$.

Remark 7.8. The Kummer variety can be represented by taking the abscissas of the points of E. It is not a group, but we can still compute scalar multiples of its points. In fact, from the addition formulas we deduce for $P \neq -P$

$$x([2]P) = \frac{x^4 - b_4x^2 - 2b_6x - b_8}{4x^3 + b_2x^2 + 2b_4x + b_6}, \tag{7.9}$$

where $x = x(P)$; and for $P \neq Q$

$$x(P + Q) + x(P - Q) = \frac{2\pi\sigma + b_2\pi + b_4\sigma + b_6}{(x(P) - x(Q))^2}, \tag{7.10}$$

where $\pi = x(P)x(Q)$ and $\sigma = x(P) + x(Q)$.

Then, to compute $x([n]P)$, we start from $x(P)$ and $x([2]P)$, and we iteratively apply

$$\begin{aligned} [2i]P &= [2][i]P, \\ [2i+1]P &= [i+1]P + [i]P, \end{aligned} \tag{7.11}$$

or

$$\begin{aligned} [2i+1]P &= [i+1]P + [i]P, \\ [2i+2]P &= [2][i+1]P, \end{aligned} \tag{7.12}$$

until we reach $[n]P$. This algorithm appeared in [Mon87] and is sometimes referred to as *Montgomery's formulas*.

The map

$$\begin{aligned} [m]_E : E(\mathbb{K}) &\to E(\mathbb{K}), \\ P &\mapsto [m]P \end{aligned} \tag{7.13}$$

is a group endomorphisms of $E(\mathbb{K})$.

Definition 7.9 The m-th *torsion subgroup* of E is

$$E[m] = \{P \in E(\bar{\mathbb{K}}) \mid [m]P = \mathcal{O}\}, \tag{7.14}$$

its points are called m-*torsion points*.

Since addition is an algebraic map, multiplication by n is algebraic too. It can be shown that there exist polynomials $\psi_m, \theta_m, \omega_m \in \mathbb{K}(E)$ such that

$$[m](x, y) = \left(\frac{\theta_m(x, y)}{\psi_m(x, y)^2}, \frac{\omega_m(x, y)}{\psi_m(x, y)^3} \right). \tag{7.15}$$

Definition 7.10 (Division polynomials) The polynomial ψ_m is called the m-*th division polynomial*.

Remark 7.11. The division polynomials can be computed from the addition formulas via a double-and-add approach. Their importance comes from the fact that ψ_m vanishes on $E[m]$.

From the formulas for the division polynomials one can deduce the structure of the m-torsion.

THEOREM 7.12 *Let p be the characteristic of $\mathbb{K}$. If p does not divide m*

$$E[m] \cong \mathbb{Z}/m\mathbb{Z} \times \mathbb{Z}/m\mathbb{Z}; \tag{7.16}$$

if $p \neq 0$, then for every $i > 0$ either

$$E[p^i] = \{\mathcal{O}\}, \qquad \text{or} \qquad E[p^i] \cong \mathbb{Z}/p^i\mathbb{Z}. \tag{7.17}$$

DEFINITION 7.13 (Supersingular, ordinary) An elliptic curve E is said to be *supersingular* if $E[p^i] = \{\mathcal{O}\}$ for any i; it is said to be *ordinary* otherwise.

DEFINITION 7.14 (Tate module) Let ℓ be a prime, the *ℓ-adic Tate module* $\mathcal{T}_\ell(E)$ is the group

$$\mathcal{T}_\ell(E) = \varprojlim_n E[\ell^n] \tag{7.18}$$

with respect to the projections

$$[\ell]_E : E[\ell^{n+1}] \to E[\ell^n]. \tag{7.19}$$

PROPOSITION 7.15 *The Tate module has a natural structure of $\mathbb{Z}_\ell$-module. As such*

$$\mathcal{T}_\ell(E) \cong \begin{cases} \mathbb{Z}_\ell \times \mathbb{Z}_\ell & \text{if } \ell \neq p, \\ \mathbb{Z}_p & \text{if } \ell = \mathrm{char}(\mathbb{K}) \text{ and } E \text{ is ordinary,} \\ \{\mathcal{O}\} & \text{if } E \text{ is supersingular.} \end{cases} \tag{7.20}$$

7.1.3 Isomorphisms

Let E and E' be two elliptic curves in Weierstrass form over a field $\mathbb{K}$, they are said to be isomorphic if there is a linear change of variables that transforms one equation in the other and preserves the point at infinity. Then, clearly $E(\bar{\mathbb{K}})$ and $E'(\bar{\mathbb{K}})$ are isomorphic as groups. If the change of variables has coefficients in $\mathbb{K}$, the curves are said to be isomorphic over $\mathbb{K}$, then $E(\mathbb{K})$ and $E'(\mathbb{K})$ are isomorphic as groups.

It can be shown that the only such changes of variables are

$$\begin{aligned} x &= u^2x' + r, \\ y &= u^3y' + u^2sx' + t, \end{aligned} \tag{7.21}$$

with $r, s, t, u \in \bar{\mathbb{K}}$ and $u \neq 0$.

PROPOSITION 7.16 *Two elliptic curves E and E' in Weierstrass form are isomorphic over $\bar{\mathbb{K}}$ if and only if $j_E = j_{E'}$.*

Weierstrass equations can be brought via isomorphism to a form that is easier to handle, called *simplified Weierstrass form*. The following classification is from [Con91].

Proposition 7.17 (Simplified Weierstrass form) *Any elliptic curve is isomorphic to one of the following Weierstrass forms:*

- *If* $\mathrm{char}(\mathbb{K}) = 0$ *or* $\mathrm{char}(\mathbb{K}) > 3$

$$E : y^2 = x^3 + ax + b \qquad \textit{and} \qquad j_E = \frac{1728(4a)^3}{16(4a^3 + 27b^2)}; \tag{7.22}$$

- *if* $\mathrm{char}(\mathbb{K}) = 3$ *and* E *is ordinary*

$$E : y^2 = x^3 + ax^2 + b \qquad \textit{and} \qquad j_E = -\frac{a^3}{b}; \tag{7.23}$$

- *if* $\mathrm{char}(\mathbb{K}) = 3$ *and* E *is supersingular*

$$E : y^2 = x^3 + ax + b \qquad \textit{and} \qquad j_E = 0; \tag{7.24}$$

- *if* $\mathrm{char}(\mathbb{K}) = 2$ *and* E *is ordinary*

$$y^2 + xy = x^3 + ax^2 + b \qquad \textit{and} \qquad j_E = \frac{1}{b}; \tag{7.25}$$

- *if* $\mathrm{char}(\mathbb{K}) = 2$ *and* E *is supersingular*

$$E : y^2 + a_3 y = x^3 + a_4 x + a_6 \qquad \textit{and} \qquad j_E = 0. \tag{7.26}$$

Definition 7.18 (Twist) Two non-isomorphic elliptic curves E and E' over $\mathbb{K}$ such that $j_E = j_{E'}$ are called *twists* (of one another).

An elliptic curve and a twist are isomorphic over the algebraic closure $\bar{\mathbb{K}}$. The degree of a twist is the degree of the smallest extension $\mathbb{K}'/\mathbb{K}$ such that the two curves are isomorphic over $\mathbb{K}'$.

Proposition 7.19 *Any curve has a quadratic twist. Any twist of an ordinary elliptic curve is quadratic.*

7.1.4 Isogenies

Definition 7.20 (Isogeny) Let E and E' be elliptic curves, an *isogeny* $E \to E$ is a morphism of varieties that preserves the point at infinity.

It turns out that isogenies preserve the group structure.

Theorem 7.21 *Let* $\mathcal{I} : E \to E'$ *be an isogeny, then it is a group morphism* $E(\bar{\mathbb{K}}) \to E'(\bar{\mathbb{K}})$*. It is surjective and its kernel is finite. Furthermore, if* $\mathcal{I}$ *is defined over* $\mathbb{K}$*, then its restriction to* $E(\mathbb{K})$ *is a group morphism* $E(\mathbb{K}) \to E'(\mathbb{K})$*.*

Definition 7.22 (Degree) Let $\mathcal{I}$ be an isogeny and define

$$\begin{aligned} \mathcal{I}^* : \bar{\mathbb{K}}(E') &\to \bar{\mathbb{K}}(E) \\ f &\to f \circ \mathcal{I}. \end{aligned} \tag{7.27}$$

The *separable (resp. inseparable) degree* of $\mathcal{I}$, denoted by $\deg_s \mathcal{I}$ (resp. $\deg_i \mathcal{I}$), is the separable (resp. inseparable) degree of the field extension $\bar{\mathbb{K}}(E)/\mathcal{I}^*(\bar{\mathbb{K}}(E'))$. The *degree* of $\mathcal{I}$ is $\deg \mathcal{I} = \deg_s \mathcal{I} \deg_i \mathcal{I}$.

An isogeny is called *separable* if $\deg_i \mathcal{I} = 1$, *inseparable* otherwise. It is called *purely inseparable* if $\deg_s = 1$.

THEOREM 7.23 *Let $\mathcal{I}$ be an isogeny, its kernel contains $\deg_s \mathcal{I}$ elements.*

Two curves are said to be isogenous if there is an isogeny between them; ℓ-isogenous if it has degree ℓ.

One example of isogeny is the map $[m]$, it is a separable isogeny of degree m^2. If $\mathbb{K}$ has characteristic p, then the map

$$\phi : (x, y) \mapsto (x^p, y^p) \tag{7.28}$$

is a purely inseparable isogeny of degree p, called the *Frobenius isogeny*. If $\mathbb{K}$ is a perfect field, any purely inseparable isogeny is a power of ϕ.

THEOREM 7.24 *Any isogeny can be factored into a product of a separable and a purely inseparable isogeny.*

One important property about isogenies is that they factor the multiplication by m map.

DEFINITION 7.25 (Dual isogeny) Let $\mathcal{I} : E \to E'$ be a degree m isogeny. There exists an unique isogeny $\hat{\mathcal{I}} : E' \to E$, called the *dual isogeny* such that

$$\mathcal{I} \circ \hat{\mathcal{I}} = [m]_E \qquad \text{and} \qquad \hat{\mathcal{I}} \circ \mathcal{I} = [m]_{E'}.$$

By endomorphism we mean an isogeny $E \to E$. The multiplication maps are endomorphisms, thus $\mathrm{End}(E)$ contains a copy of $\mathbb{Z}$. The main theorem about the endomorphism ring $\mathrm{End}(E)$ is the following.

THEOREM 7.26 *The endomorphism ring is either isomorphic to $\mathbb{Z}$, or to an order in a quadratic imaginary field, or to an order in a quaternion algebra.*

If $\mathrm{char}(\mathbb{K}) \neq 0$, we can exclude the case $\mathbb{Z}$, because $\mathrm{End}(E)$ contains the Frobenius isogeny. Furthermore, in the two cases

- $\mathrm{char}(\mathbb{K}) = 0$,
- $\mathbb{K}$ perfect and E ordinary,

$\mathrm{End}(E)$ cannot be an order in a quaternion algebra, thus it is commutative.

7.2 Curves over $\mathbb{C}$

Elliptic curves defined over $\mathbb{C}$ have a very simple structure.

DEFINITION 7.27 (Lattice) A *lattice* $\Lambda \subset \mathbb{C}$ is a discrete additive subgroup of $\mathbb{C}$ that contains a basis of the $\mathbb{R}$-vector space $\mathbb{C}$. Two lattices Λ_1, Λ_2 are said to be *homothetic* if $\Lambda_1 = \alpha\Lambda_2$ for some $\alpha \in \mathbb{C}^*$.

As a group, a lattice is isomorphic to $\mathbb{Z} \times \mathbb{Z}$. Two elements $\omega_1, \omega_2 \in \Lambda$ such that

$$\Lambda = \omega_1\mathbb{Z} + \omega_2\mathbb{Z} \tag{7.29}$$

are called a *basis* of Λ. The quotient $\mathbb{C}/\Lambda$ is called a *complex torus*.

DEFINITION 7.28 (Elliptic function) Let Λ be a lattice. An *elliptic function* on Λ is a meromorphic function f on $\mathbb{C}$ such that

$$f(z + \omega) = f(z) \tag{7.30}$$

for any $\omega \in \Lambda$. The set of elliptic functions on Λ is denoted by $\mathbb{C}(\Lambda)$.

An example of elliptic function is the *Weierstrass ℘-function*

$$\wp(z;\Lambda) = \frac{1}{z^2} + \sum_{\omega\in\Lambda\setminus\{0\}} \frac{1}{(z-\omega)^2} - \frac{1}{\omega^2}. \tag{7.31}$$

Theorem 7.29 *Let Λ be a lattice, then $\mathbb{C}(\Lambda) = \mathbb{C}(\wp(z), \wp'(z))$.*

For $k > 1$, we define the *Eisenstein series* G_{2k} as

$$G_{2k}(\Lambda) = \sum_{\omega\in\Lambda\setminus\{0\}} \frac{1}{\omega^2}. \tag{7.32}$$

Theorem 7.30 *The Laurent series expansion of $\wp(z)$ at 0 is*

$$\wp(z) = \frac{1}{z^2} + \sum_{k=1}^{\infty} (2k+1) G_{2k+2} z^{2k}. \tag{7.33}$$

At any $z \notin \Lambda$, the function $\wp(z)$ satisfies the differential equation

$$\wp'^2 = 4\wp - 60G_4\wp - 140G_6. \tag{7.34}$$

We set $g_2(\Lambda) = 60G_4(\Lambda)$ and $g_3(\Lambda) = 140G_6(\Lambda)$.

Theorem 7.31 *Let E be the curve*

$$E \,:\, y^2 = 4x^3 - g_2(\Lambda)x - g_3(\Lambda). \tag{7.35}$$

The map

$$\begin{aligned} \phi : \mathbb{C}/\Lambda &\to E(\mathbb{C}) \\ z &\mapsto \begin{cases} \mathcal{O} & \text{if } z = 0, \\ (\wp(z), \wp'(z)) & \text{otherwise,} \end{cases} \end{aligned} \tag{7.36}$$

is an isomorphism of Riemann surfaces and a group homomorphism.

If Λ is a lattice, we denote by E_Λ the elliptic curve corresponding to it as in Eq. (7.35).

Theorem 7.32 *Let Λ_1, Λ_2 be lattices and let $\alpha \in \mathbb{C}^*$ such that $\alpha\Lambda_1 \subset \Lambda_2$. The map*

$$\begin{aligned} \phi_\alpha : \mathbb{C}/\Lambda_1 &\to \mathbb{C}/\Lambda_2 \\ z &\mapsto \alpha z \end{aligned} \tag{7.37}$$

is holomorphic. The map $E_{\Lambda_1} \to E_{\Lambda_2}$ induced by ϕ_α is an isogeny.

The correspondences we just defined are actually equivalences.

Theorem 7.33 *The category of elliptic curves over $\mathbb{C}$ with isogenies as maps is equivalent to the category of lattices up to homothety with maps $z \mapsto \alpha z$ such that $\alpha\Lambda_1 \subset \Lambda_2$.*

Thus to any elliptic curve there is an unique lattice Λ associated up to homothety; the addition law on $\mathbb{C}/\Lambda$ is just addition in $\mathbb{C}$, and an isogeny can be obtained by an $\alpha \in \mathbb{C}$ such that $\alpha\Lambda_1 \subset \Lambda_2$.

7.3 Curves over finite fields

We saw previously that an elliptic curve defined over a field of characteristic $p \neq 0$ has an isogeny

$$\phi : (x, y) \mapsto (x^p, y^p) \tag{7.38}$$

called Frobenius isogeny. If the curve E is defined over the field $\mathbb{F}_q$ with $q = p^d$, then d iterations of the Frobenius map give the isogeny

$$\phi_q \stackrel{\text{def}}{=} \phi^d : (x, y) \mapsto (x^q, y^q). \tag{7.39}$$

The map ϕ_q is an endomorphism of E and fixes the points of $E(\mathbb{F}_q)$; it is called the *Frobenius endomorphism* of E. It plays an important role in determining the cardinality of $E(\mathbb{F}_q)$.

Theorem 7.34 (Hasse) *The minimal polynomial of ϕ_q in $\mathrm{End}(E)$ is of the form*

$$\phi_q^2 - c\phi_q + q = 0 \tag{7.40}$$

with $|c| \leqslant 2\sqrt{q}$.

Since φ_q acts as the identity on $E(\mathbb{F}_q)$, this implies

$$\#E(\mathbb{F}_q) = q + 1 - c \qquad \text{with } |c| \leqslant 2\sqrt{q}. \tag{7.41}$$

Theorem 7.35 *Two elliptic curves E, E' defined over $\mathbb{F}_q$ are isogenous if and only if $\#E(\mathbb{F}_q) = \#E(\mathbb{F}_q)$.*

7.4 Modular polynomials

Recall that in $\mathbb{C}$ any elliptic curve is isomorphic to a complex torus $\mathbb{C}/\Lambda$. Since we take lattices up to homothety, we can scale Λ so that $(1, \tau)$ is a basis for it. Then $\mathbb{C}/\Lambda$ is ℓ-isogenous to $\mathbb{C}/(\mathbb{Z} + \ell\tau\mathbb{Z})$ via the map $z \mapsto \ell z$.

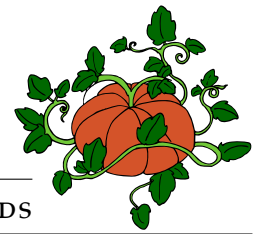
For any prime ℓ, the minimal polynomial over $\mathbb{C}$ of the modular function $j(\ell\tau)$ is called the *ℓ-th modular polynomial*, it is denoted by $\Phi_\ell(X, Y)$. It is a bivariate polynomial with coefficients in $\mathbb{Z}$, symmetric in X and Y, of degree $\ell + 1$ in X and Y.

Its importance comes from the fact that the roots of the univariate polynomial $\Phi_\ell(X, j_E)$ are the j-invariants of the elliptic curves ℓ-isogenous to E. From the knowledge of a pair of ℓ-isogenous j invariants, one can compute an isogeny using the algorithms of the next chapter.

The modular polynomial has $O(\ell^2)$ coefficients, and the coefficients themselves have logarithmic height $O(\ell \log \ell)$. For this reason minimal polynomials associated to other modular functions are often preferred to Φ_ℓ [Atk88, ES10]; for example, our implementations make use of *Atkin's modular polynomials* Φ_ℓ^* [Atk88].

Algorithms to compute the modular polynomial and its variants are described in [Mor95, BLS10, ES10].

8 Chapter



Computing isogenies over finite fields

Let E and E' be two elliptic curves defined over $\mathbb{K}$, by finding an *explicit isogeny* we mean to find a $\mathbb{K}$-rational function from $E(\bar{\mathbb{K}})$ to $E'(\bar{\mathbb{K}})$ such that the map it defines is an isogeny.

In this chapter we are interested in finding explicit isogenies of ordinary elliptic curves over finite fields. In what follows $\mathbb{F}_q$ will be a finite field of characteristic p, and d the positive integer such that $q = p^d$.

Parts of this chapter and of the following have been published in [DF11]. However, the complexity analysis we give in Proposition 8.6 benefits from recent advances on the computation of modular polynomials [BLS10]; this in turn changes the relative ranking of the algorithms of this chapter in terms of complexity. We also present a new algorithm in Section 8.9.

8.1 Overview

The problem of computing an explicit degree ℓ isogeny between two given elliptic curves over $\mathbb{F}_q$ was originally motivated by point counting methods based on Schoof's algorithm [Atk88, Elk98, Sch95]. A review of the most efficient algorithms to solve this problem is given in [BMSS08], together with a new quasi-optimal algorithm that we will review in Section 8.3.

All the algorithms of [BMSS08] only work when $\ell \ll p$. The case where p is small compared to ℓ was first treated by Couveignes in [Cou94], making use of formal groups. The complexity of his method is $\tilde{O}(\ell^3 \log q)$ operations in $\mathbb{F}_p$ assuming p is constant, however it has an exponential complexity in $\log p$.

Later, Lercier [Ler96] gave an algorithm specific to characteristic 2, that uses some linear properties of the problem to build a linear system from whose solution the isogeny can be deduced. Its complexity is conjectured to be $\tilde{O}(\ell^3 \log q)$ operations in $\mathbb{F}_p$, but it has a much better constant factor than [Cou94]. At the moment we write, this is by many orders of magnitude the fastest algorithm to solve practical instances of the problem when $p = 2$, thus being the *de facto* standard for cryptographic use.

Couveignes, again, proposed an algorithm in [Cou96] working for any p, based on the structure of the p^k torsion of ordinary elliptic curves. Using improvements from [Cou00, DFS09, DF11], this algorithm has a quadratic complexity in ℓ. We review the original algorithm as well as its improved variants in Sections 8.5 to 8.9.

After the discovery of p-adic alternatives to Schoof's algorithm [Sat00, FGH00], interest in computing isogenies in small characteristic was lost for some time. However, other cryptographic applications of isogenies soon appeared. The GHS attack uses Weil descent to reduce the discrete logarithm problem (DLP) of an

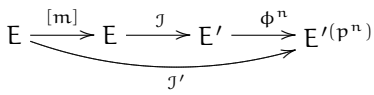


FIGURE 8.1: Factorization of an isogeny. ϑ' has kernel $E[m] \oplus \ker \vartheta$.

elliptic curve over a binary field of composite degree to the DLP of an hyperelliptic curve over a smaller field [GHS02b, GHS02a, Hes03]. A similar application is the reduction of the DLP of some genus 3 hyperelliptic curves to the DLP of genus 3 non-hyperelliptic curves [Smi09]. Isogeny graphs have been used to construct hash functions [CLG09] and to compute Hilbert class polynomials and modular polynomials [Sut10, BLS10]. New cryptographic protocols based on isogenies have also been proposed: Rostovtsef and Stolbunov [RS06] construct a Diffie-Hellman key exchange based on a DLP-like problem in a cycle of isogenous curves; Teske [MMT01, Tes06] constructs a trapdoor cryptosystem by hiding a GHS-vulnerable curve behind a random path of isogenies.

On the wave of the renewed interest for isogenies, two p-adic algorithms were recently proposed by Joux and Lercier [JL06] and Lercier and Sirvent [LS08] to compute isogenies in arbitrary characteristic. The former method has complexity $\tilde{O}(\ell^2(1+\ell/p)\log q)$ operations in $\mathbb{F}_p$, which makes it well adapted to the case where $p \sim \log q$. The latter has binary complexity $\tilde{O}(\ell^3)$, it is the best algorithm to compute isogenies when the characteristic is not fixed. We review the second algorithm in Section 8.4.

8.2 Vélu formulas

Since an isogeny can be uniquely factored in the product of a separable and a purely inseparable isogeny, we focus on the problem of computing explicit separable isogenies. Furthermore one can factor out multiplication-by-m maps, thus reducing the problem to compute explicit separable isogenies with cyclic kernel (see figure 8.2).

In the rest of this chapter, unless otherwise stated, by ℓ-isogeny we mean a separable isogeny with kernel isomorphic to $\mathbb{Z}/\ell\mathbb{Z}$.

For any finite subgroup $G \subset E(\bar{\mathbb{K}})$, Vélu formulas [Vél71] give in a canonical way an elliptic curve $\bar{E}$ and an explicit separable isogeny $\vartheta : E \to \bar{E}$ such that $\ker \vartheta = G$. The isogeny is $\mathbb{K}$-rational if and only if the polynomial vanishing on the abscissas of G belongs to $\mathbb{K}[X]$.

The isogeny computed by Vélu formulas is the map

$$\vartheta(P) = \left(x(P) + \sum_{Q \in G \setminus \{O\}} x(P+Q) - x(Q), \right.$$
$$\left. y(P) + \sum_{Q \in G \setminus \{O\}} y(P+Q) - y(Q) \right). \quad (8.1)$$

Using the addition formulas it is straightforward to obtain the coefficients of the curve $\bar{E}$ and the explicit isogeny. For simplicity, we do so only for the case $p \geqslant 3$

and E in the form

$$E : y^2 = x^3 + a_2x^2 + a_4x + a_6 \tag{8.2}$$

(note that this is always possible by Proposition 7.17).

