Efficient Multivariate Factorization Over Finite Fields

Laurent Bernardin[1] and Michael B. Monagan[2]

[1] Institut für Wissenschaftliches Rechnen
ETH Zürich, Switzerland
bernardin@inf.ethz.ch
[2] Center for Experimental and Constructive Mathematics
Department of Mathematics and Statistics
Simon Fraser University, Canada
monagan@cecm.sfu.ca

Abstract. We describe the Maple [23] implementation of multivariate factorization over general finite fields. Our first implementation is available in Maple V Release 3. We give selected details of the algorithms and show several ideas that were used to improve its efficiency. Most of the improvements presented here are incorporated in Maple V Release 4.
In particular, we show that we needed a general tool for implementing computations in $GF(p^k)[x_1, x_2, \ldots, x_v]$. We also needed an efficient implementation of our algorithms in $\mathbb{Z}_p[y][x]$ because any multivariate factorization may depend on several bivariate factorizations.
The efficiency of our implementation is illustrated by the ability to factor bivariate polynomials with over a million monomials over a small prime field.

1 Introduction

This paper describes a state-of-the-art implementation of algorithms for factoring multivariate polynomials over finite fields, based on ideas presented in [21,8,1,2].

Our implementation includes many refinements that make these algorithms efficient in practice. The motivation for improving its efficiency came from factorization problems that are part of applications in computing resultants, Galois groups [18] and Gröbner bases [4] as well as algebraic geometry [15].

A Maple [23] implementation has also been presented in [18]. Another implementation of multivariate factorization over finite fields has been done in Axiom [7]. Our implementation makes it possible to factor polynomials of much larger degree. This was achieved by identifying the performance bottlenecks and by making improvements to all the steps of the factorization; Square-free factorization, Hensel lifting and combination of extraneous factors.

From the systems design viewpoint, it was necessary to code three separate implementations of the algorithms, first a general purpose implementation,

using the Domains (formerly Gauss) package [13], second a special purpose implementation for bivariate polynomials, using a recursive dense data structure and third, an implementation using Maple's builtin sparse distributed general purpose data structure which is called the "sum-of-products" data structure.

This paper is organized as follows: Section 2 presents the basic algorithms and details of their implementation. Section 3 presents the data structures that are used. Section 4 contains the conclusion as well as remarks on future work.

2 Algorithms and Implementation Details

We factor multivariate polynomials by using a modular algorithm. Given $h \in \mathbb{F}_q[x_1, \ldots, x_n]$, an evaluation homomorphism $\phi_{x_k=\alpha_k}$, $\alpha_{2..n} \in \mathbb{F}_q$ reduces the problem to factoring a univariate polynomial. The univariate image factors are then lifted one variable at a time using Hensel lifting to reconstruct the true multivariate factors.

In the following we will assume that the polynomial to factor is primitive. This condition can be achieved by removing the content in each variable and recursively factoring the resulting polynomials in one less variable. For polynomials in more than two variables, this is a potentially expensive step. Also, sparse polynomials may become dense after the content removal [8]. Nonetheless, rendering the input polynomial primitive enables us to apply optimizations to the factorization process, which have proven very effective in practice.

The homomorphism diagram in table 1 outlines the multivariate factorization process. We will examine each step of the process and give details on how to efficiently implement the algorithms.

2.1 Squarefree Factorization

The square-free factorization removes multiple factors from the input polynomial thus reducing the degree of the polynomial. In our implementation, we use a deterministic algorithm given in [2] which is derived from Yun's square-free decomposition algorithm for characteristic zero. The cost of this step is roughly the same as computing the greatest common divisor of the input polynomial and its derivative.

Computing this GCD can be the performance bottleneck for the entire factorization, especially over finite fields with a small number of elements where it is difficult to apply a modular GCD algorithm because we might not be able to find a good evaluation homomorphism. For multivariate polynomials, Maple tries to use the Extended Hensel GCD algorithm [6]. For bivariate polynomials we use an interpolation algorithm if the coefficient field contains enough evaluation points. In both cases we fall back to the subresultant GCD algorithm if the modular method fails. For polynomials in more than two variables, it might be better to choose enough evaluation values from an extension field rather than using the subresultant GCD algorithm but this has not been implemented.

