Linear Hensel lifting for $\mathbb{Z}_p[x, y]$ for n factors with cubic cost

Michael Monagan
Department of Mathematics
Simon Fraser University
Burnaby, BC, Canada
mmonagan@sfu.ca

Garrett Paluck
Department of Mathematics
Simon Fraser University
Burnaby, BC, Canada
gpaluck@sfu.ca

ABSTRACT

We present a new algorithm for performing linear Hensel lifting on bivariate polynomials over the finite field $\mathbb{Z}_p$ for some prime p. Our algorithm lifts n monic, univariate polynomials to recover the factors of a polynomial $A(x, y) \in \mathbb{Z}_p[x, y]$ which is monic in x, and bounded by degrees $d_x = \deg(A, x)$ and $d_y = \deg(A, y)$. Our algorithm improves upon Bernardin's algorithm in [1] and reduces the number of arithmetic operations in $\mathbb{Z}_p$ from $O(n\, d_x^2\, d_y^2)$ to $O(d_x^2\, d_y + d_x\, d_y^2)$ for $p \geq d_x$. Experimental results in C verify that our algorithm compares favorably with Bernardin's for large degree polynomials. Moreover, we've implemented a Quadratic Hensel lifting algorithm in Magma to show that our cubic Linear Hensel lifting algorithm outperforms Magma's Quadratic Hensel lifting for a wide range of input sizes.

KEYWORDS

Hensel lifting, modular methods, bivariate polynomial factorization

1 INTRODUCTION

Hensel lifting is one of the main tools used to factor polynomials. It was first used by Zassenhaus in [25] to factor polynomials in $\mathbb{Z}[x]$. Musser in [18] and Wang and Rothchild in [24] subsequently developed multivariate Hensel lifting (MHL) to factor polynomials in $\mathbb{Z}[x_1, ..., x_n]$ in [17]. For a complete description of MHL we refer the reader to Ch. 6 of [7]. As far as we know, all Computer Algebra Systems use Hensel lifting to factor polynomials. However, factorization methods that do not use Hensel lifting are known, for example, Gao in [5] developed a method based on partial derivatives and linear algebra.

Hensel lifting can also be used for polynomial GCD computation. In [17] Moses and Yun applied MHL to compute the greatest common divisor of two polynomials in $\mathbb{Z}[x_1, ..., x_n]$. They called their algorithm the EZ-GCD algorithm. In [23] Wang improved the EZ-GCD algorithm for sparse polynomials. Wang's GCD algorithm is implemented in Macsyma, Maple and Magma.

In this paper, we are interested in the cost of Hensel lifting when it is used to factor bivariate polynomials in $\mathbb{Z}_p[x, y]$ for a prime p. Our work is motivated by the parallel MHL algorithms of Monagan and Tuncer [16] and Chen and Monagan [4] which reduce Hensel lifting in $\mathbb{Z}[x_1, \ldots, x_n]$ to many bivariate Hensel lifts in $\mathbb{Z}_p[x, y]$ and which use sparse interpolation. Other works that reduce multivariate polynomial factorization to bivariate polynomial factorization include Kaltofen [10] and Lecerf [13].

We begin with a description of Hensel lifting in $R[x]$ where $R = \mathbb{Z}_p[y]$. Our description follows von zur Gathen and Gerhard [6]. Let A be a polynomial in $R[x]$ which has no repeated factors. Let m be a polynomial in R with $\deg(m, y) \geq 1$. We require the modulus m to be relatively prime to $\mathrm{lc}(A)$, the leading coefficient of A. Suppose we are given polynomials $f_i^{(0)}$ for $1 \leq i \leq n$ that satisfy

(i) $A - \prod_{i=1}^{n} f_i^{(0)} \equiv 0 \mod m$,

that is, we are given a factorization of A modulo m, and

(ii) $\gcd(f_i^{(0)}, f_j^{(0)}) = 1$ in the quotient ring $R[x]/m$ when $i \neq j$.

The input to Hensel lifting is A, m, the $f_i^{(0)}$ and a lifting bound $l \in \mathbb{N}$. The output of Hensel lifting is n polynomials $f_i^{(l)}$ in $R[x]$ satisfying

(iii) $A - \prod_{i=1}^{n} f_i^{(l)} \equiv 0 \mod m^l$,
(iv) $f_i^{(l)} \equiv f_i^{(0)} \mod m$ for $1 \leq i \leq n$ and
(v) $\deg(f_i^{(l)}, y) < \deg(m^l, y)$ for $1 \leq i \leq n$.

We say Hensel lifting *lifts* a factorization of A modulo m to a factorization modulo m^l. Condition (ii) guarantees the existence of the $f_i^{(l)}$. Condition (v) imposes uniqueness on $f_i^{(l)}$ up to multiplication by a unit in the quotient ring R/m^l.

Because we want to apply Hensel lifting in $R[x]$ to factor multivariate polynomials in $\mathbb{Z}[x_1, \ldots, x_n]$, we use a modified specification for Hensel lifting in $R[x]$ – see Ch. 6 of [7]. Suppose A is monic in $R[x]$ and A factors as $A = \prod_{i=1}^{n} f_i$ with $f_i \in R[x]$. We will comment on the treatment of the non-monic case in Section 5. In a multivariate factorization context, we will have chosen an integer α and we will have factored $A(x, \alpha)$ over $\mathbb{Z}$. By Hilbert irreducibility [9] (see also [6]) the $f_i(x, \alpha)$ are probably irreducible thus starting with $f_i^{(0)} = f_i(x, \alpha)$ we have images of the factors f_i. Thus for $m = (y - \alpha)$, the $f_i^{(0)}$ satisfy condition (i). Let us assume our choice of α also satisfies (ii).

The input to our Hensel lifting is A, $m = (y - \alpha)$, and the $f_i^{(0)}$. Instead of using a lifting bound l to obtain $f_i^{(l)}$ satisfying (iii), we design the Hensel lifting to stop lifting when $\sum_{i=1}^{n} \deg(f_i^{(k)}, y) \geq \deg(A, y)$ for some $k \in \mathbb{N}$, then we test if $A = \prod_{i=1}^{n} f_i^{(k)}$. If true, we have factored A in $\mathbb{Z}_p[x, y]$. If false, then one of the $f_i(x, \alpha)$ is not irreducible over $\mathbb{Z}$ and the multivariate factorization needs to restart with a new α.