We set $G^* = G \setminus \{\mathcal{O}\}$. We denote by f, f' the two functions in $\mathbb{K}(E)$

$$\begin{aligned} f(P) &= x(P)^3 + a_2x(P)^2 + a_4x(P) + a_6, \\ f'(P) &= 3x(P)^2 + 2a_2x(P) + a_4. \end{aligned} \tag{8.3}$$

From the addition formulas, after some calculations (see Appendix C for an automatic proof of this calculation), Eq. (8.1) is equivalent to

$$\mathcal{I}(x,y) = \left(x + \sum_{Q \in G^*} \frac{f'(Q)}{x - x(Q)} + \frac{2f(Q)}{(x-x(Q))^2}, \right.$$
$$\left. y + \sum_{Q \in G^*} -\frac{yf'(Q)}{(x-x(Q))^2} - \frac{4yf(Q)}{(x-x(Q))^3} \right). \tag{8.4}$$

Observe that if $Q \in G^*$ is a 2-torsion point, then $f(Q) = 0$; while if Q is not a 2-torsion point, $x(Q)$ is counted twice in the sum of the previous equation. Then, the denominator of $\mathcal{I}_x$ is

$$h(x) \stackrel{\text{def}}{=} \prod_{Q \in G^*} (x - x(Q)), \tag{8.5}$$

and by Eq. (8.4) it is evident that

$$\mathcal{I}(x,y) = \left(\frac{g(x)}{h(x)}, y \left(\frac{g(x)}{h(x)} \right)' \right) \tag{8.6}$$

for some polynomial g. Finally, if we set

$$t = \sum_{Q \in G^*} f'(Q), \qquad u = \sum_{Q \in G^*} 2f(Q), \qquad w = u + \sum_{Q \in G^*} x(Q)f'(Q), \tag{8.7}$$

the isogenous curve has equation

$$\bar{E} : y^2 = x^3 + a_2x^2 + (a_4 - 5t)x + a_6 - 4a_2t - 7w. \tag{8.8}$$

Remark 8.1. Eqs. (8.4) and (8.8) can be used to deduce the isogeny and the image curve from the knowledge of the kernel. It is often more convenient to use the reformulation given by Elkies [Elk98]

$$\frac{g(x)}{h(x)} = x + \sum_{Q \in G^*} x - x(Q) - \frac{f'(x)}{x - x(Q)} + \frac{2f(x)}{(x-x(Q))^2} \tag{8.9}$$

(this equality is shown in Appendix C, too), which implies

$$\frac{g(x)}{h(x)} = \ell x - p_1 - f'(x)\frac{h'(x)}{h(x)} - 2f(x)\left(\frac{h'(x)}{h(x)} \right)', \tag{8.10}$$

where p_1 is the first power sum of h.

Thus, from the knowledge of $h(x)$, one can deduce the isogeny and the isogenous curve in $O(M(\deg \mathcal{I}))$ operations in $\mathbb{K}$.

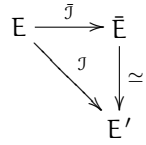


FIGURE 8.2: Using Vélu formulas to compute an explicit isogeny.

Given two curves E and E', Vélu formulas reduce the problem of finding an explicit isogeny between E and E' to that of finding the kernel of an isogeny between them. Once the polynomial $h(X)$ vanishing on $\ker \jmath$ is found, the explicit isogeny is computed composing Vélu formulas with the isomorphism between $\bar{E}$ and E' as in figure 8.2.

8.3 BMSS

In this section we present the BMSS algorithm [BMSS08] to compute isogenies of degree $\ell \neq p$ in characteristic 0 or $p \gg \ell$. It takes as input the integer ℓ and two elliptic curves E and E' over a finite field $\mathbb{F}_q$ defined by *normalized models* (see definitions below). It outputs the explicit isogeny using $O(M(\ell)\log\ell)$ operations in $\mathbb{F}_q$, or $O(M(\ell))$ in case the sum of the abscissas of the kernel of the isogeny is known.

Because of the assumption on the characteristic, we can assume $p \neq 2$ and the curves to be in the form

$$\begin{aligned} E &: y^2 = x^3 + a_2x^2 + a_4x + a_6, \\ E' &: y^2 = x^3 + a_2'x^2 + a_4'x + a_6'. \end{aligned} \tag{8.11}$$

Then, any isogeny $\jmath : E \to E'$ of odd degree is of the form

$$\jmath(x,y) = \left(\frac{g(x)}{h(x)}, cy\left(\frac{g(x)}{h(x)}\right)'\right), \tag{8.12}$$

with $c \in \bar{\mathbb{K}}$, and g, h monic polynomials in $\bar{\mathbb{K}}[X]$ (this is a consequence of Vélu formulas).

DEFINITION 8.2 (Normalized isogeny) An explicit isogeny given by Eq. (8.12) is said to be *normalized* if $c = 1$.

Given two ℓ-isogenous curves E and E', Weierstrass equations for them such that the explicit ℓ-isogeny $\jmath : E \to E'$ is normalized, are called *ℓ-normalized models* for those elliptic curves.

It is noteworthy that Vélu formulas output normalized models and a normalized isogeny. Normalized models naturally arise in point counting: in fact in the Schoof-Elkies-Atkin algorithm [Atk88, Elk98, Sch95] one factors the modular polynomial Φ_ℓ to obtain j-invariants of curves ℓ-isogenous to E. As a consequence of Vélu formulas, it is possible to obtain normalized models for such curves from the knowledge of the partial derivatives of Φ_ℓ. Details can be found in [Sch95, Mor95, Elk98, Ler97].

Algorithm 8.1: Solve differential equation

Input : $\mu > 1$, $\alpha \in \mathbb{K}$, $\beta \in \mathbb{K}^*$, $H \in \mathbb{K}[t]$, $G \in \mathbb{K}[[x]]$.
Output : $S \in \mathbb{K}[[x]]$, solution to $S'^2 = (H \circ S)G$ modulo x^{2^μ}.

1: let $U \leftarrow 1/\beta + O(x)$, $J \leftarrow 1/\sqrt{H(\alpha)} + O(x)$, $V \leftarrow \sqrt{H(\alpha)} + O(x)$;
2: let $S \leftarrow \alpha + \beta x + \frac{G'(0)H(\alpha)+G(0)H'(\alpha)\beta}{4\beta} x^2 + O(x^3)$;
3: for all $d \in \{2, 2^2, \ldots, 2^{\mu-1}\}$ do
4: $\quad U \leftarrow U(2 - S'U) \mod x^d$;
5: $\quad V \leftarrow (V + (H \circ S)J(2 - VJ))/2 \mod x^d$;
6: $\quad J \leftarrow J(2 - VJ)) \mod x^d$;
7: $\quad S \leftarrow S + V \int \left((H \circ S)G - S'^2 \right) UJ/2 \mod x^{2d+1}$;
8: output S.

Our goal is to compute the rational fraction $\frac{g(x)}{h(x)}$. From the fact that $\mathcal{I}$ is normalized and from Eq. (8.12) we deduce

$$(x^3 + a_2x^2 + a_4x + a_6)\left(\frac{g(x)}{h(x)}\right)'^2 = \left(\frac{g(x)}{h(x)}\right)^3 + a_2'\left(\frac{g(x)}{h(x)}\right)^2 + a_4'\frac{g(x)}{h(x)} + a_6'. \tag{8.13}$$

The key idea is to find a power series solution to this differential equation and then deduce the rational fraction.

However, we do not know the initial condition at 0, and we cannot look for an expansion at infinity either, because the degree of g is greater than the degree of h. Instead we set

$$S(x) = \sqrt{\frac{h(1/x^2)}{g(1/x^2)}} \quad \Leftrightarrow \quad \frac{g(x)}{h(x)} = \frac{1}{S(1/\sqrt{x})^2}, \tag{8.14}$$

so that $S(x) = x + O(x^3)$ from the monicity of g and h. Now $S(x)$ satisfies the differential equation

$$(a_6x^6 + a_4x^4 + a_2x^2 + 1)S'^2 = 1 + a_2'S^2 + a_4'S^4 + a_6'S^6, \tag{8.15}$$

hence we can use a Newton iteration to find a power series solution. In [BMSS08, 2.4], a generic iteration to solve Eq. (8.15) is used; here we present a more efficient iteration due to Lercier and Sirvent [LS08].

Let

$$G = \frac{1}{1 + a_2x^2 + a_4x^4 + a_6x^6}, \qquad H = 1 + a_2't^2 + a_4't^4 + a_6't^6, \tag{8.16}$$

Lercier and Sirvent give an algorithm to find a solution in $\mathbb{K}[[x]]$ of any equation of the form

$$S'^2 = (H \circ S)G, \tag{8.17}$$

with $G \in \mathbb{K}[[x]]$ and $H \in \mathbb{K}[t]$.

THEOREM 8.3 *Let $\mathbb{K}$ be a field of characteristic 0 or $p > 2^\mu$. Let α, β, H, G be the inputs to algorithm 8.1 such that $G(0)H(\alpha) = \beta^2$. Then Algorithm 8.1 computes a solution to*

$$S'^2 = (H \circ S)G, \quad S(0) = \alpha, \quad S'(0) = \beta \tag{8.18}$$

modulo x^{2^μ} using $O(M(2^\mu))$ operations in $\mathbb{K}$.

Proof. The complete proof is quite long and can be found in [LS08]; here we just give a sketch of it.

Let t be a solution to Eq. (8.18) modulo x^{d+1} and let h be such that

$$S = t + h \mod x^{2d+1}, \tag{8.19}$$

so that x^{d+1} divides h. Then x^{2d} divides h'^2 and, by Eq. (8.18)

$$2t'h' + t'^2 = G(x)H(t + h) \mod x^{2d}. \tag{8.20}$$

Using the Taylor expansion of H at t, we get the linearized differential equation

$$2t'h' + t'^2 = G(x)H(t) + G(x)H'(t)h \mod x^{2d} \tag{8.21}$$

with initial condition $h(0) = 0$. By Eq. (2.9), this equation has solution

$$h = \frac{1}{J} \int \frac{(G(x)H(t) - t'^2)J}{2t'} \, dx, \tag{8.22}$$

where J is

$$J = \exp\left(-\int \frac{G(x)H'(t)}{2t'} \, dx\right). \tag{8.23}$$

The key observation is that, in order to compute the above solution to precision x^{2d+1}, J must only be known to precision x^d. But t is a solution of (8.18) modulo x^{d+1}, thus

$$\frac{G(x)H'(t)}{2t'} = \frac{H'(t)t'}{2H(t)} \mod x^d, \tag{8.24}$$

hence

$$J = \exp\left(-\frac{1}{2}\log H(t)\right) = \frac{1}{\sqrt{H(t)}}. \tag{8.25}$$

Then, at each iteration, the algorithm computes the quantities

$$S, \quad U = 1/S', \quad V = \sqrt{H \circ S}, \quad J = 1/V, \tag{8.26}$$

doubling the precision at each iteration. Since the only operations are integrals and multiplications of power series, the i-th iteration costs $O(M(2^i))$ operations in $\mathbb{K}$, thus the last iteration dominates the complexity. 

Then, the algorithm to compute the isogeny goes as follows. The power series expansion of S is computed to precision 4ℓ, then we set

$$S(x) = xT(x^2), \quad R(x) = \frac{1}{T(x)^2}, \quad \text{so that} \quad \frac{g(x)}{h(x)} = xR(1/x). \tag{8.27}$$

Finally, the rational fraction is recovered by rational fraction reconstruction (see Section 2.2.6); the overall complexity is dominated by this last step.

Algorithm 8.2: BMSS

Input : $\ell > 1$, ℓ-normalized models of E and E'.
Output : An isogeny $\mathcal{I} : E \to E'$ of degree ℓ.
1: Compute $G(x) = 1/(1 + a_2x^2 + a_4x^4 + bx^6) \mod x^{4\ell-1}$;
2: find $S(x) \bmod x^{4\ell-1}$ using Algorithm 8.1;
3: let $T(x) = \sum_{i=0}^{2\ell-1} s_{2i+1}x^i$;
4: compute $R(x) = 1/T(x)^2 \mod x^{2\ell-1}$;
5: compute $\frac{g(x)}{h(x)}$ by rational fraction reconstruction.

Remark 8.4. Alternatively, if the sum of the abscissas of the kernel

$$p_1 = \sum_{Q \in G^*} x(Q) \tag{8.28}$$

is known, we can avoid the rational fraction reconstruction.

The idea is to recover the Newton sums $p_0, \dots, p_{\ell-1}$ of h from $\frac{g(x)}{h(x)}$. From Eq. (8.10) we deduce

$$\begin{aligned} &\frac{g(x)}{h(x)} = x + \sum_{i \geqslant 1} \frac{h_i}{x^i}, \quad \text{where, for any } i \geqslant 1 \\ &h_i = (2i+1)p_{i+1} + 2ia_2p_i + (2i-1)a_4p_{i-1} + (2i-2)a_6p_{i-2}. \end{aligned} \tag{8.29}$$

Thus, knowing $p_0 = \ell - 1$ and p_1 is enough to compute all the Newton sums up to $p_{\ell-1}$ using $O(\ell)$ operations (observe, in fact, that the equation for h_1 has only three non-zero summands).

From the power sums, we can recover $h(x)$ using Remark 5.12 in $O(M(\ell))$ operations. Then, $g(x)$ is obtained simply multiplying $\frac{g(x)}{h(x)}$ by $h(x)$, again in $M(\ell)$ operations.

Using this approach, we gain a logarithmic factor compared to the rational fraction reconstruction; and the number of coefficients of $S(x)$ to compute goes down to 2ℓ. This is similar to the trade-off we had in Remark 5.23.

The knowledge of p_1 (i.e. the coefficient of $x^{\ell-2}$ in h) may seem a rather bizarre requirement; however, in the Schoof-Elkies-Atkin algorithm this information is obtained, together with the normalized model for E', from the derivatives of the modular polynomial (see [Elk98, Mor95]), and this is why this algorithm has been developed.

8.4 Lercier-Sirvent

The integral at step 7 requires divisions by all the integers in the interval $[1, \dots, 2^\mu]$, thus, when $2^{\lceil \log_2(4\ell-1) \rceil} > p$, BMSS encounters a division by 0. A natural idea is to work in characteristic 0 by lifting the curves in the p-adics. However, lifting the Weierstrass models of E and E', there is no guarantee of obtaining a pair of ℓ-normalized models, thus BMSS cannot apply.

To circumvent this problem, Lercier and Sirvent [LS08] use Elkies' formulas to obtain normalized models in the p-adic, and then apply BMSS. The algorithm is summarized below; it requires $p \geqslant 5$ and it makes computations in an unramified extension of degree d of $\mathbb{Q}_p$, denoted by $\mathbb{Q}_q$.

ALGORITHM 8.3: Lercier-Sirvent

INPUT : $\ell > 1$, E, E' ℓ-isogenous defined over $\mathbb{F}_q$.
OUTPUT : An isogeny $\mathcal{I} : E \to E'$ of degree ℓ.
1: Take any lift $\bar{E} : y^2 = x^3 + \bar{a}x + \bar{b}$ of E in $\mathbb{Q}_q$;
2: Compute a root $\bar{j}'$ of $\Phi_\ell(X, j_{\bar{E}})$ in $\mathbb{Q}_q$ by lifting the solution $j_{E'}$;
3: Compute an ℓ-normalized model $\bar{E}'' : y^2 = x^3 + \bar{a}'x + \bar{b}'$ for $\bar{j}'$;
4: Apply BMSS to $\bar{E}$ and $\bar{E}''$ to obtain $\bar{\mathcal{I}} : \bar{E} \to \bar{E}''$;
5: Reduce $\bar{E}''$ and $\bar{\mathcal{I}}$ to E'' and $\mathcal{I}$ modulo p;
6: Apply an isomorphism $E' \cong E''$ to recover $\mathcal{I} : E \to E'$.

Step 3 uses Elkies' formulas [Elk98] to find the ℓ-normalized model of $\bar{j}'$; these formulas allow to compute a normalized model of the form $y^2 = x^3 + ax + b$ from the knowledge of $\partial\Phi_\ell/\partial X$ and $\partial\Phi_\ell/\partial Y$, and the sum of the abscissas of the kernel from the knowledge of $\partial^2\Phi_\ell/\partial X^2$, $\partial^2\Phi_\ell/\partial X\partial Y$ and $\partial^2\Phi_\ell/\partial Y^2$, using $O(\ell^2)$ operations in the base field ($\mathbb{Q}_q$, in this case). Analogous formulas exist for other types of modular polynomials, we address the interested reader to [Sch95, Mor95, Elk98, Ler97]. Notice that this step fails when $(j_E, j_{E'})$ is a singular point of the curve $X_0(\ell)$; this condition is very rare for ordinary curves of large discriminant, as pointed out in [Sch95, §7].

Computations in $\mathbb{Q}_q$ must be approximated to a certain precision. Lercier and Sirvent show the following fundamental property.

PROPOSITION 8.5 *If $p \geqslant 5$, on inputs ℓ, E, E', the previous algorithm computes the correct answer using at most $\left\lceil \log^2 \ell / \log p \right\rceil$ p-adic digits.*

Building on this, we now analyze the complexity of the algorithm, for simplicity we count operations in $\mathbb{F}_q$ rather than in $\mathbb{F}_p$.

PROPOSITION 8.6 *Algorithm* Lercier-Sirvent *computes an ℓ-degree isogeny in $\tilde{O}(\ell^2)$ operations in $\mathbb{F}_q$, plus $\tilde{O}(\ell^3)$ binary operations.*

Proof. We do not take into account the complexity of building the field $\mathbb{Q}_q$. Lifting E in $\mathbb{Q}_q$ can be done for free by taking a trivial lift. Since the computations in $\mathbb{Q}_q$ are truncated to a fixed precision, we only need the coefficients of Φ_ℓ mod $p^{\lceil \log^2 \ell / \log p \rceil}$; this can be computed using $O(\ell^3 \log^3 \ell \operatorname{loglog} \ell)$ binary operations and $O(\ell^2 \log^2 \ell)$ bits of space by [BLS10, Algorithm 6.1]. The output is a polynomial in $\mathbb{Z}_p[X, Y]$ having $O(\ell^2)$ coefficients, this requires an equivalent storage of $\tilde{O}(\ell^2)$ elements of $\mathbb{F}_p$.

Then, step 2 can be done in $\tilde{O}(\ell)$ operations using Hensel lifting. Steps 3 and 4 take $O(\ell^2)$ and $O(M(\ell) \log \ell)$ operations in $\mathbb{Q}_q$ respectively (slightly less if the sum of the abscissas of the kernel of the isogeny is computed by Elkies' formulas), this is equivalent to $\tilde{O}(\ell^2)$ operations in $\mathbb{F}_q$. The rest of the computation is negligible. Thus, the dominating step is 3. 

Remark 8.7. Our presentation of the algorithm slightly deviates from the original paper [LS08]. Since we want to compare it to Couveignes' algorithm, we assume that the elliptic curve E' isogenous to E is provided as an input, while in [LS08] it is assumed that only E is known.

The consequence is that in the original version, before step 2 one has to factor the univariate polynomial $\Phi(X, j_E)$ in $\mathbb{F}_q$ to find an isogenous j-invariant. Lercier

and Sirvent, citing [LN96], estimate this cost to be $O(M(\ell)\log q + M(\ell)\log \ell)$ operations in $\mathbb{F}_q$. This contribution must be added to the complexity announced in Proposition 8.6 if one wants to work in the original setting.

Another difference is that we rely on an algorithm to compute the modular polynomial Φ_ℓ in the ring $\mathbb{Z}/m\mathbb{Z}[X,Y]$, recently appeared in [BLS10]. This algorithm does not improve the binary complexity, however it permits to compute Φ_ℓ in $\mathbb{Z}_p[X,Y]$ truncated to the required precision using only $O(\ell^2 \log^2 \ell)$ bits of memory, instead of $\tilde{O}(\ell^3)$, thus improving the algebraic complexity.

Note. In the cases $p = 2, 3$, Elkies' formulas yield a curve over $\mathbb{Q}_q$ that reduces badly in $\mathbb{F}_q$. As a consequence, each iteration of algorithm 8.1 introduces some additional divisions by p, and Proposition 8.5 fails to hold. While it is still possible to apply Lercier-Sirvent in this case, its complexity gets much worse because of the higher p-adic precision needed.

In [LS08], Lercier and Sirvent say:

> "For $p = 2$ (or $p = 3$), Weierstrass models of the form $y^2 + xy = x^3 + a_2x^2 + a_6$ (or $y^2 = x^3 + a_2x^2 + a_6$) must be considered. This yields completely different equations... [The algorithm] can be easily extended to these fields but for the sake of simplicity we prefer to omit the details here."

It is true that in the case $p = 3$ it is possible to obtain, via isomorphism, normalized models for $\bar{E}$ and $\bar{E}''$ of the form $y^2 = x^3 + a_2x^2 + a_6$ that reduce well in $\mathbb{F}_q$. Hence, algorithm 8.1 can still be applied to solve the differential equation, and the isogeny can be computed using the same p-adic precision as in Proposition 8.5.

On the other hand, when $p = 2$, while it is still possible to obtain models of the form $y^2 + xy = x^3 + a_2x^2 + a_6$ that reduce well in $\mathbb{F}_q$, isogenies in such models do not verify an equation as simple as Eq. (8.13). We think that in this case the techniques known to solve differential equations are not enough to find a solution to this problem.

8.5 Couveignes' algorithm

In this section we describe Couveignes' algorithm to compute isogenies between ordinary elliptic curves in arbitrary characteristic, as it was originally presented in [Cou96]. In the next sections we will discuss more efficient variants of this algorithm; to distinguish between the variants, we call C2 this original version (it is to be understood that C1 would be the code-name of the other algorithm by Couveignes, appeared in [Cou94], that we will not present in this document).

C2 takes as input two elliptic curves E, E' and an integer ℓ prime to p, and it returns, if it exists, an $\mathbb{F}_q$-rational isogeny of degree ℓ between E and E'. It only works in odd characteristic.

8.5.1 The original algorithm

Suppose there exists an $\mathbb{F}_q$-rational isogeny $\mathcal{I} : E \to E'$ of degree ℓ. Since ℓ is prime to p one has $\mathcal{I}(E[p^k]) = E'[p^k]$ for any k.

Recall that $E[p^k]$ and $E'[p^k]$ are cyclic groups. C2 iteratively computes generators P_k, P'_k of $E[p^k]$ and $E'[p^k]$ respectively. Now C2 makes the guess $\mathcal{I}(P_k) = P'_k$;

then, if $\mathcal{I}$ is given by rational fractions as in (8.6),

$$\frac{g\big(x([i]P_k)\big)}{h\big(x([i]P_k)\big)} = x([i]P'_k) \quad \text{for } i \in \mathbb{Z}/p^k\mathbb{Z}, \tag{8.30}$$

and Vélu formulas imply that

$$\deg g = \deg h + 1 = \ell. \tag{8.31}$$

Using (8.30) one can compute the rational fraction $\frac{g(X)}{h(X)}$ through Cauchy interpolation over the points of $E[p^k]$ for k large enough. C2 takes $p^k \geqslant 4\ell + 1$, interpolates the rational fraction and then checks that it corresponds to the restriction of an isogeny to the x-axis. If this is the case, the whole isogeny is computed through Vélu formulas and the algorithm terminates. Otherwise the guess $\mathcal{I}(P_k) = P'_k$ was wrong, then C2 computes a new generator for $E'[p^k]$ and starts over again.

We now go through the details of the algorithm.

The p-torsion. The computation of the p-torsion points follows from the work of Gunji [Gun76]. Here we suppose $p \neq 2$.

Definition 8.8 Let E have equation $y^2 = f(x)$. The *Hasse invariant* of E, denoted by H_E, is the coefficient of X^{p-1} in $f(X)^{\frac{p-1}{2}}$.

Gunji shows the following proposition.

Proposition 8.9 *Let $c_1, \ldots, c_{p-1}$ be the roots of $X^{p-1} - H_E$ in its splitting field. The abscissas of the abscissas of the p-torsion points of E are given by*

$$X_i^p = \frac{\Delta_0^2 - a_6 c_i^2 \Delta_1^2}{4c_i^2},$$

where Δ_0 and Δ_1 are the determinants of the matrices shown in Figure 8.3.

with $r = \frac{p-1}{2}$, $\alpha_\nu = \nu(\nu-1)a_6$, $\beta_\nu = \nu(\nu - \frac{1}{2})a_4$, $\gamma_\nu = \nu^2 a_2$ and $\delta_\nu = \nu(\nu + \frac{1}{2})$ (Δ_1 is set to 1 when $r = 1$).