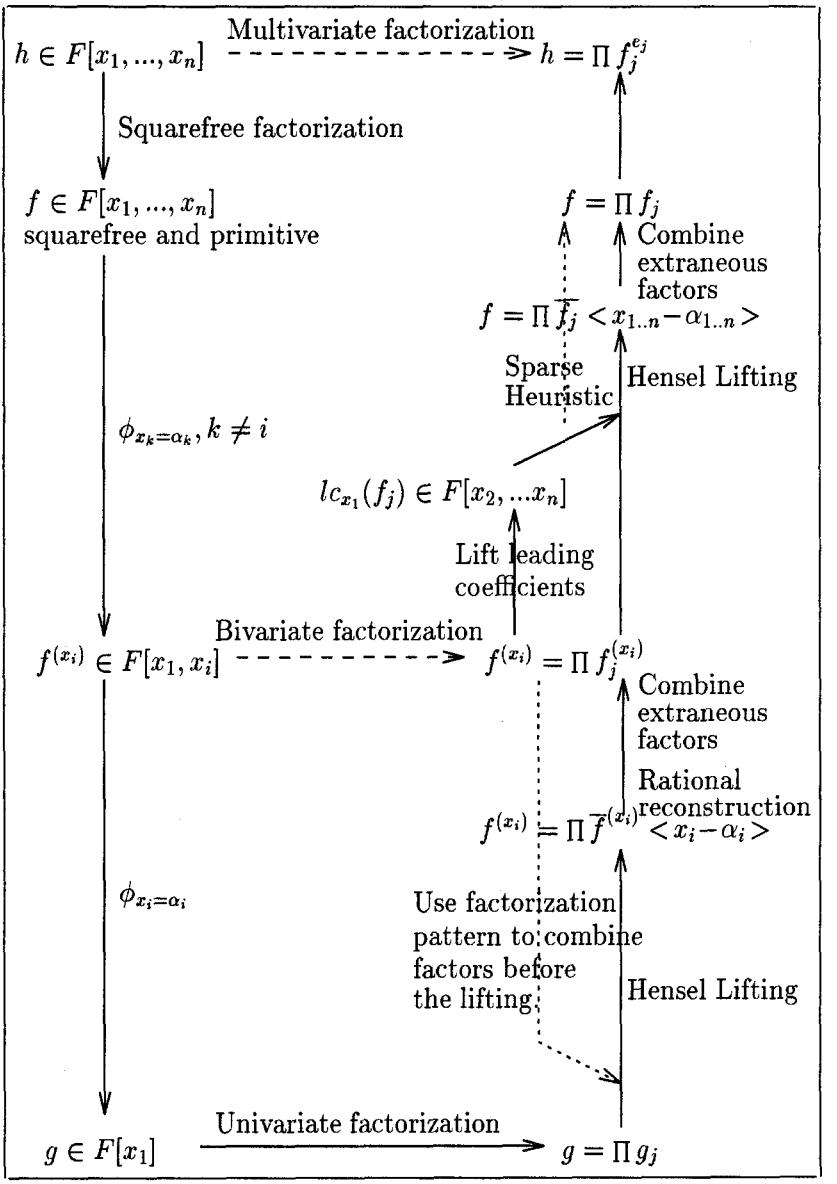

Table 1. Factorization Process

2.2 Choosing Evaluation Points and Field Extensions

We will use an evaluation homomorphism to reduce the problem of factoring a multivariate polynomial to factoring a univariate one. Not all possible evaluations will work so we might have to try several ones before successfully factoring the input polynomial. Most of the time however, we can detect unlucky evaluations before starting the costly Hensel lifting process. In particular, we can discard evaluations that produce an image polynomial with degree less than the degree of the original polynomial in the main variable (x_1). Likewise, we discard evaluations that produce non-square free factors as we know at this stage that the multivariate polynomial is square-free.

The current implementation tries to find $4v$ valid evaluation points for a polynomial in v variables in order to perform a degree analysis on the factorization patterns of the univariate images. Information gathered here can be used to deduce that the polynomial is irreducible even if none of the image polynomials is irreducible by narrowing the set of possible factorization patterns of the multivariate polynomial. Even if we can't conclude that the polynomial is irreducible, we can often use this information after the Hensel lifting to reduce the number of combinations of lifted factors that we have to do in order to find true factors. Our current implementation does not take advantage of this possibility. The potential efficiency gain seems marginal as our heuristics for combining lifted factors are very efficient (see section 2.4).