ISSAC '22, July 4–7, 2022, Villeneuve-d'Ascq, France.

ACM ISBN 978-1-4503-8688-3/22/07…$15.00
https://doi.org/10.1145/XXXXXX.XXXXXX

There are two constructions of Hensel lifting: Linear Hensel Lifting (LHL) and Quadratic Hensel Lifting (QHL). We describe both here for $m = (y - \alpha)$ so that we can compare their cost for Hensel lifting in $\mathbb{Z}_p[x, y]$. Let $d_i = \deg(f_i, y)$ and

$$f_i = \sum_{j=0}^{d_i} \sigma_{i,j}(x)(y-\alpha)^j \text{ for } \sigma_{i,j} \in \mathbb{Z}_p[x].$$

Let $\sigma_{i,j} = 0$ for $j > d_i$. For $k > 0$, let

$$f_i^{(k)} = \sum_{j=0}^{k-1} \sigma_{i,j}(x)(y-\alpha)^j. \tag{1}$$

We call $f_i^{(k)}$ a k'th order approximation of the factor f_i. The $f_i^{(k)}$ satisfy $A - \prod_{i=1}^n f_i^{(k)} \equiv 0 \mod (y-\alpha)^k$.

In LHL, for $k = 1, 2, 3, \ldots$, the $\sigma_{i,k}$ are computed and the factors are updated using $f_i^{(k+1)} \leftarrow f_i^{(k)} + \sigma_{i,k}(x)(y-\alpha)^k$, that is, the degree of the factors in y increases linearly. An algorithm for LHL for $\mathbb{Z}_p[x, y]$ is presented by Bernardin in [1].

Let $r = 2^{k-1}$. In QHL, for $k = 1, 2, 3, \ldots$, update polynomials

$$\Delta_i = \sum_{j=0}^{r-1} \sigma_{i,r+j}(x)(y-\alpha)^j \text{ for } 1 \le i \le n$$

are computed then the factors are updated using $f_i^{(k+1)} \leftarrow f_i^{(k)} + \Delta_i(x, y)\,(y-\alpha)^r$, that is, the degree of the factors in y increases quadratically. QHL is presented in Ch. 15 of [6]. See also Bostan et. al. [3].

Let $d_x = \deg(A, x)$ and $d_y = \deg(A, y)$. If fast multiplication for $\mathbb{Z}_p[x, y]$ is available, QHL can be done in $O(M(d_x \cdot d_y) \log_2 n)$ arithmetic operations in $\mathbb{Z}_p$ where $M(d_x \cdot d_y)$ is the cost of multiplying two polynomials in $\mathbb{Z}_p[x, y]$ of degree d_x in x and d_y in y.

The most expensive part of LHL is the computation of the product of the factors $P = \prod_{i=1}^n f_i$. Bernardin's algorithm computes P coefficient by coefficient. Let $P^{(k)} = \prod_{i=1}^n f_i^{(k)}$. At the k'th step, it computes $\text{coeff}(P^{(k)}(x, y), (y-\alpha)^k)$ in an efficient way. Both Bernardin's algorithm and our algorithm can be viewed as relaxed Hensel lifting algorithms; they only compute the terms of the product $\prod_{i=1}^n f_i^{(k)}$ that we need at step k to obtain $\text{coeff}(P^{(k)}, (y-\alpha)^k)$. Other works using relaxed Hensel lifting include Berthomieu and Lebreton [2] van der Hoeven [21] and Lebreton [12].

Our first contribution is a LHL algorithm for monic A for $n \ge 2$ factors that does $O(d_x^2 d_y + d_x d_y^2)$ arithmetic operations in $\mathbb{Z}_p$. It generalizes the algorithm of Monagan [15] for $n > 2$ factors. As in [15], it uses classical univariate polynomial evaluation and interpolation (see Ch. 5 of [7]) on x to compute $\text{coeff}(P^{(k)}, (y-\alpha)^k)$. In total, our algorithm does $O(d_x^2 d_y + d_x\, d_y^2)$ arithmetic operations in $\mathbb{Z}_p$.

Our first implementation of our algorithm used space for $O(n\, d_x\, d_y)$ elements of $\mathbb{Z}_p$ because it stores the $n-1$ intermediate products $\{\prod_{i=1}^j f_i^{(k)} : 2 \le j \le n\}$. This is expensive for large n. Our second contribution is a redesign of Bernardin's coefficient update algorithm so that we get an algorithm with the same time complexity of $O(d_x^2 d_y + d_x\, d_y^2)$ arithmetic operations in $\mathbb{Z}_p$ but that requires space for only $O(\log_2 n\, d_x\, d_y)$ elements of $\mathbb{Z}_p$. We adapt the sub-product tree algorithm (see Ch. 10 of [6]) for this purpose.

Our third contribution is a C implementation of our new algorithm for Hensel lifting in $\mathbb{Z}_p[x, y]$ for primes $p < 2^{63}$ and a comparison of it with our Magma implementation of QHL using Magma's fast polynomial arithmetic (we implemented Algorithm 15.17 Multifactor Hensel Lifting from [6]). We have also implemented Bernardin's algorithm in C for comparison. The C code for our new algorithm and our Magma code for QHL is available on the web at `www.cecm.sfu.ca/~mmonagan/code/BHL` . Our Magma code is also given in Appendix 1.

On our benchmarks for $n = 4$ factors, we find (see Table 1) that our new algorithm is a factor of 10 to 120 times faster than our Magma QHL code for $p = 2^{31} - 1$ and $8 \le d_x = d_y \le 16384$. Although the Magma implementation is gaining speed relative to our new algorithm as d_x and d_y increase, it never catches up to our new algorithm before Magma runs out of space on our 64 gigabyte computer. We also noticed that for fixed d_x and d_y, our new algorithm gains speed relative to our Magma QHL code as n, the number of factors, increases.