In what follows we let c be any $(p-1)$-th root of H_E. Then Gunji's formulas imply that the p-torsion points are defined in $\mathbb{F}_q[c]$ and their abscissas are defined in $\mathbb{F}_q[c^2]$.

The p^k-torsion. p^k-torsion points are iteratively computed via p-descent. The basic idea is to split the multiplication map as $[p] = \phi \circ V$ and invert each of the components. The purely inseparable isogeny ϕ is just a Frobenius map and the separable isogeny V can be computed by Vélu formulas once the p-torsion points are known. Although this is reasonably efficient, pulling V back may involve factoring polynomials of degree p in some extension field.

A finer way to do the p-descent, as suggested in the original paper [Cou96], is to use the work of Voloch [Vol90]. Suppose $p \neq 2$, let E and $\widetilde{E}$ have equations respectively

$$\begin{aligned} y^2 &= f(x) = x^3 + a_2 x^2 + a_4 x + a_6 \,, \\ \tilde{y}^2 &= \tilde{f}(\tilde{x}) = \tilde{x}^3 + \sqrt[p]{a_2}\tilde{x}^2 + \sqrt[p]{a_4}\tilde{x} + \sqrt[p]{a_6} \,, \end{aligned}$$

$$\Delta_0 = \begin{vmatrix} \beta_1 & \alpha_2 & 0 & 0 & \dots & 0 \\ \delta_1 & \gamma_2 - c^2 & \beta_3 & \alpha_4 & \ddots & \vdots \\ 0 & \delta_2 & \gamma_3 - c^2 & \beta_4 & \ddots & 0 \\ \vdots & \ddots & \delta_3 & \ddots & \ddots & \alpha_r \\ \vdots & & \ddots & \ddots & \ddots & \beta_r \\ 0 & \dots & \dots & 0 & \delta_{r-1} & \gamma_r - c^2 \end{vmatrix},$$

$$\Delta_1 = \begin{vmatrix} \gamma_2 - c^2 & \beta_3 & \alpha_4 & \dots & 0 \\ \delta_2 & \gamma_3 - c^2 & \beta_4 & \ddots & \vdots \\ 0 & \delta_3 & \ddots & \ddots & \alpha_r \\ \vdots & \ddots & \ddots & \ddots & \beta_r \\ 0 & \dots & 0 & \delta_{r-1} & \gamma_r - c^2 \end{vmatrix}, \tag{8.32}$$

Figure 8.3: The determinants Δ_0 and Δ_1 appearing in Gunji's formulas.

set

$$\tilde{f}(X)^{\frac{p-1}{2}} = \alpha(X) + H_{\widetilde{E}} X^{p-1} + X^p \beta(X) \tag{8.33}$$

with $\deg \alpha < p - 1$ and $H_{\widetilde{E}}$ the Hasse invariant of $\widetilde{E}$. Voloch shows the following proposition.

Proposition 8.10 *Let* $\tilde{c} = \sqrt[p-1]{H_{\widetilde{E}}}$, *the cover of* $\widetilde{E}$ *defined by*

$$C : \ \tilde{z}^p - \tilde{z} = \frac{\tilde{y}\beta(\tilde{x})}{\tilde{c}^p} \tag{8.34}$$

is an étale cover of degree p *and is isomorphic to* E *over* $\mathbb{F}_q[\tilde{c}]$*; the isomorphism is given by*

$$\begin{cases} (\tilde{x}, \tilde{y}) = V(x, y) \\ \tilde{z} = -\dfrac{y}{\tilde{c}^p} \displaystyle\sum_{i=1}^{p-1} \frac{1}{x - x([i]P_1)} \end{cases} \tag{8.35}$$

where P_1 *is a primitive* p*-torsion point of* E.

The descent is then performed as follows: starting from a point P on E, first pull it back along ϕ, then take one of its pre-images in C by solving equation (8.34), finally use equation (8.35) to land on a point P' in E. The proposition guarantees that $[p]P' = P$. The descent is pictured in figure 8.4.

The reason why this is more efficient than a standard descent is the shape of equation (8.34): it is an Artin-Schreier equation and it can be solved by the techniques of Chapter 6 (or by linear algebra, as was suggested in [Cou96]). Once a solution $\tilde{z}$ to (8.34) is known, solving in x and y the bivariate polynomial system (8.35) takes just a GCD computation (explicit formulas were given by Lercier

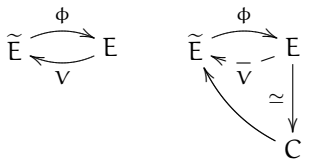


FIGURE 8.4: Two ways of doing the p-descent: standard on the left and via a degree p cover on the right

in [Ler97, §6.2], we give some slightly improved ones in Section 9.1). Compare this with a generic factoring algorithm needed by standard descent.

In this section we assume that the Artin-Schreier equations are solved using linear algebra. The impact of Chapter 6 over Couveignes' algorithm is discussed in the next section.

Cauchy interpolation. Interpolation reconstructs a polynomial from the values it takes on some points; Cauchy interpolation reconstructs a rational fraction. As we saw in Section 2.2.6, the Cauchy interpolation algorithm is divided in two phases: first find the polynomial P interpolating the evaluation points, then use the Euclidean algorithm to find a rational fraction congruent to P modulo the polynomial vanishing on the points.

Cauchy interpolation needs $n+1$ points to reconstruct a degree $(k, n-k)$ rational fraction. This, together with (8.31), justifies the choice of k such that $p^k \geqslant 4\ell+1$. Some of our variants of C2 will interpolate only on the primitive p^k-torsion points, thus requiring the slightly larger bound $\varphi(p^k) \geqslant 4\ell+1$. This is not very important to our asymptotical analysis since in both cases $p^k \in O(\ell)$.

Recognising the isogeny. Once the rational fraction $\frac{g(X)}{h(X)}$ has been computed, one has to verify that it is indeed an isogeny. The first test is to check that the degrees of g and h match equation (8.31): if this is not the case, the equation can be discarded right away and the algorithm can go on with the next trial. Next, one can check that h is indeed the square of a polynomial (or, if ℓ is even, the product of one factor of the 2-division polynomial and a square polynomial). These two tests are usually enough to detect an isogeny. In case a higher confidence is needed, one can evaluate the rational fraction on some random points of E and check that it is indeed a group morphism. Finally, if a deterministic proof is needed, one can compute the ℓ-division polynomial modulo h and verify that it is equal to 0.

8.5.2 *The case* $p = 2$

The algorithm, as we presented it, only works when $p \neq 2$, it is however an easy matter to generalise it. The only phase that does not work is the computation of the p^k-torsion points. For curves in the simplified Weierstrass form (7.25) the only 2-torsion point is $(0, \sqrt{b})$.

Voloch formulas are hard to adapt, nevertheless a 2-descent on the Kummer variety of E can easily be performed since the doubling formula reads

$$x([2]P) = \frac{b}{x(P)^2} + x(P)^2 = \phi\left(\frac{\sqrt{b} + x(P)^2}{x(P)}\right) = \phi \circ V\,. \tag{8.36}$$

Given point x_P on K_E, a pull-back along ϕ gives a point $\tilde{x}_P$ on $K_{\tilde{E}}$. Then pulling V back amounts to solve

$$x^2 + \tilde{x}_P x = \sqrt{b} \tag{8.37}$$

and this can be turned in an Artin-Schreier equation through the change of variables $x \to x'\tilde{x}_P$.

From the descent on the Kummer varieties one could deduce a full 2-descent on the curves by solving a quadratic equation at each step in order recover the y coordinate, but this would be too expensive. Fortunately, the y coordinates are not needed by the subsequent steps of the algorithm, thus one may simply ignore them. Observe in fact that even if K_E does not have a group law, the restriction of scalar multiplication is well defined and can be computed through Montgomery formulas as described in Remark 7.8. This is enough to compute all the abscissas of the points in $E[p^k]$ once a generator is known.

8.5.3 Complexity analysis

Analyzing the complexity of C2 is a delicate matter since the algorithm relies on some black-box computer algebra algorithms in order to deal with finite extensions of $\mathbb{F}_q$. The choice of the actual algorithms may strongly influence the overall complexity of C2. In this section we will only give some lower bounds on the complexity of C2, since a much more accurate complexity analysis will be carried out in Section 8.6.

p-Torsion. Applying Gunji formulas first requires to find c and c', $(p-1)$-th roots of H_E and $H_{E'}$, and build the field extension $\mathbb{F}_q[c] = \mathbb{F}_q[c']$. Independently of the actual algorithm used, observe that in the worst case $\mathbb{F}_q[c]$ is a degree $p-1$ extension of $\mathbb{F}_q$, thus simply representing one of its elements requires $\Theta(pd)$ elements of $\mathbb{F}_p$.

Subsequently, the main cost in Gunji's formulas is the computation of the determinant of a $\frac{p-1}{2} \times \frac{p-1}{2}$ quadri-diagonal matrix (see [Gun76]). This takes $\Theta(p^2)$ operations in $\mathbb{F}_q[c]$ by Gauss elimination, that is no less than $\Omega(p^3 d)$ operations in $\mathbb{F}_p$.

p^k-Torsion. During the p-descent, factoring of equations (8.34) or (8.37) may introduce some field extensions over $\mathbb{F}_q[c]$. Recall that an Artin-Schreier polynomial is either irreducible or totally split (see Proposition 6.7), so at each step of the p-descent we either stay in the same field or we take a degree p extension. This shows that in the worst case we have to take an extension of degree p^{k-1} over $F_q[c]$. The following proposition, which is a generalization of [Ler97, Proposition 26], states precisely how likely this case is.

Proposition 8.11 *Let E be an elliptic curve over $\mathbb{F}_q$, we denote by $\mathbb{U}_i$ the smallest field extension of $\mathbb{F}_q$ such that $E[p^i] \subset E(\mathbb{U}_i)$. For any $i \geqslant 1$, either $[\mathbb{U}_{i+1} : \mathbb{U}_i] = p$ or $\mathbb{U}_{i+1} = \mathbb{U}_i = \cdots = \mathbb{U}_1$.*

Before proving the proposition, we shall state a lemma. Its proof is elementary and can be found in [Ler97, §6.1].

LEMMA 8.12 *Let* p *be a prime and let* c *be prime to* p. *For any* $k > 1$, *let* $\mathrm{ord}_k(c)$ *be the order of* c *in* $\mathbb{Z}/p^k\mathbb{Z}$. *Then* $\mathrm{ord}_{k+1}(c) = \mathrm{ord}_k(c)$ *implies* $\mathrm{ord}_k(c) = \mathrm{ord}_{k-1}(c)$.

Proof of Propositon 8.11. Observe that the action of the Frobenius ϕ on $E[p]$ is just multiplication by the trace t, in fact the equation

$$\phi^2 - [t \bmod p] \circ \phi + [q \bmod p] = 0$$

has two solutions, namely $[t \bmod p]$ and $[0 \bmod p]$, but the second can be discarded since it would imply that ϕ has non-trivial kernel. By lifting this solution, one sees that the action of ϕ on the Tate module $\mathcal{T}_p(E)$ is equal to multiplication by some $\tau \in \mathbb{Z}_p$.

Let G be the absolute Galois group of $\mathbb{F}_q$, there is a well known action of G on $\mathcal{T}_p(E)$. Since G is generated by the Frobenius automorphism of $\mathbb{F}_q$, the restriction of this action to $E[p^k]$ is equal to the action (via multiplication) of the subgroup of $(\mathbb{Z}/p^k\mathbb{Z})^*$ generated by $\tau_k = \tau \bmod p^k$. Hence $[\mathbb{U}_k : \mathbb{F}_q] = \mathrm{ord}(\tau_k)$.

Then, for any $k > 1$, Lemma 8.12 applied to $\tau_{k+1} = \tau \bmod p^{k+1}$ shows that $\mathrm{ord}(\tau_{k+1}) = \mathrm{ord}(\tau_k)$ implies $\mathrm{ord}(\tau_k) = \mathrm{ord}(\tau_{k-1})$ and this concludes the proof.

Thus for any elliptic curve there is an i_0 such that $[\mathbb{U}_i : \mathbb{U}_1] = p^{i-i_0}$ for any $i \geqslant i_0$. This shows that the worst and the average case coincide since for any fixed curve $[\mathbb{U}_k : \mathbb{U}_1] \in \Theta(p^k)$ asymptotically. In this situation, one needs $\Theta(p^k d)$ elements of $\mathbb{F}_p$ to store an element of $\mathbb{U}_k$.

Now the last iteration of the p-descent needs to solve an Artin-Schreier equation in $\mathbb{U}_k$. To do this C2 precomputes the matrix of the $\mathbb{F}_q$-linear application $(x^q - x) : \mathbb{U}_k \to \mathbb{U}_k$ and its inverse, plus the matrix of the $\mathbb{F}_p$-linear application $(x^p - x) : \mathbb{F}_q \to \mathbb{F}_q$ and its inverse. The former is the most expensive one and takes $\Theta(p^{\omega k})$ operations in $\mathbb{F}_q$, that is $\Omega(p^{\omega k} d) = \Omega(\ell^\omega d)$ operations in $\mathbb{F}_p$, plus a storage of $\Theta(\ell^2 d)$ elements of $\mathbb{F}_p$. Observe that this precomputation may be used to compute any other isogeny with domain E.

After the precomputation has been done, C2 successively applies the two inverse matrices; details can be found in [Cou96, §2.4]. This costs at least $\Omega(\ell^2 d)$.

Interpolation. The most expensive part of Cauchy interpolation is the polynomial interpolation phase. In fact, simply representing a polynomial of degree $p^k - 1$ in $\mathbb{U}_k[X]$ takes $\Theta(p^{2k} d)$ elements of $\mathbb{F}_p$, thus at least $\Omega(\ell^2 d)$ operations are needed to interpolate unless special care is taken[1]. We will give more details on interpolation in Section 8.7.

Recognising the isogeny. The cost of testing for squareness of the denominator and of the other probabilistic tests is negligible compared to the rest of the algorithm. The cost of computing the ℓ-division polynomial modulo h is $O(M(\ell)\log \ell)$ operations in $\mathbb{F}_q$, thus, again, negligible.

Nevertheless it is important to realize that, on average, half of the $\varphi(p^k)$ mappings from $E[p^k]$ to $E'[p^k]$ must be tried before finding the isogeny, for only

[1] This contribution due to arithmetics in $\mathbb{U}_k$ had been underestimated in the complexity analysis of [Cou96], where an estimate of $\Omega(\ell d \log \ell)$ operations was given.

one of these mappings corresponds to it. This implies that the Cauchy interpolation step must be repeated an average of $\Theta(p^k)$ times, thus contributing a $\Omega(\ell^3 d)$ to the total complexity.

Summing up all the contributions one ends up with the following lower bound

$$\Omega(\ell^3 d + p^3 d) \tag{8.38}$$

plus a precomputation step whose cost is negligible compared to this one and a space requirement of $\Theta(\ell^2 d)$ elements. In the next sections we will see how to make all these costs drop.

8.5.4 *The case* $(p, \ell) \neq 1$

If we are interested in finding a separable isogeny whose degree is not prime to p, the best way is to compute the curve $\widetilde{E}$ such that $E = \widetilde{E}^{(p)}$, then compute an isogeny of degree ℓ/p between $\widetilde{E}$ and E' and finally compose it with the separable p-isogeny V from E to $\widetilde{E}$.

Observe however that C2 can be easily adapted to directly compute such an isogeny. In fact let $\nu = \nu_p(\ell)$, then $\mathcal{I}(E[p^k]) = E'[p^{k-\nu}]$. All one needs to do in this case is to modify the Cauchy interpolation so that it interpolates the rational function that sends a generator of $E[p^k]$ over a generator of $E'[p^{k-\nu}]$ and the other points accordingly. The maximum number of trials to do before finding the isogeny is $\varphi(p^{k-\nu})$, thus the overall complexity is

$$\Omega\left(\frac{\ell^3}{p^\nu} d + p^3 d\right) . \tag{8.39}$$

Although this method is less efficient than the first one, it will come handy in Section 8.9.

8.6 The algorithm C2-AS

One of the most expensive steps of C2 is the resolution of an Artin-Schreier equation in an extension field $\mathbb{U}_i$. We call C2-AS the variant of Couveignes' algorithm that uses the fast Artin-Schreier towers of Chapter 6; in this section we analyze the complexity of C2-AS

8.6.1 *Complexity analysis*

We borrow the complexity notations $L(i)$ (Theorem 6.23) and $PT(i)$ (Theorem 6.30) from Chapter 6.

p-torsion. The construction of $\mathbb{F}_q[c]$ may be done in many ways. The only requirements of Theorem 6.9

1. that its elements have a representation as elements of $\mathbb{F}_p[X]/Q_1(X)$ for some irreducible polynomial Q_1,
2. that either $(d, p) = 1$ or $\deg Q_1' + 2 = \deg Q_1$.

Selecting a random polynomial Q_1 and testing for irreducibility is usually enough to meet these conditions, as we saw in Remark 6.20. This costs

$$O\big(pdM(pd)\log(pd)\log(p^2d)\big)$$

according to [vzGG99, Th. 14.42].

Now we need to compute the embedding $\mathbb{F}_q \subset \mathbb{F}_q[c]$. Supposing $\mathbb{F}_q$ is represented as $\mathbb{F}_p[X]/Q_0(X)$, we factor Q_0 in $\mathbb{F}_q[c]$, which costs $O\big(pdM(pd^2)\log d\log p\big)$ using [vzGG99, Coro. 14.16]. Then the most naive technique to express the embedding is linear algebra. This requires the computation of pd elements of $\mathbb{F}_q[c]$ at the expense of $\Theta\big(pdM(pd)\big)$ operations in $\mathbb{F}_p$, then the inversion of the matrix holding such elements, at a cost of $\Theta\big((pd)^\omega\big)$ operations. This is certainly not optimal, yet this phase will have negligible cost compared to the rest of the algorithm.

Now we can compute c and c' by factoring the polynomials $Y^{p-1} - H_E$ and $Y^{p-1} - H_{E'}$ in $\mathbb{F}_p[X]/Q_1(X)$. This costs

$$O\big((pC(pd) + C(p)M(pd) + M(p)M(pd)\log p)(\log^2 p + \log d)\big)$$

using [KS97, Section 3].

Finally, computing the determinants needed by Gunji's formulas takes $\Theta(p^2)$ multiplications in $\mathbb{F}_q[c]$, that is $\Theta\big(p^2M(pd)\big)$.

Letting out logarithmic factors, the overall cost of this phase is

$$\tilde{O}\big(p^2d^3 + pC(pd) + C(p)pd + (pd)^\omega\big) \tag{8.40}$$

p^k-torsion. Application of Voloch formulas requires at each of the levels $\mathbb{U}_2, \dots, \mathbb{U}_k$

1. to solve equation (8.34) by factoring an Artin-Schreier polynomial,
2. to solve the system (8.35).

If we assume the worst case $[\mathbb{U}_2 : \mathbb{U}_1] = p$, according to Theorem 6.32, at each level i the first step costs

$$O\big((pd)^\omega i + PT(i-1) + M(p^{i+1}d)\log p\big)$$

while the second takes the GCD of two degree p polynomials in $\mathbb{U}_i[X]$ for each i (see Section 9.1), at a cost of $O\big(M(p^{i+1}d)\log p\big)$ operations using a fast Euclidean algorithm.

Summing up over i, the total cost of this phase up to logarithmic factors is

$$\tilde{O}_{p,d,\log \ell}\left((pd)^\omega \log_p^2 \ell + p^2 \ell d \log_p^4 \ell + \frac{\ell}{p} C(pd)\right). \tag{8.41}$$

Also notice that there is no need to store a $p^{k-1}d \times p^{k-1}d$ matrix to solve the Artin-Schreier equation, thus the space requirements are not anymore quadratic in ℓ.

Interpolation. The interpolation phase does not change in a significant way: one needs first to interpolate a degree $p^k - 1$ polynomial with coefficients in $\mathbb{U}_k$, then use Push-down to obtain the corresponding polynomial in $\mathbb{F}_q[X]$ and finally do a rational fraction reconstruction.

The first step costs $O\big(M(p^{2k}d)\log p^k\big)$ using fast techniques as in Section 2.2.5, then converting to $\mathbb{F}_q[c][X]$ takes $O\big(p^k L(k-1)\big)$ and further converting to $\mathbb{F}_q[X]$ takes $\Theta\big((pd)^2\big)$ by linear algebra. The rational function reconstruction then takes $O\big(M(p^k d)\log p^k\big)$.

The overall complexity of one interpolation is then

$$O\big(M(\ell^2 d)\log_p \ell + \ell L(k-1) + (pd)^2\big)\ . \tag{8.42}$$

Remember that this step has to be repeated an average number of $\varphi(p^k)/4$ times, thus the dependency of C2-AS in ℓ is still cubic.

8.7 The algorithm C2-AS-FI

The most expensive step of C2-AS is the polynomial interpolation step which is part of the Cauchy interpolation. If we use a standard interpolation algorithm, its input consists in a list of $\Theta(p^k)$ pairs $\big(P, \mathcal{I}(P)\big)$, with P having coordinates in $\mathbb{U}_k$, thus a lower bound for any such algorithm is $\Omega(p^{2k}d)$. Notice however that the output is a polynomial of degree $\Theta(p^k)$ in $\mathbb{F}_q[X]$, hence, if supplied with a shorter input, an *ad hoc* algorithm could reach the bound $\Omega(p^k d)$.

In this section we give an algorithm that reaches this bound up to some logarithmic factors. It realizes the polynomial interpolation on the primitive points of $E[p^k]$, thus its output is a degree $\varphi(p^k)/2-1$ polynomial in $\mathbb{F}_q[X]$. Using the Chinese remainder theorem it is straightforward to generalize this to an algorithm, having the same asymptotic complexity, that realizes the polynomial interpolation on all the points of $E[p^k]$. We call C2-AS-FI (FI for Fast Interpolation) the variant of C2-AS resulting from applying this new algorithm.

8.7.1 The algorithm

Let $P \in E[p^k]$ and $P' \in E'[p^k]$ be primitive p^k torsion points. We want to compute the polynomial $A \in \mathbb{F}_q[X]$ such that

$$A\big(x([n]P)\big) = x\big([n]P'\big) \quad \text{for any } n \in \big(\mathbb{Z}/p^k\mathbb{Z}\big)^*. \tag{8.43}$$

As we saw in Section 2.2.5, such a polynomial is only defined modulo the polynomial vanishing on the interpolation points

$$T(X) = \prod_{n\in(\mathbb{Z}/p^k\mathbb{Z})^*} \big(X - x([n]P)\big). \tag{8.44}$$

Thus we look for the canonical representative of A in $\mathbb{F}_q[X]/T(X)$.

We start by applying the Chinese remainder theorem to $\mathbb{F}_q[X]/T(X)$: let

$$T = \prod T^{(j)} \tag{8.45}$$

be the factorization of T over $\mathbb{U}_0$, and set

$$A^{(j)} \stackrel{\text{def}}{\equiv} A \bmod T^{(j)}\ . \tag{8.46}$$

Define

$$\mathbb{K}_j \stackrel{\text{def}}{\equiv} \mathbb{F}_q[X]/T^{(j)}(X), \tag{8.47}$$

then $\mathbb{K}_j$ is a field and $A^{(j)}$ is the projection of A in $\mathbb{K}_j$.

It was already pointed out in [Cou96, §2.3] that, knowing the factorization of T over $\mathbb{U}_0$ and all the $A^{(j)}$'s, we can recover A using the Chinese remainder algorithm of [vzGG99, §10] (see also Section 1.8). Thus we will focus on computing, say, $A^{(0)}$.