If the coefficient field does not contain any valid evaluation points we have to choose points from an extension field of $\mathbb{F}_q$. We know from [20] that *if* we would choose a field extension of prime degree larger than the total degree d of the polynomial to be factored, we are guaranteed that the polynomial has the same factorization pattern over the extension field as over the ground field $\mathbb{F}_q$.

In practice, however, it would be too expensive to compute with field extensions of that size as the cost increases quadratically with d using our general representation of finite fields. Instead we choose a small extension, knowing that we might need to combine lifted factors in order to get true factors over the ground field. Combining factors might have exponential cost but in practice we have found that extraneous factors do not appear too often at this point. Note that by first considering combinations of factors that have coefficients not in the ground field we can reduce the number of combinations that have to be considered. On the other hand it does not make sense to choose the field extension too small as we incur the risk of having to extend it again. Even if a very small extension would contain enough valid points, working over a slightly larger extension will not degrade the performance significantly. For this reason the current implementation uses a field extension of degree 3 even when an extension of degree 2 would be sufficient in most cases. If the first field extension is not sufficiently large we do not build a tower of extensions but instead extend the ground field again by choosing the smallest prime degree larger than the previous extension. Choosing a prime degree for the field extensions guarantees us that the polynomial is irreducible over the ground field if it is irreducible over the extension field, which is not guaranteed for arbitrary extensions [20].

2.3 Hensel Lifting

After evaluating the multivariate polynomial using the homomorphism $\phi_{x_k=\alpha_k}$, we factor the univariate image using the Cantor-Zassenhaus algorithm [3] for small fields or Shoup's algorithm [16] for large ones. Now we seek to reconstruct the multivariate factors from the univariate ones.

For polynomials with more than two variables, we use the Hensel lifting algorithm described in [6] to lift the bivariate factors to multivariate ones. For non-monic polynomials this means that we have to know the multivariate leading coefficients of the factors in the main variable before the lifting. We solve this "leading coefficient problem" with Kaltofen's approach [8]: We first compute bivariate factorizations in the main variable and each of the remaining variables. The leading coefficients that this step produces are polynomials in one variable which we can lift recursively with respect to the leading coefficient of the input polynomial in the main variable. This means that the first step of lifting image factors with respect to a multivariate polynomial involves recursively lifting images of the leading coefficients of the factors with respect to a polynomial in one variable less than the input polynomial.

The multivariate leading coefficients of the factors that we determined in the previous step are attached to the bivariate factors before lifting the remaining variables.

For bivariate polynomials we use a fast dense lifting algorithm [1] which, for dense polynomials, is one order of magnitude faster than previous algorithms. This algorithm solves the leading coefficient problem using the post-lifting leading coefficient computation proposed by Kaltofen [8] which uses Padé approximations to compute the univariate leading coefficients of the bivariate factors.

We also implemented a quadratic Hensel lifting algorithm. Although it is asymptotically faster, we found that the break even point with linear lifting is too high for it to be practical. For the largest example we tried, a dense, bivariate polynomial with degree 1000 in both variables, the quadratic algorithm still took twice as long as the linear one.

Note that the lifting algorithms that we implemented only work for evaluations at zero. If we have to use a different evaluation point, which is the case most of the time, we first translate the polynomial by this value, lift and undo the translation after the lifting. Translating a polynomial of degree n by a constant is fast, requiring $O(n)$ coefficient multiplications and $O(n^2)$ coefficient additions[11]. However, the disadvantage of this translation is that a sparse polynomial will be made dense. We will address this issue in section 2.5.

2.4 Combining Extraneous Factors

Applying an evaluation homomorphism to a multivariate polynomial may generate extraneous factors. These have to be recombined after the lifting in order to recover the true factors. By its nature, this step has exponential complexity. For more than two variables, keeping the running-time polynomial is possible by noticing that there exists a set of evaluations that will produce an evaluated

polynomial with the same factorization pattern [20,8], thus avoiding the combination step. For bivariate polynomials polynomial-time algorithms exist [21] but in practice it is better to use a fast lifting algorithm and accept the possibility of extraneous factors. In practice the combinatorial step will not be a problem except for pathological cases.