Our C code is being used by Chen and Monagan [4] to factor multivariate polynomials in $\mathbb{Z}[x_1, \ldots, x_n]$ given by *black boxes* (see [11]). Their algorithm reduces MHL to many bivariate Hensel lifts in $\mathbb{Z}_p[x, y]$. When used to factor multivariate polynomials, because the degree of the polynomials are often under 100 in practical applications, we are also interested in when our new algorithm first beats Bernardin's algorithm. On our benchmarks, it already beats Bernardin's algorithm at degree $d_x = d_y = 8$ for $n = 4$ factors (see row $d = 2$ in Table 1).

Our paper is organized as follows. Section 2 gives details for Bernardin's version of LHL in $\mathbb{Z}_p[x, y]$ for $n \ge 2$ factors and shows that it has quartic complexity. In Section 3, we develop our cubic LHL algorithm for $\mathbb{Z}_p[x, y]$. In Section 4 we present benchmarks comparing the Bernardin's quartic LHL algorithm, our cubic LHL algorithm, and our Magma implementation of QHL. We also give some details about our C implementation. In Section 5, we summarize our contribution.

2 QUARTIC COST HENSEL LIFTING IN $\mathbb{Z}_p[x, y]$

In this section, we will give details on Bernardin's version of LHL in $\mathbb{Z}_p[x, y]$ for $n \ge 2$ presented in [1]. His algorithm is similar to the classical LHL algorithm given in Section 1. The algorithm uses Hensel lifting to factor a polynomial $A = \prod_{i=1}^n f_i \in \mathbb{Z}_p[x, y]$ assuming we have the relatively prime initial factors $f_i^{(0)} = f_i(x, \alpha) \in \mathbb{Z}_p[x]$ for $1 \le i \le n$ for some $\alpha \in \mathbb{Z}_p$. The k'th order approximations of the factors $f_i^{(k)}$ are the same as in (1).

The difference between the algorithms is the method Bernardin used to calculate the error. The classical algorithm calculates the error e_k of the factorization at step k, that is $e_k = A - \prod_{i=1}^n f_i^{(k)}$. From there, the algorithm calculates $c_k = e_k/(y-\alpha)^k \mod (y-\alpha)$, then solves the polynomial Diophantine equation

$$c_k = \sum_{i=1}^n \frac{\sigma_{i,k}}{f_i^{(0)}} \prod_{j=1}^n f_j^{(0)}$$

for $\sigma_{i,k}$ where $\deg(\sigma_{i,k}, x) < \deg(f_i^{(0)}, x)$ for $1 \le i \le n$.

Bernardin noticed that many of the multiplications done in calculating e_k were unnecessarily repeated. The polynomial $A(x, y)$

can be rewritten as a Taylor series about the point α. In other words, $A = \sum_{i=0}^{d_y} a_i(y-\alpha)^i$ where $a_i \in \mathbb{Z}_p[x]$ are obtained by applying polynomial long division. Thus, an alternate way to calculate c_k is

$$c_k = a_k - \text{coeff}\left(\prod_{i=1}^{n} f_i^{(k)}, (y-\alpha)^k\right), \tag{2}$$

so we need merely compute $\Delta^{(k)} = \text{coeff}(\prod_{i=1}^n f_i^{(k)}, (y-\alpha)^k)$ for any given k. Bernardin used a combination of convolution and successive polynomial multiplications to calculate $\Delta^{(k)}$ efficiently. The idea is to compute the product of the factors $f_i^{(k)}$ modulo $(y-\alpha)^{k+1}$ at each step, reusing the sub-products already computed in the previous step.

At step k, we have to compute $\text{coeff}(\prod_{i=1}^n f_i^{(k)}, (y-\alpha)^k)$ as described in (2). We define

$$P_q^{(k)} := \prod_{i=1}^{q} f_i^{(k)} \mod (y-\alpha)^{k+1} \tag{3}$$

for $2 \le q \le n$ with $c_k = \text{coeff}(P_n^{(k)}, (y-\alpha)^k)$.

The product of the first two factors gives

$$P_2^{(k)} = \sum_{i=0}^{k-1}\left(\sum_{j=0}^{i} \sigma_{1,j}\cdot\sigma_{2,i-j}\right)(y-\alpha)^i + \left(\sum_{j=1}^{k-1}\sigma_{1,j}\cdot\sigma_{2,k-j}\right)(y-\alpha)^k$$

For successive q we can compute $P_q^{(k)}$ as

$$P_q^{(k)} = \sum_{i=0}^{k-1}\left(\sum_{j=0}^{i} \rho^{(j)}\cdot\sigma_{q,i-j}\right)(y-\alpha)^i + \left(\sum_{j=1}^{k}\rho^{(j)}\cdot\sigma_{q,k-j}\right)(y-\alpha)^k$$

with $\rho^{(j)} := \text{coeff}(P_{q-1}^{(k)}, (y-\alpha)^j)$. Note that ρ is used exclusively for simplifying the notation and that the ρ's are different for varying q and k. We observe that the only terms that we have to compute and that have not already been computed in the previous step are

$$\sigma_{1,k}\cdot\sigma_{2,0} \text{ and } \sigma_{1,0}\cdot\sigma_{2,k}$$

when calculating $P_2^{(k)}$ and

$$\rho^{(0)}\sigma_{q,k}$$

for the subsequent factors. These terms need to be added to $P_2^{(k)}, ..., P_n^{(k)}$ once we calculate $\sigma_{q,k}$ for $1 \le q \le n$. When calculating $P_q^{(k+1)}$ in step $k+1$, we can reduce the number of multiplications done by re-using the ones done for $P_q^{(k)}$. In fact, we can simplify $P_q^{(k+1)}$ to

$$P_2^{(k+1)} = P_2^{(k)} + \left(\sum_{j=1}^{k}\sigma_{1,j}\cdot\sigma_{2,k-j+1}\right)(y-\alpha)^{k+1}$$

for two factors and

$$P_q^{(k+1)} = P_q^{(k)} + \left(\sum_{j=1}^{k+1}\rho^{(j)}\cdot\sigma_{q,k-j+1}\right)(y-\alpha)^{k+1}$$

for the remaining factors with $\rho^{(j)} := \text{coeff}(P_{q-1}^{(k+1)}, (y-\alpha)^j)$.