Choose any root ζ of $T^{(0)}$, without loss of generality we can take $\zeta = x(P)$, then

$$\mathbf{B} = \{1, \zeta, \dots, \zeta^{d-1}\} \tag{8.48}$$

is an $\mathbb{F}_q$-basis of $\mathbb{K}_0$. Fix the $\mathbb{F}_q$-linear embedding of finite fields

$$\mathbb{K}_0 \xhookrightarrow{\iota} \mathbb{U}_k \tag{8.49}$$

given by $\iota(\zeta) = x(P)$, by linearity it is evident that

$$\iota\big(A^{(0)}(\zeta)\big) = A^{(0)}\left(\iota(\zeta)\right) = x\big(P'\big). \tag{8.50}$$

Thus $\iota^{-1}\big(x(P')\big)$ is in $\mathbb{K}_0$ and the coefficients of A_0 are its coordinates in the basis $\mathbf{B}$. In conclusion, computing A_0 is equivalent to find a rational univariate representation of $x(P')$ with respect to $x(P)$.

Unfortunately, applying algorithm RUR is not optimal: the bottleneck is the power projection proj_ζ appearing in step 1. We have seen in Section 5.6 that the dual problem to power projection is polynomial evaluation, thus in particular

$$\begin{aligned} \mathrm{proj}_\zeta^* = \mathrm{ev}_\zeta : \mathbb{F}_q[X] &\to \mathbb{K}_j, \\ g &\mapsto g(\zeta); \end{aligned} \tag{8.51}$$

so that any algorithm to evaluate polynomials at ζ yields a power projection algorithm having the same complexity, and *vice-versa*. But none of the algorithms of Chapter 6 allows to evaluate polynomials in $\mathbb{F}_q[X]$ at a generic point of $\mathbb{U}_k$, better than a Horner rule.

We shall thus give an alternative algorithm to compute the minimal polynomial T_0 of $x(P)$. It will be similar to a subproduct tree, but it will exploit the structure of the Artin-Schreier tower. This is similar to the way we solved Artin-Schreier equations in Section 6.6.

Interpolation in towers of extensions. We set $\mathbb{U}_0 = \mathbb{F}_q$. The algorithm we give here can be applied in any tower of cyclic extensions, provided the action of the Galois groups can be computed. However we will present it only for our specific tower $(\mathbb{U}_0, \dots, \mathbb{U}_k)$, to avoid adding unnecessary notation.

Consider the following problem: given elements $x, y \in \mathbb{U}_k$ such that x generates $\mathbb{U}_k$ over $\mathbb{F}_q$, find a polynomial $A \in \mathbb{F}_q[X]$ such that

$$A(x) = y. \tag{8.52}$$

Let T be the minimal polynomial of x over $\mathbb{F}_q$, then, as above, the class of A in $\mathbb{F}_q[X]/T(X)$ is uniquely determined.

Let A be a polynomial satisfying (8.52) it is clear that $A(\sigma(x)) = \sigma(y)$ for any $\sigma \in \mathrm{Gal}(\mathbb{U}_k/\mathbb{F}_q)$. Conversely, the polynomial interpolating $\sigma(x)$ over $\sigma(y)$ for any σ is invariant under $\mathrm{Gal}(\mathbb{U}_k/\mathbb{F}_q)$, thus it has coefficients in $\mathbb{F}_q$. Hence we can construct A by interpolation.

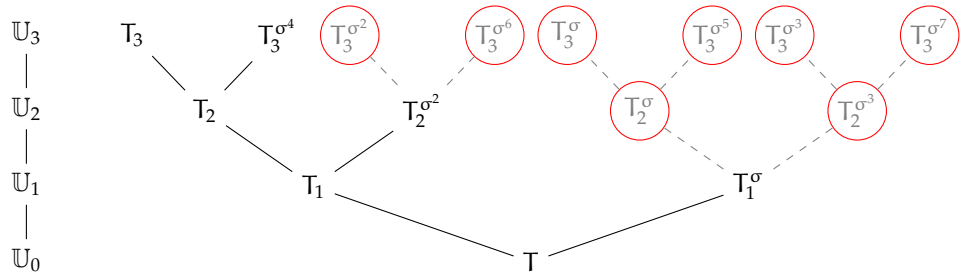


Figure 8.5: The subproduct tree of T, in the case of a tower of quadratic extensions. Any generator of $\mathrm{Gal}(\mathbb{U}_3/\mathbb{U}_0)$ can be taken as σ. We gray out and circle the nodes that the algorithm does not compute.

A fast interpolation algorithm would compute T via a binary subproduct tree, and then interpolate A recursively applying the Chinese remainder theorem along the branches of the tree. However this is too expensive. We can do better by using a non-binary subproduct tree on which the tower of Galois groups associated to $(\mathbb{U}_0, \dots, \mathbb{U}_k)$ acts.

First we need to compute T. Let T_i be the minimal polynomial of x over $\mathbb{U}_i$, we compute it recursively as

$$T_k = (X - x), \tag{8.53}$$

$$T_{i-1} = \prod_{\sigma \in \mathrm{Gal}(\mathbb{U}_i/\mathbb{U}_{i-1})} T_i^{\sigma}. \tag{8.54}$$

Then $T = T_0$. Observe that, rather than computing a whole subproduct tree of T, we have only computed one branching as shown in figure 8.5.

Now the we have something like a subproduct tree for T, we proceed as for interpolation. We compute recursively the polynomials in $A_i \in \mathbb{U}_i[X]$ such that $A_i(x) = y$. We start from $A_k = y$. Suppose A_{i+1} is known, then we use the Chinese remainder theorem to obtain the polynomial $P \in \mathbb{U}_{i+1}[X]/T_i(X)$ such that

$$P \equiv A_{i+1}^{\sigma} \bmod T_{i+1}^{\sigma} \qquad \text{for any } \sigma \in \mathrm{Gal}(\mathbb{U}_{i+1}/\mathbb{U}_i). \tag{8.55}$$

It is clear that P is invariant under $\mathrm{Gal}(\mathbb{U}_{i+1}/\mathbb{U}_i)$, hence $P \in \mathbb{U}_i[X]/T_i(X)$ and by (8.55) it is evident that $P(x) = A_{i+1}(x) = y$, thus $P = A_i$.

We have thus succeeded in interpolating $A = A_0$, without having to build the whole subproduct tree. A similar algorithm was applied by Enge and Morain to the solution of equations by radicals [EM03], although they did not recognize the application to polynomial interpolation.

Remark 8.13. Observe that, once the polynomials T_i for $0 \leqslant i \leqslant k$ are known, we have an efficient algorithm to evaluate polynomials in $g \in \mathbb{U}_0[X]$ at the point x: simply compute

$$\begin{aligned} g_0 &= g \bmod T_0, \\ g_1 &= g_0 \bmod T_1, \\ &\vdots \\ g_k &= g_{k-1} \bmod T_k, \end{aligned} \tag{8.56}$$

then $g_k = g(x)$. Transposing this algorithm gives a power projection algorithm that can be used in RUR. By the discussion in Section 2.2.8, the transpose of this algorithm amounts to iteratively extend a linearly recurring sequence. However, we do not use this method because it would not improve the overall complexity, as we shall show in the next section.

Back to our problem. It is easy to realize that, on inputs $x(P)$ and $x(P')$, the algorithm we just gave computes $A^{(0)}$. In fact, $T^{(0)}$ is the minimal polynomial of $x(P)$ over $\mathbb{F}_q$ and $A^{(0)}$ is the unique polynomial in $\mathbb{F}_q[X]/T^{(0)}(X)$ that satisfies (8.52).

This can be viewed as decomposing the morphism ι of Eq. (8.49) as the chain of $\mathbb{F}_q$-linear isomorphisms

$$\mathbb{U}_0[X_0]/_{T_0(X_0)} \xhookrightarrow{\iota_0} \cdots \xhookrightarrow{\iota_{k-1}} \mathbb{U}_k[X_k]/_{T_k(X_k)} \xhookrightarrow{\iota_k} \mathbb{U}_k \tag{8.57}$$

defined by $\iota_k \circ \cdots \circ \iota_i(X_i) = x(P)$ for any i, and then finding the preimage of $x(P')$ by inverting them one by one.

Then, the Chinese remainder theorem we applied in (8.55) amounts to invert ι_i by descending the lower path in the diagram below

$$\begin{array}{ccc} \mathbb{U}_i[X_i]/_{T_i(X_i)} & \xhookrightarrow{\iota_i} & \mathbb{U}_{i+1}[X_{i+1}]/_{T_{i+1}(X_{i+1})} \\ \Big\downarrow \varepsilon & & \Big\uparrow \pi \\ \mathbb{U}_{i+1}[Y]/_{T_i(Y)} & \xhookrightarrow{\gamma} & \bigoplus_\sigma \mathbb{U}_{i+1}[Y_j]/_{(T_{i+1})^\sigma(Y_j)} \end{array} \tag{8.58}$$

where ε is the canonical injection extending $\mathbb{U}_i \subset \mathbb{U}_{i+1}$, γ is the Chinese remainder isomorphism and π is projection onto the first coordinate.

Some care must be taken when $x(P)$ does not generate $\mathbb{U}_k$, but only a subfield of index 2. This happens when $c \notin \mathbb{F}_q[c^2]$, and in this case ι_0 is not a field isomorphism. It is not to difficult, however, to handle this case, as one only needs to take a subgroup of index 2 of $\mathrm{Gal}(\mathbb{U}_1/\mathbb{U}_0)$, instead of the whole group, in the interpolation algorithm given above.

8.7.2 Complexity analysis

In practice, the algorithms to compute $T^{(0)}$ and $A^{(0)}$ are modified versions of the subproduct tree and the interpolation (see Section 2.2.5).

We set some notation. Let i_0 be the largest index such that $\mathbb{U}_{i_0} = \mathbb{U}_1$ and let $\frac{p-1}{2r} = [\mathbb{F}_q[c^2] : \mathbb{F}_q]$. Note that all the $T^{(j)}$'s have degree $\frac{\varphi(p^{k-i_0+1})}{2r}$.

We first compute the *truncated* subproduct tree as in Figure 8.5. The product of Eq. (8.54) is computed via a classic binary subproduct tree to keep the complexity low.

Recall that we denote by $L(i)$ the cost of performing one lift-up or push-down at the i-th level (see Theorem 6.23). For any $i > i_0$, step 4 of is repeated p times, each iteration taking

$$O\big(p^{k-i} L(i-i_0)\big) \subset O\big(L(k-i_0)\big)$$

by Theorem 6.27. Step 5 takes $O\big(M(p^{k-i_0+1}d/r)\log p\big)$ using Algorithm 2.3 and step 6 takes

$$O\big(p^{k-i+1} L(i-i_0)\big) \subset O\big(pL(k-i_0)\big).$$

Algorithm 8.4: Truncated subproduct tree

Input : $x(P) \in \mathbb{U}_k$.
Output : The subproduct tree.
1: Let $T_k = (X - x(P))$;
2: for $i = k-1$ to 0 do
3: for all $\sigma \in \mathrm{Gal}(\mathbb{U}_{i+1}/\mathbb{U}_i)$ do
4: compute T_{i+1}^{σ} using IterFrobenius;
5: $T_i \leftarrow \prod_{\sigma} T_{i+1}^{\sigma}$ via a binary subproduct tree.
6: convert T_i into an element of $\mathbb{U}_i[X]$ using Push-down.

Algorithm 8.5: Truncated fast interpolation

Input : $T_i \in \mathbb{U}_i[X]$ for $0 \leqslant i \leqslant k$, $x(P') \in \mathbb{U}_k$, $T_0'(x(P)) \in \mathbb{U}_k$.
Output : $A_0 \in \mathbb{U}_0[X]$.
1: $P_k \leftarrow x(P')/T_0'(x(P))$;
2: for $i = k-1$ to 0 do
3: for all $\sigma \in \mathrm{Gal}(\mathbb{U}_{i+1}/\mathbb{U}_i)$ do
4: compute P_{i+1}^{σ} using IterFrobenius;
5: convert T_i into an element of $\mathbb{U}_{i+1}[X]$ using Lift-up.
6: $P_i \leftarrow \sum_{\sigma} P_{i+1}^{\sigma} T_i / T_{i+1}^{\sigma}$ using the binary subproduct tree computed previously;
7: convert P_i into an element of $\mathbb{U}_i[X]$ using Push-down.
8: return P_0.

For any $1 \leqslant i < i_0$, there is nothing to do because $\mathbb{U}_{i+1} = \mathbb{U}_i$. Finally, when $i = 0$ and $\mathbb{U}_1 \neq \mathbb{F}_q$ the algorithm is identical but step 4 must be computed through a generic Frobenius algorithm (using the algorithm of Section 2.2.4, for example) and step 6 must use the implementation of $\mathbb{F}_q[c]$ to make the conversion (for example, linear algebra). In this case step 4 costs $\Theta\big(\frac{p^{k-i_0}}{r} C(pd) \log d\big)$ by Eq. (2.19) and step 6 costs $\Theta\big(p^{k-i_0}(pd)^2\big)$.

Now, we have T_0 at the root of the tree. We compute its derivative T_0' and we evaluate it at $x(P)$ by reducing modulo $T_1, \ldots, T_k$. This costs strictly less than computing the subproduct tree. We finally do the interpolation of A_0.

Step 1 is just one inversion in $\mathbb{U}_k$, that is $O(M(p^{k-i_0}d/r) \log p^{k-i_0} d/r)$. Steps 4 and 7 are identical to steps 4 and 6 of the subproduct tree and step 5 is also absorbed. Finally, step 6 has the same complexity as step 5 of the subproduct tree, using the Algorithm 2.5.

In conclusion, the total cost of computing the subproduct tree and the interpolation is

$$O\left((k-i_0)pL(k-i_0) + M\left(\frac{p^{k-i_0+1}d}{r}\right) \log \frac{p^{k-i_0}d}{r} + \frac{p^{k-i_0}}{r}\big(C(pd)\log d + r(pd)^2\big)\right).$$

The complete interpolation. We compute all the $A^{(j)}$'s using this algorithm; there are $p^{i_0-1}r$ of them. We then recombine them through a Chinese remainder algorithm at a cost of $O\big(M(p^k d) \log p^k d\big)$. The total cost of the whole interpolation phase is

then

$$O\left((k-i_0)pL(k) + p^{k-1}C(pd)\log d + p^{k-1}r(pd)^2 + M(p^k d)\log p^k d\right),$$

that is

$$O\left(pL(k)\log\left(\frac{\ell}{p^{i_0}}\right) + M(\ell d)\log \ell d + \frac{\ell}{p}C(pd)\log d + \ell(pd)^2\right). \tag{8.59}$$

Alternatively, once $A^{(0)}$ is known, one could compute the other $A^{(j)}$'s using modular composition with the multiplication maps of E and E' as suggested in [Cou96]. However this approach does not give a better asymptotic complexity because in the worst case $A^{(0)} = A$. From a practical point of view, though, Brent's and Kung's algorithm for modular composition [BK78], despite having a worse asymptotic complexity, could perform faster for some set of parameters. We will discuss this matter in Section 8.8.

If more than $\varphi(p^k)/2$ points are needed, but less than $\frac{p-1}{2}$, one can use the previous algorithm to interpolate over the primitive p^i-torsion points for each $i = 1, \ldots, k$. The interpolating polynomials can then be recombined through a Chinese remainder algorithm at a cost of $O\big(M(p^k d)\log p^k\big)$, which does not change the overall complexity of C2-AS-FI.

Putting together the complexity estimates of C2-AS and C2-AS-FI, we have the following theorem.

THEOREM 8.14 *Assuming* $M(n) = n\log n\,\mathrm{loglog}\, n$, *the algorithm* C2-AS-FI *has worst case complexity*

$$\tilde{O}_{p,d,\log \ell}\left(p^2d^3 + C(p)pd + (pd)^\omega \log^2 \ell + p^3\ell^2 d\log^3 \ell + p^2\ell^2 d^2 + \left(\frac{\ell^2}{p} + p\right)C(pd)\right).$$

8.8 THE ALGORITHM C2-AS-FI-MC

However asymptotically fast, the polynomial interpolation step is quite expensive for reasonably sized data. Instead of repeating it $\frac{\varphi(p^k)}{2}$ times, one can use composition with the Frobenius endomorphism ϕ_E in order to reduce the number of interpolations in the final loop. We call this variant C2-AS-FI-MC (MC for Modular Composition).

8.8.1 The algorithm

Suppose we have computed, by the algorithm of the previous Section, the polynomial T vanishing on the abscissas of $E[p^k]$ and an interpolating polynomial $A_0 \in \mathbb{F}_q[X]$ such that

$$A_0\big(x([n]P)\big) = x\big([n]P'\big) \quad \text{for any } n.$$

We view A_0 as a morphism of varieties (not necessarily an isogeny!) from $E[p^k]$ to $E'[p^k]$.

The Frobenius automorphism ϕ_E acts on $E[p^k]$ permuting its points. We define

$$A_1 \stackrel{\text{def}}{=} A_0 \circ \phi_E = \phi_{E'} \circ A_0, \tag{8.60}$$

where the equality comes from the fact that A_0 is $\mathbb{F}_q$-rational. Hence

$$A_1 \circ [n](P) = [n] \circ \phi_{E'}(P') \quad \text{for any } n.$$

A_1 has no poles at $E[p^k]$, thus it is a polynomial map; and since $\phi_{E'}(P')$ is a generator of $E'[p^k]$, A_1 is one of the polynomials that the algorithm C2 tries to identify to an isogeny. By iterating this construction we obtain $[\mathbb{U}_k : \mathbb{F}_q]/2$ different polynomials A_i for the algorithm C2 with only one interpolation.

To compute the A_i's, we first compute $F \in \mathbb{F}_q[X]$

$$F(X) = X^q \bmod T(X), \tag{8.61}$$

then for any $1 \leqslant i < [\mathbb{U}_k : \mathbb{F}_q]/2$

$$A_i(X) = A_{i-1}(X) \circ F(X) \bmod T(X). \tag{8.62}$$

If $\frac{\varphi(p^k)}{[\mathbb{U}_k:\mathbb{F}_q]} = p^{i_0-1}r$, we must compute $p^{i_0-1}r$ polynomial interpolations and apply this algorithm to each of them in order to deduce all the polynomials needed by C2.

8.8.2 Complexity analysis

We compute (8.61) via square-and-multiply, this costs $\Theta(dM(p^k d)\log p)$ operations. Each application of (8.62) is done via a modular composition, the cost is thus $O(C(p^k))$ operations in $\mathbb{F}_q$, that is $O(C(p^k)M(d))$ operations in $\mathbb{F}_p$. Using the algorithm of [KU08] for modular composition, the complexity of C2-AS-FI-MC would not be essentially different from the one of C2-AS-FI. However, in practice the fastest algorithm for modular composition is [BK78], and in particular the variant in [KS98, Lemma 3]: these have worse asymptotic complexity, but perform better on the instances we treat in Section 9.4.

Notice that a similar approach could be used inside the polynomial interpolation step (see Section 8.7) to deduce $A_k^{(0)}$ from $A_0^{(0)}$ using modular composition with the multiplication maps of E and E' as described in [Cou96, §2.3]. This variant, though, has an even worse complexity because of the cost of computing multiplication maps.

8.9 Isogenies of unknown degree

We now present an extension to Couveignes algorithm that could be useful in cryptographic application. Recall that two curves having the same number of points over a finite field are isogenous; however this says nothing on the degree of the isogeny connecting them. Given two elliptic curves E and E' defined over $\mathbb{F}_q$ and having the same number of points, we want to find an $\mathbb{F}_q$-rational isogeny between them.

The simplest solution is to take any algorithm computing an isogeny of given degree, and try all the degrees until an isogeny is found. If ℓ is the degree of the smallest isogeny, this of course adds a factor ℓ to the complexity of any polynomial time algorithm.

Couveignes' algorithm can be easily adapted to solve this problem at no additional cost. We call this algorithm C2-UD (UD for Unknown Degree), all the variants of C2 presented until now also apply to C2-UD.

Observe that, apart for the choice of k, the computation of $E[p^k]$ and the polynomial interpolation step do not depend at all on ℓ. The degree of the isogeny only comes into play in the last part of the Cauchy interpolation, that is in the rational function reconstruction. We study more in detail this last step.

Rational Fraction Reconstruction. Recall from Section 2.2.6 that rational fraction reconstruction takes as input a degree n polynomial T, a polynomial A of degree less than n and a target degree $m \leqslant n$ and outputs the unique rational fraction such that

$$A \equiv \frac{R}{V} \bmod T$$

and $\deg R < m$, $\deg V \leqslant n - m$. This is done by computing a Bézout relation $AV + TU = R$ with the expected degrees via an XGCD algorithm. If a classical XGCD algorithm is used, one simply computes all the lines

$$\begin{array}{lll} R_0 = T, & U_0 = 1, & V_0 = 0, \\ R_1 = A, & U_1 = 0, & V_1 = 1, \\ R_{i-1} = Q_i R_i + R_{i+1}, & U_{i+1} = U_{i-1} - Q_i U_i, & V_{i+1} = V_{i-1} - Q_i V_i \end{array} \tag{8.63}$$

and stops as soon as a remainder R_{i+1} with $\deg R_{i+1} < m$ is found. If a fast XGCD algorithm such as [vzGG99, §11.1] is used, one directly aims at the two lines

$$\begin{aligned} R_{h-2} &= Q_{h-1} R_{j-1} + R_h \\ R_{h-1} &= Q_h R_h + R_{h+1} \end{aligned} \tag{8.64}$$

such that $\deg R_{h+1} < m \leqslant \deg R_h$ without computing the intermediate lines.

In Couveignes' algorithm, when looking for an ℓ-isogeny, one simply sets $m = \ell + 1$. Observe that if the algorithm does not return a rational fraction $\frac{R}{V}$ with $\deg R = \ell$ and $\deg V = \ell - 1$, then no such fraction congruent to A modulo T exists.

If ℓ is not *a priori* known, we can still use the fact that a separable isogeny with cyclic kernel must have $\deg R = \deg V + 1$. In fact, if we suppose $R = R_i$ and $V = V_i$, then

$$\begin{aligned} \deg T &= \deg V_{i+1} + \deg R_i, \\ \deg R_i - \deg V_i &= \deg R_{i-1} - \deg V_{i+1} \end{aligned}$$

implies

$$\deg T + 1 = \deg R_{i-1} + \deg R_i \ .$$

Hence, if A is congruent to an ℓ-isogeny with $\ell = \left\lfloor \frac{\deg T}{2} \right\rfloor - t$ for some $t \geqslant 0$, then

$$\deg R_{i-1} = \left\lceil \frac{\deg T}{2} \right\rceil + t + 1 > \left\lfloor \frac{\deg T}{2} \right\rfloor - t = \deg R_i \ . \tag{8.65}$$

Thus we can recover any isogeny having degree less than $\left\lfloor \frac{\deg T}{2} \right\rfloor$ using either a classical or a fast XGCD algorithm, setting $m = \left\lceil \frac{\deg T}{2} \right\rceil + 1$.

Recognizing an isogeny. Once we have a rational fraction with the required degree, we have to test if it really is an isogeny. In order to understand how often we have to make this test, we introduce some more terminology. Let $n_i = \deg R_i$, we call $(n_0, \dots, n_r)$ the *degree sequence* of A and T; a degree sequence is said *normal* if $n_i = n_{i+1} + 1$ for any i.

Proposition 8.15 *Let $f, g \in \mathbb{F}_q[X]$ be uniformly chosen random polynomials of respective degrees $n_0 > n_1 > 0$ and let $(n_0, n_1, \dots, n_r)$ be their degree sequence. For $0 \leqslant i < n_1$ define the binary random variables $X_i = 1 \Leftrightarrow i \in (n_0, n_1, \dots, n_r)$, then the X_i are independent random variables and $\mathrm{Prob}(X_i = 0) = \frac{1}{q}$.*

Proof. Pairs of polynomials f, g are in bijection with the GCD-sequence $(R_r, Q_r, \dots, Q_1)$ constituted by their GCD and the quotients of the GCD algorithm. To each such sequence is associated a degree sequence

$$(n_0, n_1, \dots, n_r) = \left(\deg R_r + \sum_{i=1}^{r} \deg Q_i, \dots, \deg R_r + \sum_{i=1}^{1} \deg Q_i, \deg R_r \right),$$

thus for any given degree sequence there are

$$(q-1)q^{n_0-n_1} \cdot (q-1)q^{n_1-n_2} \cdot \dots \cdot (q-1)q^{n_r} = (q-1)^{r+1} q^{n_0}$$

GCD-sequences.