So we have to check for divisibility of the input polynomial by each combination of the lifted factors. Because of what was said above, doing this after lifting one variable will produce the correct factorization pattern in most cases. As seen in section 2.3, in order to compute the multivariate leading coefficients in advance, we have to factor multiple bivariate polynomials which are derived from the multivariate input polynomial by evaluating all but two variables. We use the combination pattern of one bivariate factorization to combine the univariate image factors before the lifting with respect to the other bivariate polynomials. In practice this means that we have to execute the combination step only for the first bivariate factorization.

Because this step involves an exponential number of combinations it is crucial for these to be processed as fast as possible.

Combinations will be necessary when lifting from univariate to bivariate most of the time but rarely occur when lifting variables of higher order. For this reason we discuss the following heuristics for bivariate polynomials only.

If the input polynomial is monic we first try to divide only the trailing coefficients in the main variable which are univariate polynomials in the lifted variable. If this division is successful we do a full multivariate trial division.

In the non-monic case, we have to reconstruct the leading coefficients of the lifted factors as seen in section 2.3. We have to perform this for each combination of factors. We can compute a multiple of the leading coefficient by using any single univariate coefficient of the lifted factor. So, we first try to reconstruct the leading coefficient by using only the trailing coefficients of each combination of lifted factors. If this reconstruction fails, we discard the combination. If we succeed in computing a leading coefficient (or multiple thereof), we check whether the (univariate) leading coefficient of the input polynomial divides the product of the partially reconstructed leading coefficients of each combination of lifted factors.

In practice these heuristics always detected bad combinations without any need for a full multivariate trial division.

As pointed out by an anonymous referee, the number of combinations to try can be reduced by fixing one of the lifted factors. This factor can be ignored during the combinations and it is then known to either be a true factor, or belong to a combination together with those factors, that could not be resolved by the combination step. We do not currently implement this improvement.

2.5 Sparse Heuristic

In practice, the more indeterminates a polynomial involves, the sparser it will be. However, using our Hensel Lifting algorithm on a sparse polynomial might turn it into a dense problem. This happens if we have to use non-zero evaluation

points because the necessary translation by a constant turns a sparse polynomial into one with many more monomials.

One solution would be to use a sparse Hensel Lifting algorithm [19,22,8,10] but we use a different approach which proved to be efficient for practical problems. For polynomials with more than 2 variables, we first call a heuristic factorization algorithm that will try to "guess" coefficients of the multivariate factors given the bivariate factorization and the already computed multivariate leading coefficients [12]. This heuristic will succeed if the factors are sufficiently sparse. If they are not, we apply the dense Hensel Lifting algorithm.

The equations are set up in the following way. For each factor $f_i(x, y)$ of the bivariate factorization over the coefficient field $\mathbb{F}$:

$$f_i = c_0 a_0(y) + c_1 a_1(y)x + \ldots c_n a_n(y)x^n$$

where $c_i \in \mathbb{F}$ and $a_i(y) \in \mathbb{F}[y]$, $a_i(y)$ monic, we replace the c_i by symbols Z_i which represent the coefficients in the remaining variables of the multivariate factors.

$$\overline{f}_i = Z_i a_0(y) + Z_1 a_1(y)x + \ldots Z_n a_n(y)x^n$$

Now we multiply the f_i and subtract the result from the multivariate input polynomial which we are factoring. The coefficients of each monomial in x and y are the equations we are left to solve. In order to solve these, we first put in the leading coefficients that we have already computed. Next we search the set for linear equations, solve those that we find, plug the solutions back into the whole set and iterate this process until the set of equations does not change anymore. If the set is empty at this point, we succeeded in factoring the multivariate polynomial.

Our implementation uses the leading coefficients as a starting point to solving the arising system of equations. If knowing the leading coefficients is not enough to make the heuristic succeed, we could precompute the trailing coefficients using the same techniques as for the leading coefficients and apply the heuristic again using that additional piece of information. This improvement is currently not implemented.

Note that this heuristic can only succeed if the bivariate factorization does not have any spurious factors. Fortunately, this has been shown to be very probable [20].

3 Data Structures

3.1 Bivariate Polynomials

Kaltofen's method of determining the true leading coefficients of the final factors relies on the factorization of $v - 1$ bivariate polynomials (v being the number of indeterminates in the polynomial to factor). In addition to this, in practice, factoring bivariate polynomials is a very common case. Hence we need an efficient way of computing with bivariate polynomials.