The products $P_q^{(k)}$ for $2 \le q \le n$ are stored in an $n \times (d_y+1)$ matrix G of polynomials in $\mathbb{Z}_p[x]$. The matrix G holds $n \times (d_y+1) \times (d_x+1)$ elements in $\mathbb{Z}_p$ and therefore calculating $\Delta^{(k)}$ uses $O(n\, d_x\, d_y)$ space. We present Bernardin's algorithm as Algorithm 1 listed below.

Remark: Let $\mu = \max(\{\deg(f_i, y) : 1 \le i \le n\})$. If A has a factorization $A = \prod_{i=1}^n f_i$, Algorithm 1 will reconstruct the factors at step $k = \mu$. However, Algorithm 1 will continue until $k = d_y$. For steps $\mu < k \le d_y$ it uses the following to verify the factorization: $A = \prod_{i=1}^n f_i$ if and only if $c_k = 0$ for $\mu < k \le d_y$.

We present two additional sub-algorithms: *CoefficientExtraction* which calculates the products $P_2^{(k)}, ..., P_n^{(k)}$ as described previously, and *CoefficientUpdate* which adds the missing terms into the products $P_2^{(k)}, ..., P_n^{(k)}$, stored in the polynomial matrix G, during step k. These algorithms are denoted by Algorithm 2 and Algorithm 3 respectively. In each algorithm, the order terms on the right count arithmetic operations in $\mathbb{Z}_p$.

Algorithm 1: Quartic linear Hensel lifting for $\mathbb{Z}_p[x]$: Monic Case

Input: prime p, $\alpha \in \mathbb{Z}_p$, $A \in \mathbb{Z}_p[x,y]$ and
  $f_1^{(0)}, f_2^{(0)}, ..., f_n^{(0)} \in \mathbb{Z}_p[x]$ satisfying
  (i) $A, f_1^{(0)}, f_2^{(0)}, ..., f_n^{(0)}$ are monic in x,
  (ii) $A(y=\alpha) = \prod_{i=1}^n f_i^{(0)}$ and
  (iii) $\gcd(f_i^{(0)}, f_j^{(0)}) = 1$ for $i \ne j$.
Output: $f_1, f_2, ..., f_n \in \mathbb{Z}_p[x,y]$ s.t. $A = \prod_{i=1}^n f_i$ or FAIL.
$d_x \leftarrow \deg(A, x)$; $d_y \leftarrow \deg(A, y)$;
for $i = 1$ to n **do** $f_i \leftarrow f_i^{(0)}$; **end**
Compute $a_0, a_1, ..., a_{d_y} \in \mathbb{Z}_p[x]$
  s.t. $A = \sum_{k=0}^{d_y} a_k(y-\alpha)^k$; ..... $O(d_x\, d_y^2)$
$G_{1,0} \leftarrow f_1$;
for $i = 1$ to $n-1$ **do**
  $G_{i+1,0} \leftarrow G_{i,0}\cdot f_{i+1}$; ..... $O(d_x^2)$
end
for $k = 1$ to d_y **do**
  $(\Delta, G) \leftarrow$ CoefficientExtraction
    $(p, \alpha, k, f_1, f_2, ..., f_n, G) \in \mathbb{Z}_p[x]$; ..... $O(k\, d_x^2)$
  $c_k \leftarrow a_k - \Delta$;
  if $\sum_{i=1}^n \deg(f_i, y) = d_y$ and $c_k \ne 0$ **then Return** FAIL;
  if $c_k \ne 0$ **then**
    Solve $\sum_{i=1}^n \frac{\sigma_i}{f_i^{(0)}} \prod_{j=1}^n f_j^{(0)} = c_k$ in $\mathbb{Z}_p[x]$ for
      $\sigma_1, \sigma_2, ..., \sigma_n \in \mathbb{Z}_p[x]$; ..... $O(n\, d_x^2)$
    for $i = 1$ to n **do**
      $f_i \leftarrow f_i + \sigma_i\cdot(y-\alpha)^k$; ..... $O(k\, d_x)$
    end
    $G \leftarrow$ CoefficentUpdate$(p, \alpha, k, f_1, ..., f_n, G)$; . $O(n\, d_x^2)$
  end
end
if $\sum_{i=1}^n \deg(f_i, y) = d_y$ **then Return** $f_1, f_2, ..., f_n$; **end**
Return FAIL;

Algorithm 2: Coefficient Extraction

Input: prime p, $\alpha \in \mathbb{Z}_p$, $k \in \mathbb{Z}^+$, $f_1, f_2, ..., f_n \in \mathbb{Z}_p[x,y]$, G an $n \times (d_y+1)$ matrix of elements in $\mathbb{Z}_p[x]$.
Output: Δ =coeff$(\prod_{i=1}^n f_i, (y-\alpha)^k) \in \mathbb{Z}_p[x]$, G an $n \times (d_y+1)$ matrix of elements in $\mathbb{Z}_p[x]$.
$MIN \leftarrow \max(0, k - \deg(f_2, y))$;
$MAX \leftarrow \min(k, \deg(f_1, y))$;
$G_{2,k} \leftarrow \sum_{j=MIN}^{MAX} \text{coeff}(f_1, (y-\alpha)^j) \cdot$ coeff$(f_2, (y-\alpha)^{k-j})$; ..... $O(kd_x^2)$
$d \leftarrow \deg(f_1, y) + \deg(f_2, y)$;
for $i = 3$ to n **do**
  $\delta \leftarrow d$; $d \leftarrow d + \deg(f_i, y)$;
  if $k \le d$ **then**
    $MIN \leftarrow \max(0, k-\delta)$; $MAX \leftarrow \min(k, \deg(f_i, y))$;
    $G_{i,k} \leftarrow \sum_{j=MIN}^{MAX} G_{i-1,k-j} \cdot$ coeff$(f_i, (y-\alpha)^j)$; ..... $O(kd_x^2)$
  end
end
Return $(G_{n,k}, G)$;

The most expensive operation in Algorithm 1 is the computation of $\Delta^{(k)}$ using the *CoefficientExtraction* sub-algorithm during Line 11. The cost of Line 11 is $O(k\, d_x^2)$, so it does $\sum_{k=1}^{d_y} O(k\, d_x^2) = O(d_x^2\, d_y^2)$ arithmetic operations in $\mathbb{Z}_p$.