Let I and O be two disjoints subsets of $\{X_i\}$, the number of GCD-sequences such that $X \in I \Rightarrow X = 1$ and $X \in O \Rightarrow X = 0$,

$$\sum_{s=0}^{n_1 - \#I - \#O} \binom{n_1 - \#I - \#O}{s} (q-1)^{s+2+\#I} q^{n_0} = (q-1)^{2+\#I} q^{n_0} q^{n_1 - \#I - \#O} .$$

There are $(q-1)^2 q^{n_0} q^{n_1}$ pairs of polynomials of degrees n_0, n_1, thus

$$\mathrm{Prob}\big(\{X = 1 \mid X \in I\}, \{X = 0 \mid X \in O\}\big) = \left(\frac{q-1}{q}\right)^{\#I} \left(\frac{1}{q}\right)^{\#O} . \tag{8.66}$$

The claim follows. 

Degree sequences associated to isogenies are in general not normal, in fact if $\ell \leqslant \left\lfloor \frac{\deg T}{2} \right\rfloor - t$, equation (8.65) shows that there must be at least a gap of degree $2t$ in the degree sequence. Heuristically, we can expect that if the polynomial A does not correspond to an isogeny, then A and T act like random polynomials, thus, by the proposition above, the probability that A looks like an isogeny of degree $\ell \leqslant \left\lfloor \frac{\deg T}{2} \right\rfloor - t$ is less than $\frac{1}{q^{2t}}$.

Therefore, by choosing an appropriate $t \in O(\log_q p^k)$, C2-UD can find any isogeny of degree less than $\frac{p^k - 1}{4} - t$ at the same cost of one run of C2. Also notice that C2-UD is not restricted to isogenies of degree prime to p as was already mentioned in Section 8.5.4.

Note. This variant makes Couveignes' algorithm quite unique for various reasons. First of all, it is the first algorithm, other than trivial ones, to compute isogenies of unknown degree. Besides, most algorithms to compute isogenies of fixed

degree do not seem to have a similar variant: for example, BMSS requires ℓ-normalized models, and Lercier-Sirvent uses Φ_ℓ. We think that Couveignes' first algorithm [Cou94] could also be generalized to compute isogenies of unknown degree, thanks to its similarity to C2.

It is also interesting to notice that, while for computing isogenies of degree ℓ there is still a complexity gap between the large and small characteristic cases, C2-UD closes this gap in the unknown degree case. Finally, this variant explains somehow why Couveignes' algorithm is not optimal: because it solves another problem.

9 Chapter



Experimental results

In this chapter we describe the implementations we made of the algorithms of the previous chapter and some experimental results.

9.1 Implementation of Couveignes' algorithm

We implemented C2-AS-FI-MC as `C++` programs using the libraries `NTL` [Sho03] for finite field arithmetic, `gf2x` [BGTZ08] for fast arithmetic in characteristic 2 and `FAAST` (see Section 6.7) for fast arithmetic in Artin-Schreier towers. We also have a Magma [BCP97] prototype of the same algorithm, not making use of the fast algorithms of Chapter 6. This section mainly deals with some tricks we implemented in order to speed up the computation.

9.1.1 Building $E[p^k]$ and $E'[p^k]$

p-Torsion. For $p \neq 2$, C2 and its variants require to build the extension $\mathbb{F}_q[c]$ where c is a $(p-1)$-th root of H_E. In order to deal with the lowest possible extension degree, it is a good idea to modify the curve so that $[\mathbb{F}_q[c] : \mathbb{F}_q]$ is the smallest possible.

$[\mathbb{F}_q[c] : \mathbb{F}_q]$ is invariant under isomorphism, but taking a twist can save us a quadratic extension. Let $u = c^{-2}$, the curve

$$\bar{E} : y^2 = x^3 + a_2 u x^2 + a_4 u^2 x + a_6 u^3$$

is defined over $\mathbb{F}_q[c^2]$ and is isomorphic to E over $\mathbb{F}_q[c]$ via $(x, y) \mapsto (\sqrt{u}^2 x, \sqrt{u}^3 y)$. Its Hasse invariant is $H_{\bar{E}} = (u)^{\frac{p-1}{2}} H_E = 1$, thus its p-torsion points are defined over $\mathbb{F}_q[c^2]$.

In order to compute the p^k-torsion points of E we build $\mathbb{F}_q[c^2]$, we compute $\bar{P}$ a p^k-torsion points of $\bar{E}$ using p-descent, then we invert the isomorphism to compute the abscissa of $P \in E[p^k]$. Since the Cauchy interpolation only needs the abscissas of $E[p^k]$, this is enough to complete the algorithm. Scalar multiples of P can be computed without knowledge of $y(P)$ using Montgomery formulas.

Note that for $p = 2$ we use the same construction in an implicit way since we do a p-descent on the Kummer variety.

p^k-Torsion points. For $p \neq 2$ we use Voloch's p-descent to compute the p^k-torsion points iteratively as described in Section 8.5. To factor the Artin-Schreier polynomial (8.34), we use the algorithms from Section 6.6 implemented in `FAAST`.

To solve system (8.35) we first compute

$$V(x,y) = \left(\frac{g(x)}{h^2(x)}, sy\left(\frac{g(x)}{h^2(x)}\right)'\right)$$

through Vélu formulas.[1] Recall that we work on a curve having Hasse invariant 1, system (8.35) can then be rewritten

$$\begin{cases} \tilde{x}(P) = \dfrac{g(x)}{h^2(x)} \\ \tilde{y}(P) = sy\left(\dfrac{g(x)}{h^2(x)}\right)' \\ \tilde{z}(P) = -2y\dfrac{h'(x)}{h(x)} \end{cases}$$

where P is the point on the cover C that we want to pull back ($\tilde{x}(P)$, $\tilde{y}(P)$ and $\tilde{z}(P)$ are just its coordinates). After some substitutions this is equivalent to

$$\begin{cases} \tilde{x}(P)h^2(x) - g(x) = 0 \\ \left(\tilde{x}(P)h^2(x) - g(x) - \dfrac{\tilde{y}(P)}{s\tilde{x}(P)}h^2(x)\right)' = 0 \end{cases}$$

Then a solution in x to this system is given by the GCD of the two equations. Note that proposition 8.10 ensures there is one unique solution. These formulas are slightly more efficient than the ones in [Ler97, §6.2].

For $p = 2$ we use the library `FAAST` (for solving Artin-Schreier equations) on top of `gf2x` (for better performance). There is nothing special to remark about the 2-descent.

9.1.2 Cauchy interpolation and loop

The polynomial interpolation step is done as described in Section 8.7. As a result of this implementation, the polynomial interpolation algorithm was added to the library `FAAST`.

The rational fraction reconstruction is implemented using a fast XGCD algorithm on top of `NTL` and `gf2x`. This algorithm was added to `FAAST` too.

The loop uses modular composition as in Section 8.8 in order to minimise the number of interpolations. The timings in the next section clearly show that this non-asymptotically-optimal variant performs much faster in practice.

To check that the rational fractions are isogenies we test their degrees, that their denominator is a square and that they act as group morphisms on a fixed number of random points. All these checks take a negligible amount of time compared to the rest of the algorithm.

9.1.3 Parallelisation of the loop

The most expensive step of C2-AS-FI-MC, in theory as well as in practice, is the final loop over the points of $E'[p^k]$. Fortunately, this phase is very easy to parallelise with very little overhead.

[1] Vélu formulas compute this isogeny up to an indeterminacy on the sign of the ordinate, the actual value of s must be determined by composing V with ϕ and verifying that it corresponds to $[p]$ by trying some random points.

Let n be the number of processors we wish to parallelise on, suppose that $[\mathbb{U}_k : \mathbb{F}_q]$ is maximal, then we make only one interpolation followed by $\varphi(p^k)/2$ modular compositions.[2] We set $m = \left\lfloor \frac{\varphi(p^k)}{2n} \right\rfloor$ and we compute the action of ϕ^m on $E[p^k]$ as in Section 8.8:

$$F^{(m)}(X) = F(X) \circ \cdots \circ F(X) \bmod T(X) ,$$

this can be done with $\Theta(\log m)$ modular compositions via a binary square-and-multiply approach as in Section 2.2.4.

Then we compute the n polynomials

$$A_{mi}(X) = A_{m(i-1)}(X) \circ F^{(m)}(X) \bmod T(X)$$

and distribute them to the n processors so that they each work on a separate slice of the A_i's. The only overhead is $\Theta(\log(\ell/n))$ modular compositions with coefficients in $\mathbb{F}_q$, this is acceptable in most cases.

9.2 Implementation of C2-UD

We modified our C++ implementation of C2-AS-FI-MC to obtain two variants of C2-UD.

The first one takes an integer k and looks for all isogenies of degree $p^c\ell$ with ℓ prime to p, $\ell < \varphi(p^k)/4$ and c arbitrary. This is done by slightly modifying the modular composition step of Section 8.8. Suppose we know an interpolating polynomial A_0, that we view as a morphisms $E[p^k] \to E'[p^k]$ such that

$$A_0 \circ [n](P) = [n](P') \quad \text{for any } n. \tag{9.1}$$

Then we compose with the Frobenius isogeny ϕ

$$A_1 \stackrel{\text{def}}{\equiv} \phi \circ A_0 : E[p^k] \to E'^{(p)}[p^k], \tag{9.2}$$

where $E'^{(p)}$ is the curve

$$E^{(p)} \ : \ y^2 = x^3 + a'^p x + b'^p. \tag{9.3}$$

So A_1 is one of the polynomials that Couveignes algorithm computes when looking for an isogeny between E and $E'^{(p)}$. If $q = p^d$, iterating d times this construction, we fall back on E', as we would have if we had directly applied the Frobenius automorphism as in Section 8.8. Thus, paying an additional factor of $\log_p q$, we can compute any isogeny of degree $p^c\ell$ with arbitrary c and ℓ bounded as before.

When $\log_p q$ is large, the previous variant becomes unpractical. We implemented a second variant using the algorithm described in Section 8.5.4, this allows to compute any isogeny of degree $\ell \leqslant \varphi(p^k)/4$, even if p divides ℓ. The asymptotic cost of this variant is the same as one run of C2-AS-FI-MC, because the search for ℓ prime to p dominates.

[2] If $[\mathbb{U}_k : \mathbb{F}_q]$ is not maximal, the parallelisation is straightforward: we simply send one interpolation to each processor in turn.

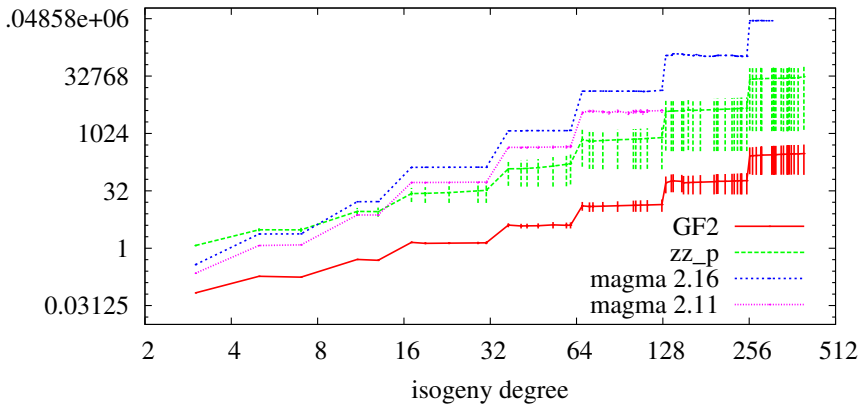


FIGURE 9.1: Comparative timings for different implementations of C2-AS-FI-MC with curves defined over $\mathbb{F}_{2^{101}}$. Plot in logarithmic scale.

9.3 Implementation of Lercier-Sirvent

We implemented a Magma prototype of BMSS and Lercier-Sirvent. In both cases we did not take Remark 8.4 into account, and only implemented the variant using rational fraction reconstruction. We used Magma native support for p-adics to construct the field $\mathbb{Q}_q$.

Instead of the classical modular polynomials Φ_ℓ we used Atkin's modular polynomials Φ_ℓ^* since they have smaller coefficients and degree; this does not change the other steps of the algorithm.

The modular polynomials were not computed on the fly as suggested in Section 8.4, instead they were taken from the tables precomputed in Magma, however this is not expected to significantly affect the running time for the sizes we treat in the next section (compare with [BLS10, Table 3]).

9.4 Benchmarks

We ran various experiments to compare the different variants of the algorithm C2 between themselves and to Lercier-Sirvent. All the experiments were run on four dual-core Intel Xeon E5520 (2.26GHz), using the parallelized version of the algorithm in some cases. Magma 2.16 was used to run Magma experiments. We also report in figure 9.1 some older timings obtained on an Intel Xeon E5430 with Magma 2.11, that show a regression between the two versions of Magma.

Magma vs. `FAAST`. The first set of experiments was run to evaluate the benefits of using the algorithms of Chapter 6. We selected pairs of isogenous curves over $\mathbb{F}_{2^{101}}$ such that the height of the tower is maximal (observe that this is always the case for cryptographic curves). We compared the Magma prototype to the `FAAST`-based implementation of C2-AS-FI-MC using the `zz_p` and `GF2` data types (see Section 6.7).

The results are in figure 9.1: we plot a line for the average running time of the algorithm and bars around it for minimum and maximum execution times of the final loop. Besides the dramatic speedup obtained by using the ad-hoc type `GF2`,

ℓ	$E[p^k]$	$E'[p^k]$	FI	RFR	MC	Avg tries	loop time
31	0.3529	0.3529	0.3569	0.00125	0.00055	32	0.058
61	0.9848	0.9848	0.8268	0.00343	0.00228	64	0.365
127	2.6636	2.6626	1.8927	0.01090	0.00872	128	2.511
251	6.9809	6.9779	4.2833	0.03092	0.03494	256	16.860
397	18.1052	18.0952	9.7385	0.07325	0.14117	512	109.783

Table 9.1: Comparative timings in seconds for the phases of C2-AS-FI-MC for curves over $\mathbb{F}_{2^{101}}$.

the algorithmic improvements of FAAST over Magma are evident as even zz_p is one order of magnitude faster.

Table 9.1 shows detailed timings for each phase of C2-AS-FI-MC. The column FI reports the time for one interpolation, the column MC the time for one modular composition; comparing these two columns the gain from passing from C2-AS-FI to C2-AS-FI-MC is evident. Columns RFR (rational fraction reconstruction) and MC constitute the Cauchy interpolation step that is repeated in the final loop. The last column reports the average time spent in the loop: it is by far the most expensive phase and this justifies the attention we paid to FI and MC; only on some huge examples we approached the crosspoint between these two algorithms.

C2-UD. Next we ran experiments on C2-UD. The first observation was that the heuristic argument –on the probability that a degree sequence not associated to an isogeny is not normal– is well verified in practice: except for a degree 2 symmetry verified in characteristic 2, polynomials not associated to an isogeny very rarely gave a degree sequence with a gap around the middle.

Looking for isogenies of unknown degree may be of some cryptographic significance. For example, Teske's trapdoor cryptosystem selects a binary field of composite degree ($\mathbb{F}_{2^{7\cdot 23}}$, in the proposal) and chooses an elliptic curve E vulnerable to the GHS attack [GHS02b]. Then hides E by taking a random path of isogenies of small degrees landing on a curve E' not vulnerable to GHS, and uses E' as public key. The security of the cryptosystem comes from the assumption that it is infeasible to find a GHS-vulnerable curve isogenous to E', without the knowledge of the isogeny path.

The *trapdoor* of the cryptosystem is the curve E: it is given to a trusted authority so that –using an isogeny path from E' to E and a GHS attack– it has the power of deciphering messages at a relatively high computational cost. This feature rests on the assumption that it is feasible, but relatively hard, to compute any isogeny path from E to E'.

In this context, it may be interesting to verify that E and E' are not related by an isogeny of too low degree. From [Tes06, Appendix A], we took the two curves defined over $F_{2^{161}} = \mathbb{F}_2[z]/(z^{161} + z^{18} + 1)$ of j invariants:

$$1/j = z^{152}+z^{143}+z^{139}+z^{136}+z^{135}+z^{133}+z^{130}+z^{125}+z^{124}+z^{122}+z^{120}+z^{119}+z^{118}+z^{117}+z^{116}+z^{114}+z^{113}+z^{112}+z^{110}+z^{109}+z^{106}+z^{105}+z^{103}+z^{102}+z^{101}+z^{99}+z^{97}+z^{96}+z^{92}+z^{91}+z^{88}+z^{87}+z^{86}+z^{85}+z^{81}+z^{78}+z^{77}+z^{76}+z^{75}+z^{73}+z^{71}+z^{69}+z^{68}+z^{67}+z^{66}+z^{63}+z^{59}+z^{58}+z^{53}+z^{51}+z^{50}+z^{49}+z^{48}+z^{46}+z^{45}+z^{44}+z^{42}+z^{38}+z^{34}+z^{3}+z^{32}+z^{31}+z^{29}+z^{27}+z^{26}+z^{24}+z^{23}+z^{22}+z^{21}+z^{20}+z^{19}+z^{18}+z^{17}+z^{16}+z^{15}+z^{14}+z^{13}+z^{12}+z^{10}+z^{7}+z^{6}+z^{4}+z^{3}+z^{2},$$

$$1/j' = z^{160}+z^{156}+z^{155}+z^{153}+z^{152}+z^{151}+z^{150}+z^{149}+z^{148}+z^{147}+z^{146}+z^{145}+z^{143}+z^{142}+z^{141}+z^{130}+z^{129}+z^{127}+z^{126}+z^{125}+z^{124}+z^{123}+z^{120}+z^{118}+z^{112}+z^{109}+z^{104}+z^{103}+$$

ℓ	Lift	DiffSolve	RFR
31	0.570	14.830	0.010
103	5.160	274.550	0.250
149	12.510	815.320	0.590
239	21.420	1470.240	1.950
331	113.500	4204.610	4.890
389	147.340	5166.730	7.360

TABLE 9.2: Comparative timings in seconds for the phases of Lercier-Sirvent for curves over $\mathbb{F}_{3^{64}}$.

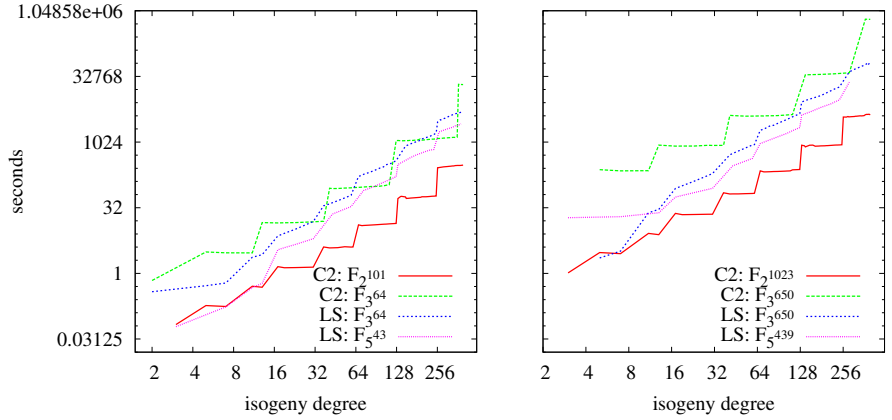


FIGURE 9.2: Comparative timings for C2-AS-FI-MC (C2) and Lercier-Sirvent (LS) over different curves. Plot in logarithmic scale.

$z^{102}+z^{101}+z^{99}+z^{98}+z^{97}+z^{96}+z^{93}+z^{92}+z^{91}+z^{90}+z^{88}+z^{85}+z^{83}+z^{77}+z^{74}+z^{70}+z^{68}+z^{65}+z^{64}+z^{63}+z^{62}+z^{61}+z^{60}+z^{58}+z^{57}+z^{55}+z^{50}+z^{48}+z^{45}+z^{41}+z^{38}+z^{37}+z^{36}+z^{33}+z^{31}+z^{30}+z^{27}+z^{26}+z^{24}+z^{23}+z^{22}+z^{21}+z^{20}+z^{19}+z^{17}+z^{16}+z^{14}+z^{13}+z^{10}+z^{8}+z^{7}+z^{4}+z^{3}+z.$

We ran our two variants of C2-UD on the two curves to certify the conjectured property that no unexpected isogeny of low degree exists between the two curves.

In 258 cpu-hours we were able to prove that no isogeny of degree $p^c\ell$ for $\ell < 2^{11}$ and c arbitrary exists between the two curves; in 694 cpu-hours we were able to prove that no isogeny of degree less than 2^{13} exists either. We stress the fact that, albeit of little interest, this computation would have been impossible without the (surprising) discovery of C2-UD.

Couveignes vs. Lercier-Sirvent. Finally, we ran experiments on Lercier-Sirvent. Table 9.2 shows timings for the different phases of the algorithm for some isogeny degrees. The first column is the time spent to lift the root of $\Phi_\ell(X, j_E)$ in $\mathbb{Q}_q$ and to apply Elkies' formulas. DiffSolve is the time spent solving the differential equation, it is clearly the most expensive phase, although asymptotically not the most important one. RFR is the time for rational fraction reconstruction.

We also compared the running times of C2-AS-FI-MC and Lercier-Sirvent over curves of half the cryptographic size in figure 9.2 (left) and five times the cryptographic size in figure 9.2 (right). We only plot average times for C2, in characteristic 2 we only plot the timings for `GF2`. From the plot it is clear that C2-AS-FI-MC

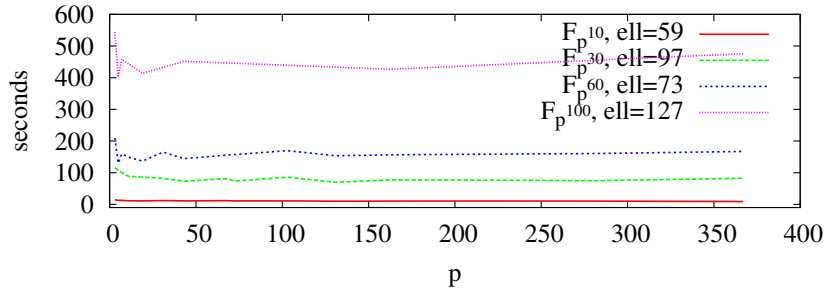


Figure 9.3: Timings for Lercier-Sirvent for different fields. We increase p while keeping constant d and the isogeny degree.

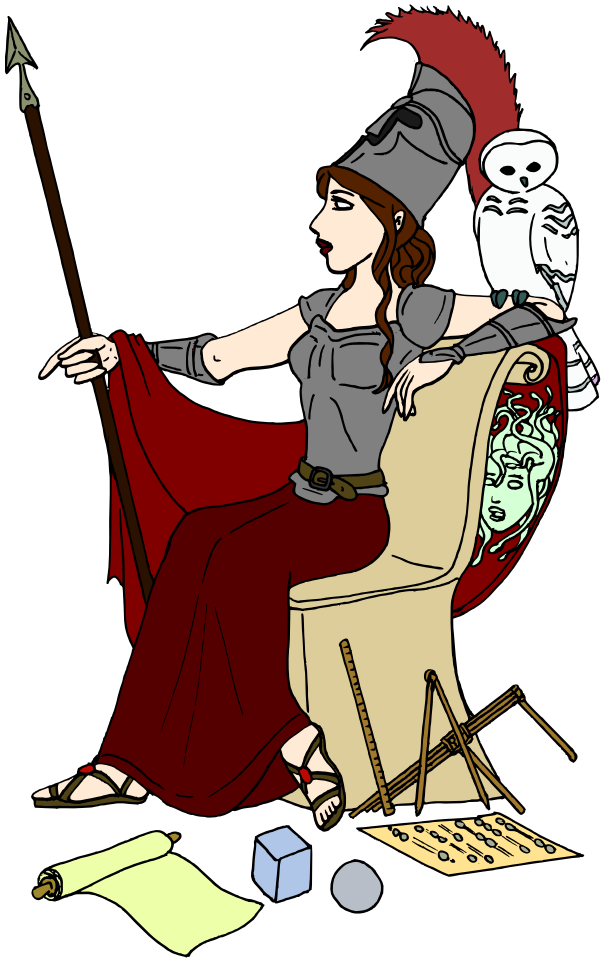
only performs better than Lercier-Sirvent for $p = 2$, but in this case Lercier's algorithm [Ler96] is much faster. The slant of the plots of Lercier-Sirvent looks worse than those of C2-AS-FI-MC; however comparing a Magma prototype to our highly optimized implementation of C2-AS-FI-MC is somewhat unfair.