Maple [23] already has an efficient data structure for dense univariate polynomials over prime fields, called the modp1 representation [14]. If the characteristic is sufficiently small, a polynomial is represented as an array of machine integers, otherwise as a Maple list of arbitrary precision integers. The basic arithmetic operations have been coded in "C" and, for small characteristics, they run in place without function call overhead for the coefficient arithmetic.

On top of this data structure, we have implemented dense bivariate polynomials over prime fields as Maple lists of modp1 polynomials representing the univariate coefficients of the bivariate polynomial in the main variable. Computing with this new data structure (called the modp2 representation) proves to be quite efficient. We obtained a speedup of a factor of 5-60 compared to Maple's general purpose "sum of products" representation for multivariate polynomials.

Internally a list in Maple is a (non-mutable) array of pointers to the list entries. So the modp2 representation is a recursive dense data structure [17]. We have to store all the coefficients of the polynomial in the main variable, even if they are zero. Still, the space taken by the univariate coefficients depends on their degree and the representation of a sparse polynomial of degree n in both variables needs far less space than d^2 field elements.

We have implemented all basic polynomial operations for the modp2 representation: $*$, $+$, $-$, pseudo-quotient and pseudo-remainder, GCD, evaluation, differentiation, translation by a constant, swapping variables. Multiplication uses an implementation of Karatsuba's algorithm. The break-even with classical multiplication depends on the size of the univariate coefficients. It varies from degree 40 in the main variable (with univariate coefficients of degree 100 and less) to degree 2 in the main variable (with univariate coefficients of degree 1000 and larger). Note that Karatsuba multiplication is not yet implemented for the modp1 data structure.

The GCD algorithm being used is a modular interpolation-type algorithm. It falls back to the subresultant GCD algorithm if the coefficient field is too small to provide enough evaluation values [6].

3.2 Multivariate Polynomials

For polynomials with more than two variables we use Maple's builtin "sum-of-products" data structure if we are working over a prime field. This is a sparse distributed data structure. Arithmetic is reasonably efficient and it allows us to tackle sparse polynomials in many variables.

Over general Galois fields, we use the Domains (formerly Gauss) package [13]. This package provides a way of setting up domains of computation, each with their unique set of operations. The Domains package provides a sparse distributed as well as a dense recursive representation for multivariate polynomials. We made the choice of implementing our factorization algorithms using the recursive dense representation partly because the nature of the algorithms favor a recursive data structure and partly because for polynomials which are not too sparse, the dense representation offers better performance.

For sparse polynomials in many variables, the data structure of choice should be black boxes [10] but we have not implemented this. A reasonably efficient black box implementation would require a complete system design which allows for dynamic creation and compilation of black box programs.

3.3 Computing with Field Extensions

As mentioned previously, if the ground field $GF(q)$ is small, it may not be possible to find enough sets of evaluation points. When this happens it is necessary to extend the field to $GF(q^k)$ for a reasonable choice of k. Consider for example the following polynomial over $\mathbb{Z}_2$:

$$f = x^2y^3 + xy + xy^2 + 1$$

There are only 2 possible substitution values for y: 0 and 1. Substituting 0 results in a univariate image whose degree is smaller than the degree of f in x. Substituting 1 results in a univariate image which is not square-free while f itself is square-free. Both evaluation values are thus invalid.

In our implementation we use $k = 3, 5, 7, 11, \ldots$ until enough evaluation points can be found. To compute in $GF(p^k)$ our implementation again makes use of the modp1 representation for efficient univariate polynomial arithmetic in $\mathbb{Z}_p[x]$ to implement the field operations in $GF(p^k) \simeq \mathbb{Z}_p[x]/m(x)$ with $m(x)$ an irreducible polynomial of degree k. Thus each field operation requires univariate polynomial operations, which are done using compiled machine code.

For our example from above we would choose $k = 3$ and

$$m(x) = x^3 + x + 1$$

Note that $m(x)$ is irreducible over $\mathbb{Z}_2$. Let α denote a root of $m(x)$. Substituting α for y leads to the univariate factorization

$$(\alpha + 1)\left(x + \alpha^2 + 1\right)\left(x + \alpha^2 + \alpha + 1\right)$$