Algorithm 3: Coefficient Update

Input: prime p, $\alpha \in \mathbb{Z}_p$, $k \in \mathbb{Z}^+$, $f_1, f_2, ..., f_n \in \mathbb{Z}_p[y]$, G an $n \times (d_y+1)$ matrix of elements in $\mathbb{Z}_p$.
Output: G an $n \times (d_y+1)$ matrix of elements in $\mathbb{Z}_p[x]$.
if $n > 2$ **then**
  $t \leftarrow \text{coeff}(f_1, (y-\alpha)^k)$;
  for $i = 2$ to n **do**
    // $t = \sum_{j=1}^{i} \frac{\sigma_{j,k}}{f_j^{(0)}} \prod_{m=1}^{i} f_m^{(0)}$
    $t \leftarrow \text{coeff}(f_i, (y-\alpha)^0) \cdot t +$ coeff$(f_i, (y-\alpha)^k) \cdot G_{i-1,0}$; ..... $O(d_x^2)$
    $G_{i,k} \leftarrow G_{i,k} + t$; ..... $O(d_x^2)$
  end
end
Return G;

3 CUBIC COST HENSEL LIFTING IN $\mathbb{Z}_p[x,y]$

In this section, we develop a cubic cost Hensel lifting algorithm which improves upon the arithmetic and space cost of Bernardin's quartic cost LHL algorithm. We state the key result of this paper as Theorem 1. We will justify the result of the theorem in this section.

Theorem 3.1. *Let $A \in \mathbb{Z}_p[x,y], d_x = \deg(A,x) > 1$, and $d_y = \deg(A,y) > 1$. Suppose $A = \prod_{i=1}^n f_i$ and we are given pairwise relatively prime images $f_i(x,\alpha)$ for $1 \le i \le n$ for some $\alpha \in \mathbb{Z}_p$. If $p \ge d_x$, we can compute $f_1, f_2, \ldots, f_n$*
(i) using $O(d_x^2\, d_y + d_x\, d_y^2)$ arithmetic operations in $\mathbb{Z}_p$, and
(ii) using space for $O(\log_2 n\, d_x\, d_y)$ elements of $\mathbb{Z}_p$.

We made two improvements to Bernardin's algorithm. The first improvement changed how we calculate $\Delta^{(k)}(x)$, while the second improvement changed how we calculated the required coefficient of the product $\prod_{i=1}^n f_i^{(k)}$ during each iteration of Algorithm 1.

The arithmetic cost of Algorithm 1, described in Section 2, is dominated by Line 11. Therefore, to reduce the cost of the algorithm, a better method is needed to compute $\Delta^{(k)}(x)$ =coeff$(\prod_{i=1}^n f_i^{(k)}, (y-\alpha)^k)$ for a given $1 \le k \le d_y$. Instead of computing the product of n bivariate polynomials during step k of the algorithm, we use a system of polynomial evaluation, single integer multiplication in $\mathbb{Z}_p$, and interpolation to compute $\Delta^{(k)}$. We begin by evaluating each of the n bivariate polynomials $f_1^{(k)}, f_2^{(k)}, ..., f_n^{(k)} \in \mathbb{Z}_p[x,y]$ at $x = \beta_j$ for $0 \le j < d_x$ where $\beta_j \in \mathbb{Z}_p$. We discuss the evaluation points we chose in Section 4.1. Next, we compute the coefficient of $(y-\alpha)^k$ for the product of evaluated polynomials at each of the d_x evaluation points,

$$\Delta_j^{(k)} = \text{coeff}\left(\prod_{i=1}^{n} f_i^{(k)}(\beta_j, y), (y-\alpha)^k\right) \text{ for } 0 \le j < d_x.$$

Finally, we interpolate the set $\{\Delta_0^{(k)}, ..., \Delta_{d_x-1}^{(k)}\}$ in x to obtain $\Delta^{(k)}(x)$. This method of calculating $\Delta^{(k)}(x)$ is presented as a homomorphism diagram in Figure 1.

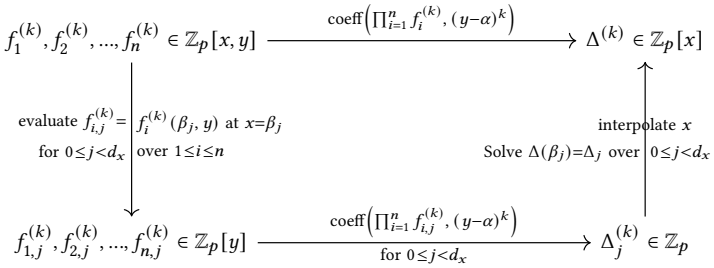

Figure 1: Homomorphism diagram for computing $\Delta(x)$ at iteration $k \ge 1$

The second improvement we've implemented is how we calculate the product $\prod_{i=1}^n f_{i,j}^{(k)} \in \mathbb{Z}_p[y]$ for $0 \le j < d_x$. When Bernardin calculated $\Delta(x)$, he multiplied the n polynomials sequentially. By doing this, Bernardin had to store the results in a matrix of size $O(n\, d_x\, d_y)$. Instead, we chose to multiply the factors together as pairs. For example, we first multiply the initial factors as the products $f_{1,j}^{(k)} f_{2,j}^{(k)}, f_{3,j}^{(k)} f_{4,j}^{(k)}, ..., f_{n-1,j}^{(k)} f_{n,j}^{(k)}$. We then multiply those products together as pairs, i.e. $\prod_{i=1}^4 f_{i,j}^{(k)}, ..., \prod_{i=n-3}^n f_{i,j}^{(k)}$. We repeat this process until we arrive at the final product $\prod_{i=1}^n f_{i,j}^{(k)}$. As for Bernadin's algorithm we will store these polynomial products in the G matrix.

For simplicity, let $n = 2^\ell$ for some $\ell \in \mathbb{N}$. We define

$$P_{q,r}^{(k)} := \prod_{i=(r-1)2^q+1}^{r2^q} f_{i,j}^{(k)} \mod (y-\alpha)^{k+1} \tag{4}$$

with c_k :=coeff$(P_{\ell,1}^{(k)}, (y-\alpha)^k)$ for $1 \le q \le \ell$ and $1 \le r \le 2^{\ell-q}$.