Furthermore, it is unlikely that C2-AS-FI-MC could be practical for any $p > 3$ because of its high dependence on p, while Lercier-Sirvent scales pretty well with the characteristic as shown in figure 9.3.

Considering that the asymptotic dependency of Couveignes' algorithm in $\log q$ and in p is worse than the one of Lercier-Sirvent (compare Eq. (8.59) to Proposition 8.6), there are very few regions where Couveignes' algorithm stays of practical or theoretical interest.

Ironically, the techniques presented in this document were developed in view of an efficient implementation of Couveignes' algorithm, but, for the moment, their only practical application seems to be C2-UD. Our hope is that other interesting applications may be found in the future.

Appendices



A Appendix



Categorical considerations

The aim of this chapter is to give a simpler proof of the transposition theorem for arithmetic circuits shedding new light on it. The main idea is to bring duality back where it belongs: category theory. This is done through the use of some basic categorical semantics [Pit01, AL91]. We also discuss the relationship with some Haskell type classes and perspectives for the implementation of the transposition principle. This chapter is joint work with Boespflug.

A.1 Categorical semantics of arithmetic circuits

Categorical semantics introduce the notion of *structure valued in a category* $\mathcal{C}$, which is a generalization of the concept of *structure* in model theory. This permits us to reason about operations that "preserve some algebraic structure". See [Pit01] for formal definitions.

Take the example of arithmetic circuits. In Section 3.1 we defined them using left modules and morphisms, this allowed us to state that the evaluation of a circuit is a module morphism simply because the composition laws for arithmetic circuits "preserve" the structure. Then in Section 3.2 we extended the notion of circuit to embrace circuits containing multiplication nodes; however this meant that we had to redefine circuits from scratch, considering maps that are not module morphisms (we did not actually write the new definition in Section 3.2, because we were lazy).

The clean way to give both definitions at once, is to consider *circuits valued in a category* $\mathcal{C}$. It is soon evident that the only requirement on the category is that it has finite products; in what follows, we let $\mathcal{C}$ be a category with finite products.

The definition of the syntax of circuits, i.e. what are the nodes and ports, how they are composed to build a circuit, etc., stay the same; we only change the definitions of operator and of evaluation of an arithmetic circuit.

Definition A.1 (Arithmetic operator, arity) Let A be an object of $\mathcal{C}$. An *arithmetic operator* over A is an arrow $f : \prod^i A \to \prod^o A$ for some $i, o \in \mathbb{N}$. Here i is called the *in-arity* of f or simply *arity*, o is called the *out-arity* of f.

Definition A.2 (Evaluation of an arithmetic circuit) Let A be an object of $\mathcal{C}$. Let C be an arithmetic circuit with i inputs and o outputs, then its evaluation is an arrow $\mathrm{eval}_C : \prod^i A \to \prod^o A$.

In order to define it, we simultaneously define the evaluation eval_v of each $v \in V$ and the evaluation eval_e of each $e \in E$. We will denote by $<_v$ the orders on the input and the output ports of v.

- Let $v \in V$ have out-degree n, let its evaluation be $\mathrm{eval}_v : \prod^i A \to \prod^n A$ and let $\pi_1, \ldots, \pi_n$ be the canonical projections from $\prod^n A$ to A. Let $o_1 <_v \cdots <_v$

o_n be the output ports of v and let $e_j = \big(o_j, E(o_j)\big)$ be the corresponding edges stemming from v, then $\mathrm{eval}_{e_j} = \pi_j \circ \mathrm{eval}_v$ for any j.

- Let $x_1 <_i \cdots <_i x_i$ be the input nodes and let $\pi_1, \dots, \pi_i$ be the canonical projections from $\prod^i A$ to A, then $\mathrm{eval}_{x_j} = \pi_j$ for any j.
- For every evaluation node v with in-degree m, let $i_1 <_v \cdots <_v i_m$ be the input ports of v and let $e_j = \big(E^{-1}(i_j), i_j\big)$ be the corresponding edges incident to v, then

$$\mathrm{eval}_v = \beta(v) \circ (\mathrm{eval}_{e_1} \times \cdots \times \mathrm{eval}_{e_m}). \tag{A.1}$$

- For every output node y, let $e \in E$ be the only edge incident to y, then $\mathrm{eval}_y = \mathrm{eval}_e$.

We can finally define $\mathrm{eval}_C : \prod^i A \to \prod^o A$. Let $y_1 <_o \cdots <_o y_o$ be the output nodes, then

$$\mathrm{eval}_C = (\mathrm{eval}_{y_1} \times \cdots \times \mathrm{eval}_{y_o}). \tag{A.2}$$

We also say that C *computes* eval_C.

As the reader will have noticed, we have simply taken Definition 3.6 and changed direct sums with categorical products. Hence, the fact that the evaluation of a circuit in the category of left modules is a left module morphism is now tautology; but we can also consider circuits valued in **Set**, then all the theory of Section 3.2 can be carried out on those circuits.

A.2 Coevaluation

When dealing with a construction in category theory, it is natural to simultaneously study its dual, that is the construction obtained by *reversing all the arrows*. If in definition 3.6 we substitute the product $\prod^n R$ by its dual $\coprod^n R$, we obtain a new way of evaluating an arithmetic circuit that we will call *coevaluation*. In this section we let $\mathcal{D}$ be a category with finite coproducts.

An arithmetic cobasis is just an arithmetic basis in $\mathcal{D}^{op}$, and the coevaluation of an arithmetic circuit is just its evaluation in $\mathcal{D}^{op}$. For completeness, we give the detailed definitions.

Definition A.3 (Arithmetic co-operator, arity) Let A be an object of $\mathcal{D}$. An *arithmetic co-operator* over A is an arrow $f : \coprod^i A \to \coprod^o A$ for some $i, o \in \mathbb{N}$.

Definition A.4 (coevaluation of an arithmetic circuit) Let A be an object of $\mathcal{D}$. Let C be an arithmetic circuit with i inputs and o outputs over a cobasis $\mathcal{B}$. Its coevaluation is an arrow $\mathrm{coeval}_C : \coprod^i A \to \coprod^o A$.

We use the same notation as in the previous definition. As we did there, we simultaneously define coeval_v for each $v \in V$ and coeval_e for each $e \in E$.

- Let $v \in V$ have in-degree m, let its coevaluation be $\mathrm{coeval}_v : \coprod^m A \to \coprod^o A$ and let $\iota_1, \dots, \iota_n$ be the canonical injections from A to $\coprod^m A$. Let $i_1 <_v \cdots <_v i_m$ be the input ports of v and let $e_j = \big(i_j, E^{-1}(i_j)\big)$ be the corresponding edges incident to v, then $\mathrm{coeval}_{e_j} = \mathrm{coeval}_v \circ \iota_j$ for any j.

- Let $y_1 <_V \cdots <_V y_n$ be the output nodes and let $\iota_1, \ldots, \iota_o$ be the canonical injections from A to $\coprod^o A$, then $\mathrm{coeval}_{y_j} = \iota_j$ for any j.
- For every evaluation node v with out-degree n, let $o_1 <_v \cdots <_v o_n$ be the output ports of v and let $e_j = \big(E(o_j), o_j\big)$ be the corresponding edges stemming from v, then

$$\mathrm{coeval}_v = (\mathrm{coeval}_{e_1} + \cdots + \mathrm{coeval}_{e_n}) \circ \beta(v). \tag{A.3}$$

- For every input node x, let $e \in E$ be the only edge stemming from x, then $\mathrm{coeval}_x = \mathrm{coeval}_e$.

We can finally define $\mathrm{coeval}_C : \coprod^i A \to \coprod^o A$. Let $x_1 <_V \cdots <_V x_i$ be the input nodes, then

$$\mathrm{coeval}_C = \mathrm{coeval}_{x_1} + \cdots + \mathrm{coeval}_{x_i}\,. \tag{A.4}$$

The coevaluation in general does not attach the same semantics to a circuit as the evaluation. For example in the case of ***Set*** the coevaluation is a function from the disjoint union of i copies of A to the disjoint union of o copies of A. We can regard circuits over cobases in ***Set*** as objects that are fed one single element of A on one out of their n inputs and then take decisions depending on which input was fed. An example is given in figure A.1.

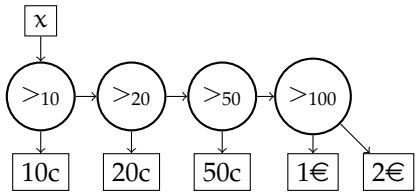


Figure A.1: The coffee machine circuit. On input $n \in \mathbb{Z}$, the operator $>_x : \mathbb{Z} \to \mathbb{Z} \uplus \mathbb{Z}$ gives n on its right output if $n > x$, on its left output otherwise. The circuit is an euro coin separator.

In some cases, howevever, evaluation and coevaluation coincide. Recall that an *additive category* is a category if every hom-set is an Abelian group, composition of morphisms is bilinear, and every finite biproduct exists [ML98, VIII.2]. In particular, in an additive category finite products and coproducts are isomorphic.

Lemma A.5 *Let* C *be a circuit valued in an additive category, then* $\mathrm{eval}_C \cong \mathrm{coeval}_C$ *naturally.*

We just sketch the proof.

Proof. First observe that since products and coproducts are naturally isomorphic, the basis of C can be interpreted both as a basis and a cobasis. Hence both evaluation and coevaluation C are meaningful.

Then, we proceed by induction on the size of the circuit. First, it is obvious that for circuits with one unique evaluation node v we have $\mathrm{eval}_C \cong \beta(v) \cong \mathrm{coeval}_C$. Now if C has n evaluation nodes, we choose any topological order on C and remove the last evaluation node v and the output nodes connected to it. This new circuit satisfies the lemma by induction. The claim follows by connecting v back to the circuit. ∎

A.3 The tranposition theorem

We restate the notion of dual circuit in our new context. The dual circuit is obtained by reversing all the arrows, and it is valued in the opposite category $\mathcal{C}^{op}$. Although we use the same notation, the reader shall not confuse this definition of the dual circuit C^{op} with the definition of the *opposite circuit* we gave in Section 3.2.2.

Definition A.6 (Dual basis) Let A be an object of $\mathcal{C}$ and let $\mathcal{B}$ be an arithmetic basis over A. We define the dual basis $\mathcal{B}^{op}$ as

$$\mathcal{B}^{op} = \{f^{op} \mid f \in \mathcal{B}\}. \tag{A.5}$$

Definition A.7 (Dual circuit) Let A be an object of $\mathcal{C}$. Let $C = (V, E, \leqslant, \leqslant_i, \leqslant_o)$be a circuit over $(A, \mathcal{B})$. For any $v \in V$ define

$$v^{op} = \begin{cases} (O, I, f^{op}) & \text{if } v = (I, O, f) \text{ with } f \neq \emptyset, \\ (O, \emptyset, \emptyset) & \text{if } v = (\emptyset, O, \emptyset), \\ (\emptyset, I, \emptyset) & \text{if } v = (I, \emptyset, \emptyset). \end{cases} \tag{A.6}$$

The *dual circuit* of C, denoted by C^{op}, is the circuit over $(A, \mathcal{B}^{op})$ defined as

$$C^{op} = (V^{op}, E^{-1}, \leqslant^{op}, \leqslant'_i, \leqslant'_o),$$

where $V^{op} = \{v^{op} | v \in V\}$ and the orderings are defined as follows:

$$v \leqslant v' \;\Leftrightarrow\; v'^{op} \leqslant^{op} v^{op}, \tag{A.7}$$

$$v \leqslant_o v' \;\Leftrightarrow\; v^{op} \leqslant'_o v'^{op}, \tag{A.8}$$

$$v \leqslant_i v' \;\Leftrightarrow\; v^{op} \leqslant'_i v'^{op}. \tag{A.9}$$

We now have all the elements to prove the transposition theorem. We let $\mathcal{C}$ and $\mathcal{C}'$ be categories with finite products. The first, trivial, observation is that certain functors "preserve" the semantic of a circuit. If $F : \mathcal{C} \to \mathcal{C}'$ is a functor, we define by $F(\mathcal{B})$ the basis obtained by substituting any arrow f with $F(f)$, and by $F(C)$ the circuit obtained by substituting any node with the corresponding node in $F(\mathcal{B})$.

Proposition A.8 *Let $F : \mathcal{C} \to \mathcal{C}'$ be a continuous functor (i.e. a functor that preserves small limits; we actually only need it to preserve products). Let C be a circuit valued in $\mathcal{C}$. Then $F(C)$ is valued in $\mathcal{C}'$ and $F(\mathrm{eval}_C) = \mathrm{eval}_{F(C)}$.*

Corollary A.9 *Let $\mathcal{C}$ be a category with finite products, $\mathcal{D}$ a category with finite coproducts and $F : \mathcal{C} \to \mathcal{D}^{op}$ a continuous functor. Then $F(C)^{op}$ is valued in $\mathcal{D}$ and $F(\mathrm{eval}_C)^{op} = \mathrm{coeval}_{F(C)^{op}}$.*

Corollary A.10 (Transposition theorem) *Let $\mathcal{C}$ be an additive category and let $F : \mathcal{C} \to \mathcal{C}^{op}$ be a continuous functor. Then $F(\mathrm{eval}_C)^{op} \cong \mathrm{eval}_{F(C)^{op}}$.*

The transposition theorem of Section 3.1.3, then follows by considering the transposition functor $()^* : R\text{–}\mathbf{Mod} \to R\text{–}\mathbf{Mod}$ (note that the transposition functor is traditionally written in a contravariant fashion).

A.4 From circuits to function-level programming

People who think that categories are just abstract nonsense, may be surprised discovering that arithmetic circuits valued in **Hask** (the category of Haskell types)

are already implemented in Haskell. Alternatively, they might think that Haskell is concrete nonsense.

The package `Control.Arrow` implements what are commonly called *arrows* in Haskell jargon. Arrows were introduced in [Hug98] as a generalization of *monads*, they have been successfully applied to many different settings such as, for example, solving ordinary differential equations [LH10]. Paterson [Pat01] was the first to realize the relationship between circuits and arrows, and to propose a DSL for arrows that is amazingly similar to straight line programs.

The standard library class `Arrow` is roughly equivalent to arithmetic circuits valued in **Hask** (or **Set**), while `ArrowChoice` is roughly equivalent to arithmetic circuits valued in $\mathbf{Hask}^{\mathrm{op}}$. So we asked ourselves the question of whether it is possible to write a type class `AdditiveArrow` that has the same properties of circuits valued in additive categories. A desirable feature of *additive arrows* is that they could be evaluated both in $\mathcal{C}$ and $\mathcal{C}^{\mathrm{op}}$, thus they share some similarities with *invertible arrows* [ASvW+05].

We sketch what an additive arrow should look like, by giving an hypothetical list of type classes; following [Yor09], we use infix operators `(~>)` instead of prefix ones as in the standard Haskell library. As for arrows, we start from the class `Category`.

```
class Category (~>) => where
  id :: (a ~> a)
  (.) :: (b ~> c) -> (a ~> b) -> (a ~> c)
```

In order to behave as a category, an instance of this class shall form a monoid for the operation `(.)`, with `id` being the identity element. Now this class can be extended to model additive categories: we first define a class that mimics *Ab-categories*, or *preadditive* categories, that is categories whose hom-sets are Abelian groups, then we define additive categories.

```
class Category (~>) => AbCategory (~>) where
  zeroArrow :: (a ~> b)
  (<+>) :: (a ~> b) -> (a ~> b) -> (a ~> b)


class AbCategory (~>) => AdditiveCategory (~>) where
  (&&&) :: (a ~> b) -> (a ~> c) -> (a ~> (b, c))
  (|||) :: (a ~> c) -> (b ~> c) -> ((a, b) ~> c)
  (***) :: (a ~> b) -> (c ~> d) -> ((a, c) ~> (b, d))
```

Where `&&&` roughly corresponds to the operator & of arithmetic circuits, `|||` corresponds to $+$, and `***` corresponds to forming a new circuit by putting two circuits side by side.

However, these type classes cannot be implemented as expected because Haskell tuples do not behave like $\mathbb{R}^n$: in particular, it is impossible to properly implement the operators `|||` and `***` on tuples. To circumvent this, we have to use some form of dependent types [KLS04, McB03] to encode the free module $\mathbb{R}^{n+m}$ and its projections over $\mathbb{R}^n$ and $\mathbb{R}^m$.

After some unsuccessful experiments with GADT's, we succeeded in implementing additive circuits in the category of $\mathbb{Z}$-modules and transposable multiplication in $\mathbb{Z}[X]$ using type level arithmetic from the package `Data.TypeLevel`. We thank Jacques Carette for having suggested this solution to us.

The source code is presented in the next section. Notice, however that our implementation needs to suggest some trivialities to the type checker (for example, $a \leqslant a + b$ for any $b \in \mathbb{N}$) in order for the compilation to succeed.

Another problem is that the implementation of polynomial multiplication is far from being self-evident. In fact, we had to follow Kiselyov and Peyton-Jones [KPJ08] to implement *advanced overlapping instances*.

Future research directions include:

- use a language natively implementing dependent data types to avoid hacks;
- implement a DSL similar to Paterson's do-notation for arrows [Pat01].

Our hope is that these techniques could provide an efficient and easy to use library for automated theorem provers, to prove the correctness of programs based on the transposition principle.

A.5 Implementation of *self-transposing* polynomial multiplication in Haskell

```
import Data.TypeLevel hiding (Mul)
import Data.Param.FSVec
import qualified Prelude as P

------ The ring

type R = P.Int
zero :: R
zero = 0::P.Int
plus :: R -> R -> R
plus = (P.+)
times :: R -> R -> R
times = (P.*)

------ Additive circuits

class Category (~>) where
  id :: Nat a => a -> (a ~> a)
  (.) :: (Nat a, Nat b, Nat c) => (b ~> c) -> (a ~> b) -> (a ~> c)

class Category (~>) => AbCategory (~>) where
  zeroArrow :: (Nat a, Nat b) => a -> b -> (a ~> b)
  (<+>) :: (Nat a, Nat b) => (a ~> b) -> (a ~> b) -> (a ~> b)

class AbCategory (~>) => AdditiveCategory (~>) where
  (&&&) :: (Nat a, Nat b, Nat c, Add b c bc,
            Min bc b b)
           => (a ~> b) -> (a ~> c) -> (a ~> bc)
  (|||) :: (Nat a, Nat b, Nat c, Add a b ab,
            Min ab a a)
           => (a ~> c) -> (b ~> c) -> (ab ~> c)
  (***) :: (Nat a, Nat b, Nat c, Nat d, Add a c ac, Add b d bd,
            Min ac a a, Min bd b b)
           => (a ~> b) -> (c ~> d) -> (ac ~> bd)

------ Bicategories

data Cat a b = Cat a b
               (FSVec a R -> FSVec b R)
               (FSVec b R -> FSVec a R)
```

```
apply :: Cat a b -> FSVec a R -> FSVec b R
apply (Cat n m f g) x = f x
transapply :: Cat a b -> FSVec b R -> FSVec a R
transapply (Cat n m f g) x = g x

instance Category Cat where
  id n = Cat n n
         (\x -> x)
         (\x -> x)
  (Cat n m f g) . (Cat n' m' f' g') = Cat n' m
                                      (\x -> f (f' x))
                                      (\x -> g' (g x))

instance AbCategory Cat where
  zeroArrow n m = Cat n m
                  (\x -> iterate m (\x -> zero) zero)
                  (\x -> iterate n (\x -> zero) zero)
  (Cat n m f g) <+>
    (Cat n' m' f' g') = Cat n m
                        (\x -> zipWith plus (f x) (f' x))
                        (\x -> zipWith plus (g x) (g' x))

instance AdditiveCategory Cat where
  (Cat n m f g) &&& (Cat n' m' f' g') =
    Cat n (m + m')
    (\x -> (f x) ++ (f' x))
    (\x -> zipWith plus (g (take m x)) (g' (drop m x)))
  (Cat n m f g) ||| (Cat n' m' f' g') =
    Cat (n + n') m
    (\x -> zipWith plus (f (take n x)) (f' (drop n x)))
    (\x -> (g x) ++ (g' x))
  (Cat n m f g) ***
    (Cat n' m' f' g') = Cat (n + n') (m + m')
                        (\x -> f (take n x) ++ f' (drop n x))
                        (\x -> g (take m x) ++ g' (drop m x))

-- injections
scalar r = Cat d1 d1
           (\x -> singleton (times (head x) r))
           (\x -> singleton (times (head x) r))

---- Scalar multiplication

-- HList hacks, following Kiselyov and Peyton-Jones:
-- http://www.haskell.org/haskellwiki/GHC/AdvancedOverlap
-- instead of ShowPred as in the online article, we use Trich as
-- provided by Data.Typelevel

class Nat a => SMul a where
  smul :: R -> Cat a a

class Nat a => SMul' flag a where
  smul' :: flag -> R -> Cat a a

-- the instantiations of SMul'
instance SMul' EQ D0 where
  smul' _ x = zeroArrow d0 d0

instance (Nat b, Succ b a, Trich b D0 f,
          SMul' f b, Add b D1 a, Min a D1 D1)
         => SMul' GT a where
```

```
  smul' _ x = (scalar x) ***
              ((smul'::f -> R -> Cat b b) P.undefined x)

-- and finally the instantiation of SMul !!
instance (Trich a D0 flag, SMul' flag a) => SMul a where
  smul = smul' (P.undefined::flag)


---- Full multiplication

class (Nat preca, Succ preca a, Nat b, Add preca b c)
      => Mul preca a b c where
  mul :: FSVec a R -> Cat b c

class (Nat preca, Succ preca a, Nat b, Add preca b c)
      => Mul' flag preca a b c where
  mul' :: flag -> FSVec a R -> Cat b c

-- the instantiations of Mul'
instance (Nat b, Add D0 b b, Trich b D0 f, SMul' f b)
         => Mul' EQ D0 D1 b b where
  mul' _ x = smul (head x)

instance (Nat ppa, Succ ppa pa, Succ pa a, Nat b,
          Add ppa b pc, Add pa b c,
          Trich ppa D0 f, Mul' f ppa pa b pc,
          Trich b D0 f', SMul' f' b,
          Add pc D1 c, Min c D1 D1, Min b D0 D0,
          Add b D0 b, Min b b b, Min c b b)
         => Mul' GT pa a b c where
  mul' _ x = (zeroArrow d0 d1 ***
              (mul'::f -> FSVec pa R -> Cat b pc)
              P.undefined (tail x))
             <+>
             ((smul::R -> Cat b b) (head x) ***
              zeroArrow P.undefined P.undefined)

-- and finally the instantiation of Mul !!
instance (Trich pa D0 flag, Mul' flag pa a b ab)
         => Mul pa a b ab where
  mul = mul' (P.undefined::flag)

-- Run ghci -fglasgow-exts -XUndecidableInstances
-- then try (for example)
--   ghci> apply (mul (1 +> 2 +> empty)) (1 +> 2 +> 45 +> 10 +> empty)
--   ghci> transapply (mul (1 +> 2 +> empty)) (1 +> 2 +> 45 +> 10 +> empty)
-- enjoy
```

B Appendix



Linearity inference of Karatsuba multiplication

We show here an example of inference of linearity in Haskell, using the technique described in Section 4.1. We define type classes `Ring` and `Module` to represent left-linear operations on rings and free modules. We instantiate them with integers as base ring, and lists of integers as free module (representing polynomials over $\mathbb{Z}[X]$).