These univariate images can now be lifted in order to recover the multivariate factorization:

$$(xy + 1)\left(xy^2 + 1\right)$$

Since we use a polynomial representation, there is no limit to the size of k or p^k, in fact the additional cost of moving from, e.g., $k = 3$ to $k = 5$ is low. However, in terms of computing in $GF(p^k)[x_1, x_2, \ldots, x_v]$ the overhead is noticeable. For small p^k, it would be more efficient to implement the field arithmetic using the Zech-Jacobi representation which uses the fact that $GF(p^k)$ is isomorphic to $\{0, e, e^2, e^3, ..., e^{p^k-1}\}$ for some primitive element $e \in GF(p^k)$. For this one would have to write a package similar to the modp1 package which provides efficient arithmetic in $GF(p^k)$ and then implement the factorization algorithms on top of that package. However, we have not done this and consequently there is a noticeable performance loss when going from $\mathbb{Z}_p$ to $GF(p^k)$ for small p^k.

4 Experimental Results

In this section, we present a selection of examples that we used to test our implementation. Times are in CPU seconds on a Sparcstation 20/51. The example polynomials used in this section are available from the authors on request or from `http://www.inf.ethz.ch/personal/bernardi/factor/`. We describe them in detail because we believe these are typical examples of such factorizations.

Note that we failed to factor any of the polynomials from this section with Axiom 2.0 [7] which incorporates another implementation of multivariate factorization over finite fields.

Example 1: Our first example is a random dense polynomial in two variables with degree 100 in both variables. The leading coefficient in each of the two variables is a univariate polynomial of degree 20. The polynomial is primitive and square-free and has two factors of equal degrees over $\mathbb{Z}_7$. The following table shows the time spent in each of the factorization steps.

	Time	% of total time
Square-free Factorization	68s	27%
Evaluations and Univariate Factorization	4s	2%
Hensel Lifting	136s	54%
Discard Bad Combinations	22s	9%
Trial Divisions	21s	8%
Total	251s	100%

Notes: Of the 7 elements of $\mathbb{Z}_7$ only one is found valid for evaluating this polynomial. The corresponding univariate image splits into 9 factors which have to be lifted to degree 121. After that, 60 bad combinations are discarded before finding a combination of 3 of the lifted factors which lead to a true one. The algorithm then goes on to discard 20 more bad combinations before it concludes that the remaining 6 lifted factors all correspond to the same true factor because all combinations of 3 and less factors have been tried.

Example 2: The next example is a random dense polynomial in two variables with degree 300 in both variables, primitive and square-free. The polynomial is monic which means that we don't have to worry about the leading coefficient problem.

	Time	% of total time
Square-free Factorization	103s	16%
Evaluations and Univariate Factorization	147s	23%
Hensel Lifting	364s	59%
Discard Bad Combinations	2s	1%
Trial Divisions	0s	0%
Total	616s	100%

Notes: All 11 elements of $\mathbb{Z}_{11}$ are tried and the best evaluation produces 3 univariate factors which are lifted to degree 301. After verifying that none of the lifted factors divides the input polynomial, the algorithm concludes that the

polynomial is irreducible. This verification is done solely by the heuristics of the combination step and does not need any multivariate trial division.

Example 3: The next example is a primitive and square-free polynomial in three variables x, y, z with degrees $(35, 15, 10)$. The sparsity quotient

$$\frac{\text{Number of possible terms}}{\text{Number of actual terms}}$$

of this polynomial is 30, which means that it is rather sparse. Over $\mathbb{Z}_5$ the polynomial splits into two factors of degrees $(15, 5, 5)$ and $(20, 10, 5)$. This polynomial is part of solving a classification problem in algebraic geometry [5].

	Time	% of total time
Square-free Factorization	1s	11%
Evaluations and Univariate Factorization	3s	33%
Leading Coefficient Determination	3s	33%
Sparse Heuristic	2s	23%
Total	9s	100%

Notes: z is chosen as main variable in order to reduce the degree of the univariate polynomial and thus the possibility of spurious factors. The multivariate leading coefficients are computed from the factorization of the bivariate polynomials obtained by evaluating the variable x, then the variable y. Using the leading coefficients, the sparse heuristic succeeds in producing the complete multivariate factorization without the need of any further lifting.

Example 4: The next example is a small bivariate polynomial of degree $(7, 5)$. It is primitive and square-free and splits into two factors of degrees $(4, 3)$ and $(3, 2)$ over $\mathbb{Z}_2$. However, because there are no valid evaluation points to be found in the ground field, a field extension is necessary.