In the following, we will denote the coefficients of $(y-\alpha)^j$ in $f_i^{(k)}$ as $\sigma_{i,j}$ as they are defined in (1). The product of the initial pairs gives

$$P_{1,r}^{(k)} = \sum_{s=0}^{k-1}\left(\sum_{t=0}^{s}\sigma_{2r-1,t}\cdot\sigma_{2r,s-t}\right)(y-\alpha)^s + \left(\sum_{t=1}^{k-1}\sigma_{2r-1,t}\cdot\sigma_{2r,k-t}\right)(y-\alpha)^k$$

for $1 \le r \le 2^{\ell-1}$. Let $\rho_r^{(t)}$:=coeff$(P_{q-1,r}, (y-\alpha)^t)$. For successive q we can compute $P_{q,r}^{(k)}$ as

$$P_{q,r}^{(k)} = \sum_{s=0}^{k}\left(\sum_{t=0}^{s}\rho_{2r-1}^{(t)}\cdot\rho_{2r}^{(s-t)}\right)(y-\alpha)^s$$

for $2 \le q \le \ell$ and $1 \le r \le 2^{\ell-q}$ for each q. Note that ρ_{2r-1} and ρ_{2r} are used exclusively to simplify the notation and that the ρ's are different for varying q, r and t. We observe that the only terms that we have to compute and that have not already been computed in the previous steps are

$$\sigma_{2r-1,k}\cdot\sigma_{2r,0} \text{ and } \sigma_{2r-1,0}\cdot\sigma_{2r,k}$$

in the initial pairs $P_{1,r}$ for $1 \le r \le 2^{\ell-1}$. These terms will also be absent from any of the calculations which produce the polynomials $P_{q,r}$ for $2 \le q \le \ell$ and $1 \le r \le 2^{\ell-q}$.

By making these changes, we present our cubic LHL algorithm as Algorithm 4. The main difficulty in understanding and implementing the Algorithm 4 is the indexing. To verify that the indexing of the pseudo-code in the paper is correct, we re-implemented the algorithms in Maple using the pseudo-code.

We begin our analysis with the evaluation of the n bivariate polynomials $f_1^{(k)}, ..., f_n^{(k)} \in \mathbb{Z}_p[x, y]$ at $x = \beta_j$ for $0 \le \beta_j < d_x$. By applying Horner's method at Lines 14 and 31, we can evaluate all of the n polynomials at $\beta_j \in \mathbb{Z}_p$ using $O(d_x d_y)$ arithmetic operations. Thus, evaluating has a total cost of $\sum_{i=0}^{d_x-1} O(d_x d_y) = O(d_x^2 d_y)$. Next, we compute $\Delta_j^{(k)}$ as described above. We modified Algorithm 2 to multiply polynomials in $\mathbb{Z}_p[y]$ instead of $\mathbb{Z}_p[y][x]$. Additionally, we modified Algorithm 2 to multiply the polynomials as pairs instead of sequentially. We present this algorithm as Algorithm 5.

Algorithm 5 has an arithmetic complexity of $O(kn)$ which is bounded above by $O(d_y)$. The overall cost of this sub-algorithm is $\sum_{i=1}^{d_y}\sum_{j=0}^{d_x-1} O(d_y) = O(d_x\, d_y^2)$.

Finally, we use Lagrange interpolation at Line 23 of Algorithm 4 to recover $\Delta^{(k)}(x)$ from $\{\Delta_0^{(k)}, ..., \Delta_{d_x-1}^{(k)}\}$. Lagrange interpolation has an arithmetic cost of $O(d_x^2)$, so, accounting for the loop on k it has an overall cost of $O(d_x^2 d_y)$.

We conclude that our method of calculating $\Delta^{(k)}$ has an arithmetic cost of $O(d_x^2 d_y + d_x d_y^2)$. An iterative method for solving the multi-Diophantine equation $\sum_{i=1}^{n}\sigma_i M_i = c_k$ in line 27 of Algorithm 4 is is presented in Ch. 6 of [7]. The analysis in [19] shows that it can be done in $O(d_x^2)$ arithmetic operations in $\mathbb{Z}_p$. Accounting for the loop on k the total cost is $d_y\, O(d_x^2) = O(d_x^2\, d_y)$.

We have two more operations to consider in our cubic LHL algorithm. Firstly, like Algorithm 1, we need a way to update the products stored in G after we calculate $\sigma_1, ..., \sigma_n$ in Line 27. We can