We implement Karatsuba multiplication over $\mathbb{Z}[X]$, using only the methods defined in `Ring` and `Module`. This allows the type checker to deduce that Karatsuba multiplication is linear in its first argument, once the second argument and the degree of the polynomials are fixed. The code makes use of functional dependencies, it must be run with the switch `-fglasgow-exts` on.

```
-- Linear and Scalar wrappers
newtype L = Lin Integer deriving (Show)
newtype S = Sca Integer deriving (Num, Show, Eq)

-- The ring
class Ring r where
  zero :: r
  (<+>) :: r -> r -> r
  (<*>) :: r -> S -> r
  neg :: r -> r

-- Free modules
class Ring r => Module m r | m -> r where
  zeroM :: m
  (<<*) :: m -> S -> m
  (>>>) :: m -> Integer -> r
  (<<<) :: r -> Integer -> m
  (<++>) :: m -> m -> m
  add :: m -> m -> Integer -> m
  add a b n = foldl (<++>) zeroM
              [((a>>>i) <+> (b>>>i))<<<i | i <- [1..n]]

-- Linear is an instance of Ring
instance Ring L where
  zero = Lin 0
  (Lin x) <+> (Lin y) = Lin (x+y)
  (Lin x) <*> (Sca y) = Lin (x*y)
  neg (Lin x) = Lin (-x)
```

```
-- Scalar is an instance of Ring
instance Ring S where
  zero = 0::S
  (<+>) = (+)
  (<*>) = (*)
  neg = negate

-- We can add any other constant we like to S
one = 1::S

-- Lists (polynomials) are free modules
instance Ring r => Module [r] r where
  zeroM = [zero]
  [] <<* x = []
  (x:xs) <<* y = (x <*> y):(xs <<* y)
  [] >>> i = zero
  (x:xs) >>> i =
    if i < 1
    then zero
    else if i == 1
         then x
         else xs >>> (i-1)
  x <<< i = if i <= 1 then [x] else zero:(x <<< (i-1))
  [] <++> [] = []
  (x:xs) <++> [] = x:(xs <++> [])
  [] <++> (y:ys) = y:([] <++> ys)
  (x:xs) <++> (y:ys) = (x <+> y):(xs <++> ys)
  add [] [] n = []
  add [] (y:ys) n =
    if n > 0 then y:(add [] ys (n-1)) else []
  add (x:xs) [] n =
    if n > 0 then x:(add xs [] (n-1)) else []
  add (x:xs) (y:ys) n =
    if n > 0 then (x<+>y):(add xs ys (n-1)) else []


-- Karatsuba multiplication : the system will infer
-- shift :: Ring r => [r] -> Int -> [r]
-- split :: Ring r => [r] -> Int -> ([r], [r])
-- kara :: Ring r => [r] -> [S] -> Int -> [r]

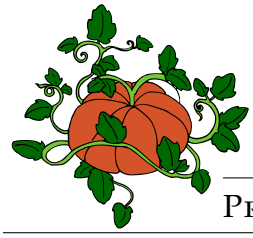
shift x n = if n <= 0 then x else shift (zero:x) (n-1)

split [] n = ([], [])
split (x:xs) n =
  if n <= 0
  then ([], x:xs)
  else let (a, b) = split xs (n-1) in (x:a, b)

kara [] y n = []
kara x [] n = []
kara x y n =
  if n <= 0
  then []
```

```
  else if n == 1
       then [(x!!0) <*> (y!!0)]
       else
         let h = n `div` 2 in
         let (a0, a1) = split x h in
         let (b0, b1) = split y h in
         let x0 = kara a0 b0 h in
         let x2 = kara a1 b1 (n-h) in
         let xx1 = kara (a1 <++> a0) (b1 <++> b0) (n-h) in
         let x1 = xx1 <++> ((x0 <++> x2) <<* (neg one)) in
         (shift x2 n) <++> (shift x1 h) <++> x0

-- Run ghci -fglasgow-exts
-- and at the prompt type
--    ghci> :t kara
-- enjoy
```



Appendix C

Proof of Vélu's formulas

We always had admiration for our colleagues who can develop by hand two pages full of calculations without making mistakes. When it comes to us, we usually make a sign mistake at the third term. Tired of having to check for sign errors in other people's papers any time we had to use Vélu formulas, we decided to make an automatic proof of it.

The following Magma code proves the passage from Eq. (8.1) to Eq. (8.4) and from there to (8.9).

```
// Set the a-invariants and the Weierstrass equation
aInv<a2,a4,a6> := FunctionField(Rationals(),3);
_<X> := PolynomialRing(aInv);
f := X^3 + a2*X^2 + a4*X + a6;
fprim := Derivative(f);

// Set the affine coordinate ring
// (variables repeated twice)
P<Xp,Xq,Yp,Yq> := PolynomialRing(aInv, 4);
p<xp,xq,yp,yq> := quo<P|Yp^2-Evaluate(f,Xp),
                        Yq^2-Evaluate(f,Xq)>;

// The numerators of x(P+Q) - x(Q) and y(P+Q) - y(Q)
// using additions formulas
numlambda := yp - yq;
denlambda := xp - xq;

xPplusQ := numlambda^2 + (-a2 -xp -xq)*denlambda^2;
xPplusQminusxQ := xPplusQ - xq*denlambda^2;

yPplusQ := -numlambda*xPplusQ + numlambda*xp*denlambda^2
    - yp*denlambda^3;
yPplusQminusyQ := yPplusQ - yq*denlambda^3;

// Velu summands
veluX := Evaluate(fprim,xq)*denlambda + 2*Evaluate(f,xq);
veluY := -yp*Evaluate(fprim,xq)*denlambda
    - 4*yp*Evaluate(f,xq);

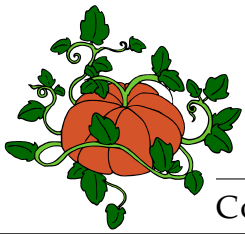
// Here's the proof!
// Observe how these quantities only have odd powers of yq
veluX - xPplusQminusxQ;
veluY - yPplusQminusyQ;
```

```
// Elkies summmands (only abscissa)
elkiesX := (xp - xq)*denlambda^2
    - Evaluate(fprim,xp)*denlambda
    + 2*Evaluate(f,xp);

// The proof, again (this time we directly get 0)
elkiesX - veluX;
// Done!
```

The first and second line of output are the differences between each term of the sums in Eqs. (8.1) and (8.4). In both lines, to conclude one must observe that all the terms in the difference contain an odd power of $y(Q)$, thus they sum up to 0 over G^*.

The third line is the difference between each term of the sums in Eqs. (8.4) and (8.9). The result is self-explanatory.



Conclusion

We have presented our contributions to the study of efficient algorithms for towers of finite fields and isogenies between elliptic curves. In view of these applications, we have employed advanced algebraic and algorithmic techniques, and developed new tools that have an interest of their own. Our contributions span three directions.

Transposition principle. Before this work, the transposition principle used to be considered difficult to apply, and transposed algorithms were notoriously difficult to interpret and implement correctly. Following [BLS03], one could apply the principle in an automatic fashion to a very special class of algebraic algorithms, that we call *algebraic transforms* in Section 3.3. More general algebraic algorithms, such as polynomial multiplication or Euclidean division, were still treated case-by-case.

In this document we have shown that typed functional languages allow to automatically infer the linear algebraic structure of any algebraic algorithm. Once this structure is known, the transposition of the algebraic algorithm is automatically produced by partial evaluation. It would be interesting to explore new ways of implementing the transposition principle in higher order languages such as Haskell, Coq or Agda, as this could have applications to the formal verification of computer algebra systems by automated theorem provers. We have sketched some relevant ideas in Appendix A.

Towers of finite fields. With the help of transposed algorithms, we have constructed a family of Artin-Schreier towers of finite fields with quasi-optimal arithmetic operations. Thanks to Couveignes' algorithm [Cou00], such fast arithmetic generalize to any Artin-Schreier tower.

Since any separable extension of degree equal to the characteristic is Artin-Schreier, our construction provides –at least in theory– fast arithmetics for any such tower of extensions. This can be applied, for example, to the computation of torsion points of Abelian varieties, as we did in this document. It would be interesting to generalize this construction to the case of function fields, as this could have applications to coding theory [GS96, SAK+01].

Elliptic curves. Using our construction for Artin-Schreier towers, we were able to give the first complete implementation of Couveignes' second algorithm for isogeny computation [Cou96]. This, together with a further improvement we have presented in Section 8.7, yields an algorithm whose complexity is quadratic in the degree of the isogeny.

The comparison of our implementation with Lercier and Sirvent's algorithm [LS08] concludes in favor of the latter, however our improvements to Couveignes'

algorithm stay of theoretical interest for several reasons. First, Couveignes' algorithm can be easily generalized to Jacobians of hyperelliptic curves, although with a much worse complexity. Improving such generalization, at least for the case of genus 2 hyperelliptic curves, would be of some relevance for point counting [Sch95, Pil90, GS04]; although p-adic methods are likely to remain the best algorithms for the small characteristic case [Ked01, DV06].

Second, our generalization of Couveignes' algorithm to compute isogenies of unknown degree sheds new light on Couveignes' algorithm and on the complexity of the isogeny computation problem; and could have applications in cryptography [Tes06, RS06]. Looking for similar generalizations of other algorithms, such as Couveignes' first algorithm [Cou94], is a first step towards a better understanding of the problem, and could ultimately lead to an optimal algorithm to compute isogenies of given degree between elliptic curves: a result that is still out of reach today.



List of symbols

$\mathbb{A}^n$	n-dimensional affine space	16
$\mathbb{P}^n$	n-dimensional projective space	16
$\lvert x\rangle$	Ket, element of a left module	12
$\langle x\rvert$	Bra, element of a right module	12
$\langle x\rvert y\rangle_f$	Bilinear form	12
$\langle x\rvert g\rvert y\rangle$	Bilinear form with linear operator	13
$\mathrm{depth}(C)$	Depth of an arithmetic circuit	29
$\mathrm{size}_X(C)$	Size (X-weighted) of an arithmetic circuit	29
$L(n)$	Push-down/Lift-up function (Artin-Schreier towers)	74
$C(n)$	Modular composition function	17
$M(n)$	Multiplication function	17
Ω	big-Omega complexity notation	17
$PT(n)$	Pseudotrace function (Artin-Schreier towers)	79
Θ	big-Theta complexity notation	17
$\tilde{O}$	soft-Oh complexity notation	17
O	big-Oh complexity notation	17
Δ_E	Discriminant of an elliptic curve E	88
det	Determinant of a matrix, of a linear operator	13
$\mathbf{B}^*$	Dual basis	13
g^*	Dual operator: $\langle g(x)\rvert y\rangle = \langle x\rvert g^*(y)\rangle$	13
M^*	Dual of a module or vector space: $M^* = \hom(M, R)$	12
$E(\mathbb{K})$	Set of $\mathbb{K}$-rational points of an elliptic curve E	89
$E[m]$	m-torsion subgroup of an elliptic curve E	91
eval_C	Evaluation of an arithmetic circuit	29
Φ_n	n-th Cyclotomic polynomial	14
φ	Euler totient function	14
φ_q, φ	Frobenius automorphism	15
ϕ	Frobenius isogeny: $(x, y) \mapsto (x^p, y^p)$	93
ϕ_q, ϕ_E	Frobenius endomorphism of an elliptic curve	94
Φ_ℓ	ℓ-th modular polynomial	95
$\mathbb{F}_q$	Finite field of cardinality q	15
G_{2k}	Eisenstein series	93
$\mathrm{Gal}(\mathbb{L}/\mathbb{K})$	Galois group	14
$\sqrt{I}$	Radical of an ideal	15
$I(V)$	Ideal vanishing at the algebraic set V	16
$V(\mathbb{K})$	$\mathbb{K}$-rational points of an algebraic set	16
$V(I)$	Set of zeros of the ideal I	15
j_E	j-invariant of an elliptic curve E	89
K_E	Kummer variety of an elliptic curve	90

$\mathbb{K}^G$	Fixed field, the subfield of $\mathbb{K}$ fixed by the action of G	14
$\mathbb{K}(V)$	Function field of an algebraic variety	16
$\mathbb{K}[V]$	Coordinate ring of an algebraic variety	16
$[m]P$	Scalar multiple of a point of an elliptic curve	90
μ_n	Group of the n-th roots of unity	14
$N_{\mathbb{L}/\mathbb{K}}$	Norm of a field extension	14
$\mathcal{O}$	Point at infinity of an elliptic curve	88
$\wp$	Weierstrass $\wp$-function	93
$\mathrm{in}(v)$	Input ports of a node	28
$\mathrm{out}(v)$	Output ports of a node	28
$\mathcal{T}_\ell(E)$	ℓ-adic Tate module	91
Tr	Trace of a matrix, of a linear operator	13
$\mathrm{Tr}_{\mathbb{L}/\mathbb{K}}$	Trace of a field extension	14
$\mathbb{Z}_p$	p-adic integers	91



Index

AD, *see* automatic differentiation
adjoint code, 40
affine space, 16
algebraic algorithm, 39
algebraic set
    affine, 16
    defined over a field, 16
    irreducible, 16
    projective, 16
algebraic transform, 38
algebraic type, 48, 49
arithmetic basis, 27
    commutative, 27
arithmetic circuit, 27, 28, 122
    depth, 29, 33
    differential, 41
    dual, 31, 124
    evaluation, 29, 122
    ℓ-dual, 35
    opposite, 35
    semantic, 29, 122
    size, 29, 33
    substitution, 29, 30
    valued in a category, 122
    weighted size, 29
arithmetic operator, 27, 122
arity, 27, 122
arrow, 124
Artin-Schreier
    extension, 66
    isomorphism of towers, 80
    polynomial, 66
    tower, 66
automatic differentiation, 40
    adjoint code, 40
    checkpoint method, 40
    forward mode, 40
    reverse mode, 40
automatic transposition, 42, 43

Bézout relation, 22
big-Oh, *see* complexity notation
big-Omega, *see* complexity notation
big-Theta, *see* complexity notation
bilinear chain, 36
bilinear form, 12
    degenerate, 12
bra, 12
bra-ket notation, 12

categorical semantics, 122
Cauchy interpolation, 23
checkpoint method, 40
Chinese remainder theorem, 15
chord-tangent law, 89
circuit family, 33
    uniform, 33
complex torus, 93
complexity notation, 17
conjugate element, 14
coordinate ring, 16
cyclotomic polynomial, 14

degree of an isogeny, 92
determinant, 13
differential of a circuit, 41
dimension of a variety, 16
discrete Fourier transform, 18
discrete logarihm problem, 96
discriminant of an elliptic curve, 88
division polynomial, 91
DLP, *see* discrete logarithm problem
double use, 38
dual arithmetic basis, 30
dual basis, 13
dual isogeny, 93
dual module, 12
dual operator, 13

edge, 28

Eisenstein series, 93
electrical network lemma, 31
elliptic curve, 88
    group law, 89
    isomorphic, 91
    ordinary, 91
    supersingular, 91
    torsion subgroup, 91
elliptic function, 93
Euclidean algorithm, 22
Euler function, 14
extended Euclidean algorithm, 22

fast Fourier transform, 18
FFT, *see* fast Fourier transform
Fibonacci number, 38
finite field, 15
formal power series, 18
    composition, 19
    derivative, 18
    exponential, 18
    integral, 18
    logarithm, 18
forward mode, 40
forward sweep, 52
Frobenius automorphism, 15, 20, 77
Frobenius endomorphism, 94
Frobenius isogeny, 93
function field, 16

Galois field extension, 14
Galois group, 14
GCD, 22
GHS attack, 96
Gorenstein algebra, 57
Gunji's formulas, 102

Hasse bound, 94

ideal
    irreducible, 15
    maximal, 15
    primary, 15
    prime, 15
    radical, 15
    reducible, 15
    vanishing, 16
interpolation, 22
isogeny, 92
    degree, 92
    dual, 93
    inseparable, 92
    normalized, 98
    purely inseparable, 92
    separable, 92
isomorphism
    of elliptic curves, 91
iterated Frobenius, 20

j-invariant, 89

Karatsuba multiplication, 17
ket, 12
Kronecker substitution, 23
Kummer variety, 90

Lagrange interpolant, 21
lattice, 93
lift-up, 73
linearization, 34, 43
    left, 34
    right, 34
    trivial, 34
linearized circuit, 34
linear chain, 35
linear edge, 34
linear input, 34, 43

middle product, 24
modular composition, 17, 19
modular polynomial, 95
Montgomery's formulas, 90
morphism of varieties, 16
    defined over a field, 16
multiDAG, 28
multilinear circuit, 33
multiplication function, 17
multipoint evaluation, 21
multivariate basis, 67

Newton's iteration, 19
node, 28
    degree, 28
    evaluation node, 28
    input node, 28
    output node, 28
    value, 28
Noetherian ring, 15
norm, 14, 58
normalized isogeny, 98
normalized model, 98

normal field extension, 14
null edge, 34

Padé approximant, 23
parameter space, 33
partial evaluation, 39
path, 28
    evaluation, 31
point at infinity, 88
port
    input port, 28
    output port, 28
primary decomposition, 15
prime field, 15
primitive element, 14
primitive tower, 67, 69
projective space, 16
pseudotrace, 20, 68, 77
push-down, 73

RAM model, 36
rational fraction reconstruction, 23
rational map, 16
    defined over a field, 16
rational point, 89
rational univariate representation, 58–61, 62, 65, 83
residue, 57
    global, 57
    local, 57
reverse mode, 40
reverse sweep, 52
RFR, *see* rational fraction reconstruction
root of unity, 14
    primitive, 14
RUR, *see* rational univariate representation

scalar edge, 34
scalar input, 34, 43
separable
    element, 14
    field extension, 14
set of zeros, 15
signature, 50
SLP, *see* straight line program
soft Oh, *see* complexity notation
splitting field, 14
standard left-linear basis, 27
standard multilinear basis, 33
straight line program, 36
structure, 122
    valued in a category, 122
subproduct tree, 21
supersingular, 91

Tate module, 91
torsion point, 91
tower of extensions
    Artin-Schreier, 66
    primitive, 67, 69
trace, 13, 58, 68
    of an operator, 13
    of a field extension, 14
trace form, 57
trace formulas, 56–57
trace map, 20
`transalpyne`, 47
transposed algorithm, 24
transposed modular multiplication, 25, 77
transposed modular reduction, 25, 76, 110
transposed multiplication, 24
transposition principle, 24, 40
transposition theorem, 30, 31, 38–40
twist, 92
    degree of a, 92
type class, 44

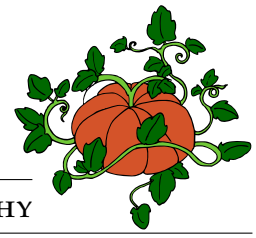
underlying graph, 28
univariate basis, 67

variety
    affine, 16
    algebraic, 16
    projective, 16
vertex, 28
Voloch's formulas, 103

Weierstrass equation, 88
Weierstrass form
    affine, 88
    homogeneous, 88
    simplified form, 92
Weierstrass $\wp$-function, 93

XGCD, 22

zero edge, 33



[ABRW96] María-Emilia Alonso, Eberhard Becker, Marie-Françoise Roy, and Thorsten Wörmann, *Zeros, multiplicities, and idempotents for zero-dimensional systems*, Progress in Mathematics, vol. 143, pp. 1–15, Birkhäuser, Basel, 1996. MR MR1414442

[AL91] Andrea Asperti and Giuseppe Longo, *Categories, types, and structures: An introduction to category theory for the working computer scientist*, Foundations of Computing, The MIT Press, August 1991.

[AM93a] A. O. L. Atkin and François Morain, *Elliptic curves and primality proving*, Mathematics of Computation **61** (1993), no. 203, 29–68.

[AM93b] ———, *Finding suitable curves for the elliptic curve method of factorization*, Mathematics of Computation **60** (1993), 399–405.

[ASvW+05] Artem Alimarine, Sjaak Smetsers, Arjen van Weelden, Marko van Eekelen, and Rinus Plasmeijer, *There and back again: Arrows for invertible programming*, Proceedings of the 2005 ACM SIGPLAN workshop on Haskell, ACM, 2005, pp. 97+.

[Atk88] A. O. L. Atkin, *The number of points on an elliptic curve modulo a prime*, 1988.

[BBLP08] Daniel J. Bernstein, Peter Birkner, Tanja Lange, and Christiane Peters, *ECM using Edwards curves*, Cryptology ePrint Archive, Report 2008/016, January 2008.

[BCG+10] Alin Bostan, Frédéric Chyzak, Marc Giusti, Romain Lebreton, Bruno Salvy, and Éric Schost, *Algorithmes en calcul formel et en automatique*, Lecture notes for the MPRI course 2-22, 2010.

[BCP97] Wieb Bosma, John Cannon, and Catherine Playoust, *The MAGMA algebra system I: the user language*, Journal of Symbolic Computation **24** (1997), no. 3-4, 235–265.

[BCS97a] Wieb Bosma, John Cannon, and Allan Steel, *Lattices of compatibly embedded finite fields*, Journal of Symbolic Computation **24** (1997), no. 3-4, 351–369.

[BCS97b] Peter Bürgisser, Michael Clausen, and Mohammad A. Shokrollahi, *Algebraic complexity theory*, A Series of Comprehensive Studies in Mathematics, vol. 315, Springer, February 1997.

[Ber] Daniel J. Bernstein, *The transposition principle*, Last accessed 26-Aug-2010. `http://cr.yp.to/transposition.html`.

[Ber98] _____, *Composing power series over a finite ring in essentially linear time*, Journal of Symbolic Computation **26** (1998), no. 3, 339–341.

[BGTZ08] Richard P. Brent, Pierrick Gaudry, Emmanuel Thomé, and Paul Zimmermann, *Faster multiplication in GF(2)[x]*, ANTS-VIII'08: Proceedings of the 8th international conference on Algorithmic number theory (Berlin, Heidelberg), Springer-Verlag, 2008, pp. 153–166.

[BK78] Richard P. Brent and Hsiang-Tsung Kung, *Fast algorithms for manipulating formal power series*, Journal of the ACM **25** (1978), no. 4, 581–595.

[BKL98] Valery Bykov, Alexander Kytmanov, and Mark Lazman, *Elimination methods in polynomial computer algebra*, Kluwer Academic Publishers, Norwell, MA, USA, 1998.

[BKSF59] L. M. Beda, L. N. Korolev, N. V. Sukkikh, and T. S. Frolova, *Programs for automatic differentiation for the machine BESM*, Tech. report, Institute for Precise Mechanics and Computation Techniques, Academy of Science, Moscow, USSR, 1959.

[BLMM01] François Boulier, François Lemaire, and Marc Moreno Maza, *PARDI!*, ISSAC '01: Proceedings of the 2001 international symposium on Symbolic and algebraic computation (New York, NY, USA), ACM, 2001, pp. 38–47.

[BLS03] Alin Bostan, Grégoire Lecerf, and Éric Schost, *Tellegen's principle into practice*, ISSAC '03: Proceedings of the 2003 international symposium on Symbolic and algebraic computation (New York, NY, USA), ACM, 2003, pp. 37–44.

[BLS10] Reinier Broker, Kristin Lauter, and Andrew V. Sutherland, *Modular polynomials via isogeny volcanoes*, January 2010.

[BMSS08] Alin Bostan, François Morain, Bruno Salvy, and Éric Schost, *Fast algorithms for computing isogenies between elliptic curves*, Mathematics of Computation **77** (2008), 1755–1778.

[Bor57] J. Bordewijk, *Inter-reciprocity applied to electrical networks*, Applied Scientific Research, Section B **6** (1957), no. 1, 1–74.

[Bre93] Richard P. Brent, *On computing factors of cyclotomic polynomials*, Mathematics of Computation **61** (1993), no. 203, 131–149.

[BS83] Walter Baur and Volker Strassen, *The complexity of partial derivatives*, Theoretical Computer Science **22** (1983), no. 3, 317–330.

[BSS89] Lenore Blum, Mike Shub, and Steve Smale, *On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines*, Bulletin of the American Mathematical Society **21** (1989), no. 1, 1–47.

[BSS99] Ian F. Blake, Gadiel Seroussi, and Niegel P. Smart, *Elliptic curves in cryptography*, Cambridge University Press, New York, NY, USA, 1999.

[BSS03] Alin Bostan, Bruno Salvy, and Éric Schost, *Fast algorithms for Zero-Dimensional polynomial systems using duality*, Applicable Algebra in Engineering, Communication and Computing **14** (2003), no. 4, 239–272.

[Buc65] Bruno Buchberger, *An algorithm for finding a basis for the residue class ring of a Zero-Dimensional polynomial ideal (in german)*, Ph.D. thesis, Institute of Mathematics, University of Innsbruck, Austria, 1965.

[Can89] David G. Cantor, *On arithmetical algorithms over finite fields*, Journal of Combinatorial Theory, Series A **50** (1989), no. 2, 285–300.

[Car87] Luca Cardelli, *Basic polymorphic typechecking*, Science of Computer Programming **8** (1987), 147–172.

[CD93] Charles Consel and Olivier Danvy, *Tutorial notes on partial evaluation*, POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM Press, 1993, pp. 493–501.