	Time	% of total time
Exhaust Values from the Ground Field	1s	2%
Search Evaluation Points in $GF(2^3)$	7s	13%
Hensel Lifting	15s	29%
Combinations and Trial Divisions	29s	56%
Total	52s	100%

Notes: The size of the example above is much smaller than that of the previous ones. This is partly because the Domains code is slower due to the overhead of its genericity but also because we have not implemented all of the performance improvements in that version of the code. In particular the heuristics for detecting bad combinations of lifted factors are missing.

Example 5: Following is an example of a bivariate polynomial over $\mathbb{Z}_{13}$ of degree 1000 in both variables. It is dense and its expanded form would have more than one million terms. It splits into two monic factors of equal degree.

	Time	% of total time
Square-free Factorization	1317s	1%
Evaluations and Univariate Factorization	5528s	4%
Hensel Lifting	82128s	60%
Discard Bad Combinations	169s	0%
Trial Division	47752s	35%
Total	136894s	100%

Notes: The square-free factorization quickly realizes that the GCD of the polynomial and its derivative is one because the modular GCD algorithm succeeds in finding a suitable evaluation value. If the modular algorithm would fail, the subresultant algorithm would take a long time to compute that GCD. Thirteen univariate factorizations are then tried, taking from 5 to 20 minutes each. The best evaluation leads to 9 univariate image factors which are then lifted to degree 1001. After the lifting, 183 bad combinations of the lifted factors are tried and discarded by the heuristic. The trial division, which accounts for more than one third of the total factorization time, amounts to dividing a polynomial of degree 1000 in both variables by a polynomial of degree 500 in both variables. The factorization of this huge polynomial needed a total of 112MB of main memory.

Example 6: The last example exercises the combination of extraneous factors. We use a bivariate Swinerton-Dyer polynomial [9] of degrees 32 and 16 with coefficients from $\mathbb{Z}_{17}$. The polynomial is irreducible but any evaluation to a univariate polynomial splits it into 16 factors.

	Time	% of total time
Square-free Factorization	1s	0%
Evaluations and Univariate Factorization	1s	0%
Hensel Lifting	2s	0%
Discard Bad Combinations	1250s	100%
Total	1254s	100%

Notes: After the lifting, 16 factors have to be combined in groups of 1, 2, 3, 4, 5, 6, 7 and 8 before it can be concluded that the bivariate polynomial is irreducible. Thus, a total of 39202 bad combinations have to be discarded.

5 Code Outline

Our Maple implementation consists of three separate versions of the algorithms described herein.

The most general version of the factorizer has been implemented using the Domains (formerly Gauss) package [13]. The Domains package provides generic coding, in that the same code will work for different implementations of multivariate polynomials and for different implementations of finite fields. Therefore, this version of the factorizer can be used over arbitrary finite fields. This in-

cludes finite fields given by a tower of algebraic extensions over a ground field. The Domains code is used for polynomials over a general Galois field $GF(p^k)$.

The standard Maple factorization code using the "sum of products" data structure is about 5 times faster than the Domains version but works only for multivariate polynomials over prime fields. If we have to use evaluation values from an extension field, the Domains code is used instead.

Finally, the modp2 version, using the dense modp2 data structure for bivariate polynomials over prime fields gives another factor of 5 in efficiency. Field extensions are handled by code written using the Domains package but tuned for bivariate polynomials.

All these routines are wrapped by the top-level command "Factor". The only exception is when the coefficient field is given as a tower of extensions and the Domains package has to be used directly.

6 Conclusions and Future Work

The conclusions of this work are first, that in order to factor multivariate polynomials over finite fields, we need an efficient implementation of factorization of bivariate polynomials over finite fields. To obtain a good implementation, it is sufficient to use a recursive dense representation, $\mathbb{Z}_p[y][x]$ where the implementation for $\mathbb{Z}_p[y]$ is an array of machine integers and operations in $\mathbb{Z}_p[y]$ are done in place with no function call overhead for the coefficient arithmetic. Second, a reasonably efficient implementation for $GF(p^k)[x]$ can be obtained similarly, by representing the coefficients in $GF(p^k)$ as polynomials in $\mathbb{Z}_p[z]$. But, for small p^k, a better implementation would encode elements of $GF(p^k)$ as a single machine integer, and represent elements of $GF(p^k)[x]$ as an array of machine integers. However, we have not implemented this and it requires a substantial amount of work to do so.