Algorithm 4: Cubic Bivariate Hensel Lifting Algorithm: Monic Case

1 **Input**: prime p, $\alpha \in \mathbb{Z}_p$, $A \in \mathbb{Z}_p[x, y]$ and
  $f_1^{(0)}, f_2^{(0)}, ..., f_n^{(0)} \in \mathbb{Z}_p[x]$ satisfying
  (i) $A, f_1^{(0)}, f_2^{(0)}, ..., f_n^{(0)}$ are monic in x,
  (ii) $A(y=\alpha) = f_1^{(0)} f_2^{(0)} ... f_n^{(0)}$ and
  (iii) $\gcd(f_i^{(0)}, f_j^{(0)}) = 1$ for $i \ne j$.
2 **Output**: $f_1, f_2, ..., f_n \in \mathbb{Z}_p[x, y]$ s.t. $A = f_1\, f_2 ... f_n$ or FAIL.
3 $d_x \leftarrow \deg(A, x)$; $d_y \leftarrow \deg(A, y)$;
4 $t \leftarrow n$; $np \leftarrow 1$; $M \leftarrow \prod_{j=1}^{n} f_j^{(0)}$;
5 **while** $t > 1$ **do** $np \leftarrow np + t$; $t \leftarrow \lceil t/2 \rceil$; **end**
6 **for** $j = 0$ to $d_x - 1$ **do**
7 **for** $i = 0$ to $np - 1$ **do**
8 $G_{j,i}, D_{j,i}, S_{j,i} \leftarrow$ Array(0..0), 0, 1;
9 **end**
10 **end**
11 Compute $a_0, a_1, ..., a_{d_y} \in \mathbb{Z}_p[x]$ s.t.
  $A = \sum_{k=0}^{d_y} a_k (y-\alpha)^k$; ..... $O(d_x\, d_y^2)$
12 **for** $i = 1$ to n **do**
13 $f_i \leftarrow f_i^{(0)}$; $M_i \leftarrow M / f_i^{(0)}$;
14 $G_{j,i,0} \leftarrow f_i^{(0)}(x = \beta_j) \in \mathbb{Z}_p$ for $0 \le j < d_x$;
15 **end**
16 **for** $j = 0$ to $d_x - 1$ **do**
17 $(\Delta_j, G_j, S_j, D_j) \leftarrow$ CoefficientExtraction
  $(p, \alpha, 0, np, S_j, D_j, G_j)$;
18 **end**
19 **for** $k = 1$ to d_y **do**
20 **for** $j = 0$ to $d_x - 1$ **do**
21 $(\Delta_j, G_j, S_j, D_j) \leftarrow$ CoefficientExtraction
  $(p, \alpha, k, np, S_j, D_j, G_j)$; ..... $O(k\, n)$
22 **end**
23 interpolate $\Delta(x) \in \mathbb{Z}_p[x]$ s.t. $\Delta(\beta_j) = \Delta_j$; ..... $O(d_x^2)$
24 $c_k \leftarrow a_k - \Delta$;
25 **if** $\sum_{i=1}^{n} \deg(f_i, y) = d_y$ and $c_k \ne 0$ **then Return** FAIL;
26 **if** $c_k \ne 0$ **then**
27 Solve $\sum_{i=1}^{n} \sigma_i M_i = c_k$ for $\sigma_1, ..., \sigma_n \in \mathbb{Z}_p[x]$ with
  $\deg(\sigma_i, x) < \deg(f_i^{(0)}, x)$; ..... $O(d_x^2)$
28 **for** $i = 1$ to n **do**
29 $f_i \leftarrow f_i + \sigma_i \cdot (y-\alpha)^k$;
30 **for** $j = 0$ to $d_x - 1$ **do**
31 $s \leftarrow \sigma_i(x = \beta_j) \in \mathbb{Z}_p$; ..... $O(\deg(f_i, x))$
32 **if** $s \ne 0$ **then**
33 $(G_{j,i}, S_{j,i}) \leftarrow$ Extend$(S_{j,i}, G_{j,i}, D_{j,i}, k)$;
34 $D_{j,i} \leftarrow k$; $G_{j,i,k} \leftarrow s$;
35 **end**
36 **end**
37 **end**
38 $(G_j, S_j, D_j) \leftarrow$CoefficientUpdate$(p, \alpha, k,$
  $np, S_j, D_j, G_j)$ for $0 \le j \le d_x - 1$; ..... $O(n\, d_x)$
39 **end**
40 **end**
41 **if** $\sum_{i=1}^{n} \deg(f_i, y) = d_y$ **then Return** $f_1, f_2, ..., f_n$; **end**
42 **Return** FAIL;

Algorithm 5: Cubic CoefficientExtraction algorithm

Input: prime p, $\alpha \in \mathbb{Z}_p$, $k \in \mathbb{Z}$, $np \in \mathbb{Z}^+$, $S, D \in \mathbb{Z}^{np}$, and H an array of arrays of length np, the lengths of each array are stored in S.
Output: $\Delta = \text{coeff}(\prod_{i=1}^{n} f_i, (y-\alpha)^k) \in \mathbb{Z}_p$, H the array of arrays of length np, and $S, D \in \mathbb{Z}^{np}$.
$t \leftarrow n$; $s \leftarrow 0$;
while $t > 1$ **do**
  $i \leftarrow 0$;
  while $i < \lfloor t/2 \rfloor$ **do**
    $d \leftarrow D_{s+2i} + D_{s+2i+1}$; $m \leftarrow s+t+i$;
    $(H_m, S_m) \leftarrow$Extend(S_m, H_m, D_m, d);
    if $k \le d$ **then**
      $MIN, MAX \leftarrow \max(0, k - D_{s+2i+1}), \min(k, D_{s+2i})$;
      $H_{m,k} \leftarrow \sum_{\ell=MIN}^{MAX} H_{s+2i,\ell} \cdot H_{s+2i+1,k-\ell}$; .... $O(k)$
    end
    $i \leftarrow i+1$;
  end
  if t *is odd* **then**
    $m \leftarrow s+t+i$;
    $(H_m, S_m) \leftarrow$Extend$(S_m, H_m, D_m, D_{s+2i})$;
    if $D_{s+2i} \ge k$ **then**
      $D_m \leftarrow D_{s+2i}$ $H_{m,k} \leftarrow H_{s+2i,k}$
    end
  end
  $s \leftarrow s+t$; $t \leftarrow \lceil t/2 \rceil$;
end
if $k+1 > S_s$ **then Return** 0; **end**
Return $(H_{s,k}, H, S, D)$;

do this by modifying Algorithm 3 similarly to how we modified Algorithm 2. We multiply polynomials in $\mathbb{Z}_p[y]$ instead of $\mathbb{Z}_p[x][y]$ and we account for the fact that polynomials are multiplied as pairs instead of sequentially. We present this algorithm as Algorithm 6. Algorithm 6 does $O(n)$ multiplications. Accounting for the loops in Lines 19 and 38 of Algorithm 4, the total number of operations of Algorithm 6 is $O(n\, d_x\, d_y)$. This ends the proof of Theorem 1 (i).

The second and final problem is how we store the polynomials $P_{q,r}^{(k)} \in \mathbb{Z}_p[y]$ for $1 \le q \le \ell$ and $1 \le r \le 2^{\ell-q}$ in the matrix G. As a reminder to the reader, we assumed $n = 2^\ell$ for some $\ell \in \mathbb{N}$ and there is a different set of $P_{q,r}^{(k)}$ polynomials for each of the d_x evaluation points. G is a $d_x \times np$ matrix, where each entry is a polynomial in $\mathbb{Z}_p[y]$. The term np refers to the number of polynomials calculated in (4), that is $np = |\{P_{q,r} : 1 \le q \le \ell, 1 \le r \le 2^{\ell-q}\}|$. Each polynomial may require a different amount of space allocated to it.