[CK91] David G. Cantor and Erich Kaltofen, *On fast multiplication of polynomials over arbitrary algebras*, Acta Informatica **28** (1991), no. 7, 693–701.

[CKS09] Jacques Carette, Oleg Kiselyov, and Chung C. Shan, *Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages*, Journal of Functional Programming **19** (2009), no. 05, 509–543.

[CKY89] John F. Canny, Eric Kaltofen, and Lakshman N. Yagati, *Solving systems of nonlinear polynomial equations faster*, ISSAC '89: Proceedings of the ACM-SIGSAM 1989 international symposium on Symbolic and algebraic computation (New York, NY, USA), ACM, 1989, pp. 121–128.

[CLG09] Denis Charles, Kristin Lauter, and Eyal Goren, *Cryptographic hash functions from expander graphs*, Journal of Cryptology **22** (2009), no. 1, 93–113.

[CLO05a] David Cox, John Little, and Donal O'Shea, *Ideals, varieties, and algorithms : An introduction to computational algebraic geometry and commutative algebra*, (Undergraduate Texts in Mathematics), Springer, July 2005.

[CLO05b] David A. Cox, John Little, and Donal O'Shea, *Using algebraic geometry*, Graduate Texts in Mathematics, Springer-Verlag, Berlin-Heidelberg-New York, March 2005.

[Con91] Ian Connell, *Elliptic curve handbook*, Lecture Notes from class at McGill University, 1991.

[Con00] John H. Conway, *On numbers and games*, 2nd ed., AK Peters, Ltd., December 2000.

[Cou94] Jean-Marc Couveignes, *Quelques calculs en théorie des nombres*, Ph.D. thesis, Université de Bordeaux, 1994.

[Cou96] ———, *Computing l-Isogenies using the p-Torsion*, ANTS-II: Proceedings of the Second International Symposium on Algorithmic Number Theory (London, UK), Springer-Verlag, 1996, pp. 59–65.

[Cou00] ———, *Isomorphisms between Artin-Schreier towers*, Mathematics of Computation **69** (2000), no. 232, 1625–1631.

[DF11] Luca De Feo, *Fast algorithms for computing isogenies between ordinary elliptic curves in small characteristic*, Journal of Number Theory **131** (2011), no. 5, 873–893.

[DFS09] Luca De Feo and Éric Schost, *Fast arithmetics in Artin-Schreier towers over finite fields*, ISSAC '09: Proceedings of the 2009 international symposium on Symbolic and algebraic computation (New York, NY, USA), ACM, 2009, pp. 127–134.

[DFS10] ———, *transalpyne: a language for automatic transposition*, SIGSAM Bulletin **44** (2010), no. 1/2, 59–71.

[DJMMS08] Xavier Dahan, Xin Jin, Marc Moreno Maza, and Éric Schost, *Change of order for regular chains in positive dimension*, Theoretical Computer Science **392** (2008), no. 1-3, 37–65.

[DM82] Luis Damas and Robin Milner, *Principal type-schemes for functional programs*, Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), POPL '82, ACM, 1982, pp. 207–212.

[Dor87] Jean-Louis Dornstetter, *On the equivalence between Berlekamp's and Euclid's algorithms*, IEEE Transactions on Information Theory **33** (1987), no. 3, 428–431.

[DTGV01] Gema M. Díaz-Toca and Laureano González-Vega, *An explicit description for the triangular decomposition of a zero-dimensional ideal through trace computations*, Symbolic computation: solving equations in algebra, geometry and engineering, Contemporary Mathematics, vol. 286, AMS, 2001, pp. 21–35.

[DV06] Jan Denef and Frederik Vercauteren, *An extension of Kedlaya's algorithm to hyperelliptic curves in characteristic 2*, Journal of Cryptology **19** (2006), no. 1, 1–25.

[Elk98] Noam D. Elkies, *Elliptic and modular curves over finite fields and related computational issues*, Computational perspectives on number theory (Chicago, IL, 1995) (Providence, RI), Studies in Advanced Mathematics, vol. 7, AMS International Press, 1998, pp. 21–76. MR MR1486831

[EM03] Andreas Enge and François Morain, *Fast decomposition of polynomials with known galois group*, AAECC'03: Proceedings of the 15th international conference on Applied algebra, algebraic algorithms and error-correcting codes (Berlin, Heidelberg), Springer-Verlag, 2003, pp. 254–264.

[EM07] Mohamed Elkadi and Bernard Mourrain, *Introduction à la résolution des systèmes polynomiaux*, Springer Verlag, 2007.

[ES10] Andreas Enge and Andrew V. Sutherland, *Class invariants by the CRT method*, ANTS IX: Proceedings of the Algorithmic Number Theory 9th International Symposium, Lecture Notes in Computer Science, vol. 6197, Springer–Verlag, 2010, pp. 142–156.

[Fau99] Jean-Charles Faugère, *A new efficient algorithm for computing Gröbner bases (F4)*, Journal of Pure and Applied Algebra **139** (1999), no. 1-3, 61–88.

[Fau02] ———, *A new efficient algorithm for computing Gröbner bases without reduction to zero (F5)*, ISSAC '02: Proceedings of the 2002 international symposium on Symbolic and algebraic computation, 2002.

[FGH00] Mireille Fouquet, Pierrick Gaudry, and Robert Harley, *An extension of Satoh's algorithm and its implementation*, Journal of the Ramanujan Mathematical Society **15** (2000), no. 4, 281–318.

[FGLM93] Jean-Charles Faugère, Patricia Gianni, Daniel Lazard, and Teo Mora, *Efficient computation of zero-dimensional Gröbner bases by change of ordering*, Journal of Symbolic Computation **16** (1993), no. 4, 329–344. MR MR1263871

[Fid73] Charles M. Fiduccia, *On the algebraic complexity of matrix multiplication.*, Ph.D. thesis, Brown University, Providence, RI, USA, 1973.

[GG05] Sergey B. Gashkov and Igor B. Gashkov, *On the complexity of calculation of differentials and gradients*, Discrete Mathematics and Applications **15** (2005), no. 4, 327–350.

[GHS02a] Steven D. Galbraith, Florian Hess, and Nigel P. Smart, *Extending the GHS Weil descent attack*, EUROCRYPT '02: Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (London, UK), Springer-Verlag, 2002, pp. 29–44.

[GHS02b] Pierrick Gaudry, Florian Hess, and Niegel Smart, *Constructive and destructive facets of Weil descent on elliptic curves*, Journal of Cryptology **15** (2002), no. 1, 19–46–46.

[GK08] Shay Gueron and Michael E. Kounavis, *Carry-less multiplication and its usage for computing the GCM mode*, white paper, Intel Corporation, 2008.

[GLS01] Marc Giusti, Grégoire Lecerf, and Bruno Salvy, *A Gröbner free alternative for polynomial system solving*, Journal of Complexity **17** (2001), no. 1, 154–211.

[GLVM91] Jean Gilbert, Georges Le Vey, and John Masse, *La Differentiation automatique de fonctions representees par des programmes*, Research Report RR-1557, INRIA, 1991.

[Gri92] Andreas Griewank, *Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation*, Optimization Methods and software **1** (1992), 35–54.

[GS96] A. Garcia and H. Stichtenoth, *On the asymptotic behaviour of some towers of function fields over finite fields*, Journal of Number Theory (1996), 248–273.

[GS04] Pierrick Gaudry and Éric Schost, *Construction of secure random curves of genus 2 over prime fields*, Advances in cryptology—EUROCRYPT 2004, Lecture Notes in Comput. Sci., vol. 3027, Springer, Berlin, 2004, pp. 239–256.

[Gun76] Hiroshi Gunji, *The Hasse invariant and p-division points of an elliptic curve*, Archiv der Mathematik **27** (1976), no. 1, 148–158.

[GW08] Andreas Griewank and Andrea Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*, second ed., Society for Industrial and Applied Mathematics (SIAM), 2008.

[Hac97] Dirk Hachenberger, *Finite fields: normal bases and completely free elements*, Kluwer Academic Pub, 1997.

[Har09] David Harvey, *Faster polynomial multiplication via multipoint Kronecker substitution*, Journal of Symbolic Computation **44** (2009), no. 10, 1502–1510.

[Hes03] Florian Hess, *The GHS attack revisited*, EUROCRYPT'03: Proceedings of the 22nd international conference on Theory and applications of cryptographic techniques (Berlin, Heidelberg), Springer-Verlag, 2003, pp. 374–387.

[HM73] John E. Hopcroft and Jean Musinski, *Duality applied to the complexity of matrix multiplications and other bilinear forms*, STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing (New York, NY, USA), ACM, 1973, pp. 73–87.

[HP98] Xiaohan Huang and Victor Y. Pan, *Fast rectangular matrix multiplication and applications*, Journal of Complexity **14** (1998), no. 2, 257–299.

[HQZ04] Guillaume Hanrot, Michel Quercia, and Paul Zimmermann, *The middle product algorithm I*, Applicable Algebra in Engineering, Communication and Computing **14** (2004), no. 6, 415–438.

[Hug98] John Hughes, *Generalising monads to arrows*, Science of Computer Programming **37** (1998), 67–111.

[JL06] Antoine Joux and Reynald Lercier, *Counting points on elliptic curves in medium characteristic*, Cryptology ePrint Archive, Report 2006/176, May 2006.

[Kal87] Erich Kaltofen, *Computer algebra algorithms*, Annual Review in Computer Science **2** (1987), 91–118.

[Kal88] ———, *Greatest common divisors of polynomials given by Straight-Line programs*, Journal of the Association for Computing Machinery **35** (1988), no. 1, 231–264.

[Kal00] ———, *Challenges of symbolic computation: My favorite open problems*, Journal of Symbolic Computation **29** (2000), no. 6, 891–919.

[Ked01] Kiran S. Kedlaya, *Counting points on hyperelliptic curves using Monsky-Washnitzer cohomology*, Journal of the Ramanujan Mathematical Society **16** (2001), no. 4, 323–338. MR MR1877805

[KLS04] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke, *Strongly typed heterogeneous collections*, Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell (New York, NY, USA), ACM Press, 2004, pp. 96–107.

[KO63] Anatolii Karatsuba and Yuri Ofman, *Multiplication of multidigit numbers on automata*, Soviet Physics-Doklady **7** (1963), 595–596.

[Kob87] Neal Koblitz, *Elliptic curve cryptosystems*, Mathematics of Computation **48** (1987), no. 177, 203–209.

[KPJ08] Oleg Kiselyov and Simon Peyton-Jones, *GHC/Advanced overlap*, Last accessed 31-Aug-2010 `http://www.haskell.org/haskellwiki/GHC/AdvancedOverlap`, April 2008.

[KS97] Erich Kaltofen and Victor Shoup, *Fast polynomial factorization over high algebraic extensions of finite fields*, ISSAC '97: Proceedings of the 1997 international symposium on Symbolic and algebraic computation (New York, NY, USA), ACM, 1997, pp. 184–188.

[KS98] ———, *Subquadratic-time factoring of polynomials over finite fields*, Mathematics of Computation **67** (1998), no. 223, 1179–1197.

[KU08] Kiran S. Kedlaya and Christopher Umans, *Fast modular composition in any characteristic*, FOCS '08: Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science (Washington, DC, USA), IEEE Computer Society, 2008, pp. 146–155.

[KY89] Erich Kaltofen and Lakshman N. Yagati, *Improved sparse multivariate polynomial interpolation algorithms*, ISSAC '88: Proceedings of the International Symposium ISSAC'88 on Symbolic and Algebraic Computation (London, UK), Springer-Verlag, 1989, pp. 467–474.

[Lan02] Serge Lang, *Algebra*, Springer, January 2002.

[Len87] Hendrik W. Lenstra, *Factoring integers with elliptic curves*, Annals of Mathematics **126** (1987), 649–673.

[Ler96] Reynald Lercier, *Computing isogenies in GF(2,n)*, ANTS-II: Proceedings of the Second International Symposium on Algorithmic Number Theory (London, UK), Springer-Verlag, 1996, pp. 197–212.

[Ler97] ———, *Algorithmique des courbes elliptiques dans les corps finis*, Ph.D. thesis, LIX – CNRS, June 1997.

[LH10] Hai Liu and Paul Hudak, *An ode to arrows*, Practical Aspects of Declarative Languages (Berlin, Heidelberg) (Manuel Carro and Ricardo Peña, eds.), Lecture Notes in Computer Science, vol. 5937, Springer Berlin / Heidelberg, 2010, pp. 152–166–166.

[LMMS07] Xin Li, Marc Moreno Maza, and Éric Schost, *Fast arithmetic for triangular sets: from theory to practice*, ISSAC '07: Proceedings of the 2007 international symposium on Symbolic and algebraic computation (New York, NY, USA), ACM, 2007, pp. 269–276.

[LN96] Rudolf Lidl and Harald Niederreiter, *Finite fields (encyclopedia of mathematics and its applications)*, Cambridge University Press, October 1996.

[LS08] Reynald Lercier and Thomas Sirvent, *On Elkies subgroups of ℓ-torsion points in elliptic curves defined over a finite field*, Journal de théorie des nombres de Bordeaux **20** (2008), no. 3, 783–797.

[Mas03] Jim Massey, *Shift-register synthesis and BCH decoding*, IEEE Transactions on Information Theory **15** (2003), no. 1, 122–127.

[Mat08] Todd Mateer, *Fast Fourier transform algorithms with applications*, Ph.D. thesis, Clemson University, Clemson, SC, USA, 2008.

[McB03] Connor McBride, *Faking it—simulating dependent types in Haskell*, Journal of functional programming **12** (2003), no. 4-5, 375–392.

[Mil86] Victor S. Miller, *Use of elliptic curves in cryptography*, Lecture notes in computer sciences; 218 on Advances in cryptology—CRYPTO 85 (New York, NY, USA), Springer-Verlag New York, Inc., 1986, pp. 417–426.

[Mil96] James S. Milne, *Elliptic curves*, 1996.

[ML98] Saunders Mac Lane, *Categories for the working mathematician*, 2nd ed., Graduate Texts in Mathematics, Springer, September 1998.

[MMT01] Markus Maurer, Alfred Menezes, and Edlyn Teske, *Analysis of the GHS Weil descent attack on the ECDLP over characteristic two finite fields of composite degree*, INDOCRYPT '01: Proceedings of the Second International Conference on Cryptology in India (London, UK), Springer-Verlag, 2001, pp. 195–213.

[Moe76] Robert T. Moenck, *Another polynomial homomorphism*, Acta Informatica **6** (1976), no. 2, 153–169.

[Mon87] Peter L. Montgomery, *Speeding the pollard and elliptic curve methods of factorization*, Mathematics of Computation **48** (1987), no. 177, 243–264.

[Mor95] François Morain, *Calcul du nombre de points sur une courbe elliptique dans un corps fini: aspects algorithmiques*, Journal de Théorie des Nombres de Bordeaux **7** (1995), 255–282.

[Mor07] ———, *Implementing the asymptotically fast version of the elliptic curve primality proving algorithm*, Mathematics of Computation **76** (2007), 493–505.

[OWB71] G. M. Ostrowski, J. M. Wolin, and W. W. Borisow, *Über die Berechnung von Ableitungen*, Wissenschaftliche Zeitschrift der Technischen Hochschule für Chemie, Leuna-Merseburg **13** (1971), no. 4, 382–384.

[Pat01] Ross Paterson, *A new notation for arrows*, ICFP '01: International Conference on Functional Programming, 2001, pp. 229–240.

[Pie02] Benjamin C. Pierce, *Types and programming languages*, 1 ed., The MIT Press, February 2002.

[Pil90] Jonathan Pila, *Frobenius maps of Abelian varieties and finding roots of unity in finite fields*, Mathematics of Computation **55** (1990), no. 192, 745–763.

[Pit01] Andrew M. Pitts, *Categorical logic*, Handbook of Logic in Computer Science, Volume 5: Algebraic and Logical Structures (S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, eds.), Clarendon Press, Oxford, 2001.

[PS06] Cyril Pascal and Éric Schost, *Change of order for bivariate triangular sets*, ISSAC '06: Proceedings of the 2006 international symposium on Symbolic and algebraic computation (New York, NY, USA), ACM, 2006, pp. 277–284.

[Rou99] Fabrice Rouillier, *Solving Zero-Dimensional systems through the rational univariate representation*, Applicable Algebra in Engineering, Communication and Computing **9** (1999), no. 5, 433–461.

[RS06] Alexander Rostovtsev and Anton Stolbunov, *Public-key cryptosystem based on isogenies*, Cryptology ePrint Archive, Report 2006/145, April 2006.

[RV04] Alexandre Riazanov and Andrei Voronkov, *Efficient checking of term ordering constraints*, Automated Reasoning (David Basin and Michael Rusinowitch, eds.), Lecture Notes in Computer Science, vol. 3097, Springer, Berlin, Heidelberg, 2004, pp. 60–74.

[SAK+01] Kenneth W. Shum, Ilia Aleshnikov, P. Vijay Kumar, Hennning Stichtenoth, and Vinay Deolalikar, *A low-complexity algorithm for the construction of algebraic-geometric codes better than the Gilbert-Varshamov bound*, IEEE Transactions on Information Theory **47** (2001), no. 6, 2225–2241.

[Sat00] Takakazu Satoh, *The canonical lift of an ordinary elliptic curve over a finite field and its point counting*, Journal of the Ramanujan Mathematical Society **15** (2000), no. 4, 247–270.

[Sch77] Arnold Schönhage, *Schnelle Multiplikation von Polynomen über Körpern der Charakteristik 2*, Acta Informatica **7** (1977), no. 4, 395–398.

[Sch85] René Schoof, *Elliptic curves over finite fields and the computation of square roots mod* p, Mathematics of Computation **44** (1985), no. 170, 483–494.

[Sch95] ———, *Counting points on elliptic curves over finite fields*, Journal de Théorie des Nombres de Bordeaux **7** (1995), no. 1, 219–254. MR MR1413578

[Sch05] Éric Schost, *Multivariate power series multiplication*, ISSAC '05: Proceedings of the 2005 international symposium on Symbolic and algebraic computation (New York, NY, USA), ACM, 2005, pp. 293–300.

[Ser08] Igor Sergeev, *On the complexity of the gradient of a rational function*, Journal of Applied and Industrial Mathematics **2** (2008), no. 3, 385–396.

[Sho94] Victor Shoup, *Fast construction of irreducible polynomials over finite fields*, Journal of Symbolic Computation **17** (1994), no. 5, 371–391.

[Sho95] ———, *A new polynomial factorization algorithm and its implementation*, Journal of Symbolic Computation **20** (1995), no. 4, 363–397.

[Sho99] ———, *Efficient computation of minimal polynomials in algebraic extensions of finite fields*, ISSAC '99: Proceedings of the 1999 international symposium on Symbolic and algebraic computation (New York, NY, USA), ACM, 1999, pp. 53–58.

[Sho03] ———, *NTL: A library for doing number theory*, `http://www.shoup.net/ntl`, 2003.

[Sil86] Joseph H. Silverman, *The arithmetic of elliptic curves*, Graduate Texts in Mathematics, no. 106, Springer, 1986.

[Smi09] Benjamin Smith, *Isogenies and the discrete logarithm problem in Jacobians of genus 3 hyperelliptic curves*, Journal of Cryptology **22** (2009), no. 4, 505–529–529.

[SS71] Arnold Schönhage and Volker Strassen, *Schnelle Multiplikation großer Zahlen*, Computing **7** (1971), no. 3, 281–292.

[Str00] Christopher Strachey, *Fundamental concepts in programming languages*, Higher-Order and Symbolic Computation **13** (2000), no. 1, 11–49.

[Sut10] Andrew V. Sutherland, *Computing Hilbert class polynomials with the Chinese remainder theorem*, Mathematics of Computation (2010).

[Tes06] Edlyn Teske, *An elliptic curve trapdoor system*, Journal of Cryptology **19** (2006), no. 1, 115–133.

[TW95] Richard Taylor and Andrew Wiles, *Ring-theoretic properties of certain Hecke algebras*, The Annals of Mathematics **141** (1995), no. 3, 553+.

[Uma08] Christopher Umans, *Fast polynomial factorization and modular composition in small characteristic*, STOC '08: Proceedings of the 40th annual ACM symposium on Theory of computing (New York, NY, USA), ACM, 2008, pp. 481–490.

[Vél71] Jean Vélu, *Isogénies entre courbes elliptiques*, Comptes Rendus de l'Académie des Sciences de Paris **273** (1971), 238–241.

[Vol90] José F. Voloch, *Explicit* p*-descent for elliptic curves in characteristic* p, Compositio Mathematica **74** (1990), no. 3, 247–258.

[Vol99] Heribert Vollmer, *Introduction to circuit complexity: A uniform approach*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[vzGG99] Joachim von zur Gathen and Jürgen Gerhard, *Modern computer algebra*, Cambridge University Press, New York, NY, USA, 1999.

[vzGG02] ———, *Polynomial factorization over GF(2)*, Mathematics of Computation **71** (2002), no. 240, 1677–1698.

[vzGS92] Joachim von zur Gathen and Victor Shoup, *Computing Frobenius maps and factoring polynomials*, STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing (New York, NY, USA), ACM, 1992, pp. 97–105.

[WB89] Philip Wadler and Stephen Blott, *How to make ad-hoc polymorphism less ad-hoc*, Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages, ACM, January 1989, pp. 60–76.

[Wie86] Douglas H. Wiedemann, *Solving sparse linear equations over finite fields*, IEEE Transactions on Information Theory **32** (1986), no. 1, 54–62.

[Wil95] Andrew Wiles, *Modular elliptic curves and Fermat's last theorem*, The Annals of Mathematics **141** (1995), no. 3, 443+.

[WZ88] Yao Wang and Xuelong Zhu, *A fast algorithm for the Fourier transform over finite fields and its VLSI implementation*, IEEE Journal on Selected Areas in Communications **6** (1988), no. 3, 572–577.

[Yor09] Brent Yorgey, *Typeclassopedia*, The Monad.Reader (2009), no. 13, 17–68.

Abstract

In this thesis we apply techniques from computer algebra and language theory to speed up the elementary operations in some specific towers of finite fields. We apply our construction to the problem of computing isogenies between elliptic curves and obtain faster (both asymptotically and in practice) variants of Couveignes' algorithm.

The document is divided in four parts. In Part I we recall some basic notions from algebra and complexity theory. Part II deals with the transposition principle: in it we generalize ideas of Bostan, Schost and Lecerf, and show that it is possible to automatically transpose computer programs without losses in time complexity and with a small loss in space complexity. Part III combines the results on the transposition principle with classical techniques from elimination theory; we apply these ideas to obtain asymptotically optimal algorithms for the arithmetic of Artin-Schreier towers of finite fields. We also describe an implementations of these algorithms. Finally, in Part IV we use the previous results to speed up Couveignes' algorithm and compare the result with the other state of the art algorithms for isogeny computation. We also present a new generalization of Couveignes' algorithm that computes isogenies of unknown degree.

Résumé

Dans cette thèse nous appliquons des techniques provenant du calcul formel et de la théorie des langages afin d'améliorer les opérations élémentaires dans certaines tours de corps finis. Nous appliquons notre construction au problème du calcul d'isogénies entre courbes elliptiques et obtenons une variante plus rapide (à la fois en théorie et en pratique) de l'algorithme de Couveignes.

Le document est divisé en quatre parties. Dans la partie I nous faisons des rappels d'algèbre et de théorie de la complexité. La partie II traite du principe de transposition : nous généralisons des idées de Bostan, Schost et Lecerf et nous montrons qu'il est possible de transposer automatiquement des programmes sans pertes en complexité-temps et avec une petite perte en complexité-espace. La partie III combine les résultats sur le principe de transposition avec des techniques classiques en théorie de l'élimination ; nous appliquons ces idées pour obtenir des algorithmes asymptotiquement optimaux pour l'arithmétique des tours d'Artin-Schreier de corps finis. Nous décrivons aussi une implantation de ces algorithmes. Enfin, dans la partie IV nous utilisons les résultats précédents afin d'accélérer l'algorithme de Couveignes et de comparer le résultat avec les autres algorithmes pour le calcul d'isogénies qui font l'état de l'art. Nous présentons aussi une nouvelle généralisation de l'algorithme de Couveignes qui calcule des isogénies de degré inconnu.