In order to make the factorization algorithms practical for high degree polynomials, improvements were necessary at all the major steps of the algorithms. Our implementation includes new techniques for the square-free factorization, Hensel lifting and extraneous factor combination steps.

Another area for future work is the data structure used for representing polynomials in many variables. For polynomials over prime fields we use a sparse distributed data structure but over general Galois fields, the representation is recursive dense. Better would be to implement the factorization algorithm for polynomials in many variables using the black-box representation [10]. This algorithm could rely on our fast bivariate factorization algorithm.

References

1. BERNARDIN, L. Fast dense Hensel lifting. manuscript, ETH Zürich, 1995. Available via `http://www.inf.ethz.ch/personal/bernardi`.
2. BERNARDIN, L. On square-free factorization of multivariate polynomials over a finite field. *Theoretical Computer Science 187* (1997). to appear.

3. CANTOR, D. G., AND ZASSENHAUS, H. A new algorithm for factoring polynomials over finite fields. *Mathematics of Computation 36*, 154 (1981), 587–592.
4. CZAPOR, S. R. Solving algebraic equations:combining Buchberger's algorithm with multivariate factorization. *Journal of Symbolic Computation 7*, 1 (January 1989), 49–54.
5. DA ROSA, R. M. Private communication, February 1996.
6. GEDDES, K. O., CZAPOR, S. R., AND LABAHN, G. *Algorithms for Computer Algebra.* Kluwer Academic Publishers, Boston, 1992.
7. JENKS, R., AND SUTOR, R. *AXIOM: The Scientific Computation System.* Springer Verlag, 1992.
8. KALTOFEN, E. Sparse Hensel lifting. In *Proceedings of Eurocal '85, Vol. II* (1985), B. F. Caviness, Ed., vol. 204 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 4–17.
9. KALTOFEN, E., MUSSER, D. R., AND SAUNDERS, B. D. A generalized class of polynomials that are hard to factor. *SIAM Journal on Computing 12*, 3 (1983), 473–485.
10. KALTOFEN, E., AND TRAGER, B. M. Computing with polynomials given by black boxes for their evaluations: Greatest common divisors, factorization, separation of numerators and denominators. *Journal of Symbolic Computation 9*, 3 (March 1990), 300–320.
11. KNUTH, D. E. *Seminumerical Algorithms*, vol. 2 of *The Art of Computer Programming.* Addison Wesley, 1981.
12. LUCKS, M. A fast implementation of polynomial factorization. In *SYMSAC '86: Proceedings of the 1986 ACM Symposium on Symbolic and Algebraic Computation* (1986), pp. 228–232.
13. MONAGAN, M. B. Gauss: A parameterized domain of computation system with support for signature functions. In *Proceedings of DISCO '93* (1993), vol. 722 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 81–94.
14. MONAGAN, M. B. In-place arithmetic for polynomials over Z_n. In *Proceedings of DISCO '92* (1993), vol. 721 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 22–34.
15. POPP, H. *Moduli Theory and Classification Theory of Algebraic Varieties*, vol. 620 of *Lecture Notes in Mathematics.* Springer-Verlag, 1977.
16. SHOUP, V. A new polynomial factorization algorithm and its implementation. *Journal of Symbolic Computation 20*, 4 (1995), 363–397.
17. STOUTMYER, D. R. Which polynomial representation is best? In *Proceedings of the 1984 MACSYMA User's Conference* (1984), V. E. Golden, Ed., pp. 221–243.
18. SWANSON, S. L. *On the Factorization of Multivariate Polynomials over Finite Fields.* PhD thesis, Purdue University, 1993.
19. VON ZUR GATHEN, J. Factoring sparse multivariate polynomials. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science* (1983), pp. 172–179.
20. VON ZUR GATHEN, J. Irreducibility of multivariate polynomials. *Journal of Computer and System Sciences 31* (1985), 225–264.
21. VON ZUR GATHEN, J., AND KALTOFEN, E. Polynomial-time factorization of multivariate polynomials over finite fields. In *Proceedings of ICALP '83* (1983), vol. 154 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 250–262.
22. VON ZUR GATHEN, J., AND KALTOFEN, E. Factoring sparse multivariate polynomials. *Journal of Computer and System Sciences 31* (1985), 265–287.
23. WATERLOO MAPLE INC. *Maple V learning guide.* Springer-Verlag, 1996.