Consider the set of polynomials $P_{q,r}^{(k)}$ for a particular q. This set contains some combination of products of the polynomials $f_{1,j}^{(k)}, ..., f_{n,j}^{(k)}$ and therefore requires at most $d_y + n$ elements of $\mathbb{Z}_p$ in memory if one accounts for the constants. We have elected to dynamically store the coefficients for each polynomial. We initially give each polynomial $P_{q,r}^{(0)}$ a single element of memory as they are initialized as constants at Line 14. When the polynomials grows to need more space, we allocate a new block of memory of size 2^t, where t is the smallest integer such that $\deg(P_{q,r}^{(k)}, y) + 1 \le 2^t$.

Algorithm 6: Cubic CoefficientUpdate Algorithm

Input: prime p, $\alpha \in \mathbb{Z}_p$, $k \in \mathbb{Z}^+$, $np \in \mathbb{Z}^+$, $S, D \in \mathbb{Z}^{np}$, and H an array of arrays of length np, the lengths of each array are stored in S.
Output:the list of arrays H, and the arrays S and D.
$s \leftarrow 0$; $t \leftarrow n$;
while $t > 1$ **do**
  $i \leftarrow 0$;
  while $i < \lfloor t/2 \rfloor$ **do**
    if $s = 0$ **then**
      $T_i \leftarrow 0$;
      if $D_{2i} = k$ **then** $T_i \leftarrow H_{s+2i,k} \cdot H_{s+2i+1,0}$;
      if $D_{2i+1} = k$ **then** $T_i \leftarrow T_i + H_{s+2i,0} \cdot H_{s+2i+1,k}$;
    else
      $T_i \leftarrow H_{s+2i,0} \cdot T_{2i+1} + H_{s+2i+1,0} \cdot T_{2i}$;
    end
    $m \leftarrow s+t+i$; $d \leftarrow D_{s+2i} + D_{s+2i+1}$;
    $(H_m, S_m) \leftarrow$ Extend(S_m, H_m, D_m, d);
    $H_{m,k} \leftarrow H_{m,k} + T_i$;
    $D_m \leftarrow d$; $i \leftarrow i+1$;
  end
  if t is odd **then**
    $T_i \leftarrow H_{s+2i,k}$; $m \leftarrow s+t+i$;
    $(H_m, S_m) \leftarrow$ Extend$(H_m, S_m, D_m, D_{s+2i})$;
    $H_{m,k} \leftarrow T_i$; $D_m \leftarrow D_{s+2i}$
  end
  $s \leftarrow s+t$; $t \leftarrow \lceil t/2 \rceil$;
end
Return (H, S, D);

For a particular q, we can store the polynomials $P_{q,r}^{(k)}$ for $1 \le r \le 2^{\ell-q}$ using at most $4(d_y + n)$ elements of space throughout the execution of Algorithm 4. Therefore, since there are ℓ values for q, and we must repeat this process for each of the d_x evaluation points, the matrix G will store as much as $4d_x \times \lceil \log_2 n \rceil \times (d_y + n)$ elements of $\mathbb{Z}_p$. Thus, our cubic LHL algorithm uses $O(\log_2 n\, d_x\, d_y)$ space. This completes our proof of Theorem 1 (ii).

Algorithm 7: Extend Algorithm

Input: $S \in \mathbb{Z}$, A an array of length S and $dOld, dNew \in \mathbb{Z}$
Output: an array of elements in $\mathbb{Z}_p$, the length of the array
if $dNew + 1 \le S$ **then Return** (A, S); **end**
$k \leftarrow \lceil \log_2(dNew + 1) \rceil$;
$B \leftarrow$ Array$(0..2^k - 1)$;
for $i = 0$ to $dOld$ **do** $B_i \leftarrow A_i$; **end**
for $i = dOld + 1$ to $2^k - 1$ **do** $B_i \leftarrow 0$; **end**
Return $(B, 2^k)$;

We present the algorithm which allocates more space for the polynomials in matrix G as Algorithm 7. The notation used in Algorithm 7 is based on Maple notation where it's possible to create and return an array from within a local function, as well as store an array of arbitrary size within a matrix. In our C implementation, we

initially allocate an array of pointers of size $d_x \times np$, each pointing at a single block of memory. When a polynomial needs more space, we use the next available 2^t contiguous spaces to store the larger polynomial. The total storage used for the factors and their products is $4d_x \times \lceil \log_2 n \rceil \times (d_y + n)$ words for the coefficients and for $d_x \times np$ words for the pointers.

4 IMPLEMENTATION AND BENCHMARKS FOR $\mathbb{Z}_p[x, y]$

Table 1 shows the timings for Hensel lifting in $\mathbb{Z}_p[x, y]$ for three algorithms using prime $p = 2^{31} - 1$. The timings were obtained using a server with 64 gigabytes of RAM and two Intel Xeon E5-2660 8 core processors running at 2.20GHz base and 3.00GHz turbo.

For our benchmarks in Table 1, we factor a polynomial $A = f_1 f_2 f_3 f_4$ for $n = 4$ factors. Each of the factors has the form $f_k = x^d + \sum_{i=0}^{d-1} \sum_{j=1}^{d} c_{ij} x^i y^j$ for $1 \le k \le 4$ where the coefficients c_{ij} are chosen at random from $[0, p)$. The variable d denotes the degree of x and y in the factors, i.e. $d = \deg(f_i, x) = \deg(f_i, y)$. As A is the product of 4 factors, $d_x = d_y = 4d$. We input $\alpha = 3$, A and $f_i^{(0)} = f(x, 3)$ for $1 \le i \le 4$ to Hensel lifting.

We have implemented our cubic LHL algorithm and Bernardin's quartic LHL algorithm in C. In both algorithms, we have implemented Shaw and Traub's method [20] to perform the change of base. We previously calculated it using polynomial long division, but Shaw and Traub's method performed significantly better. We used Lagrange interpolation to calculate $\Delta^{(k)}(x)$ in our cubic LHL algorithm. In section 4.1, we discuss how we implemented optimization improvements to Lagrange interpolation.

In Table 1, the second column labelled "Old LHL" is for the Bernardin's quartic $O(d_x^2 d_y^2)$ algorithm. The third column labelled "New LHL" is for the cubic $O(d_x^2 d_y + d_x d_y^2)$ algorithm where we used Horner evaluation and Lagrange interpolation to compute $\Delta^{(k)}(x)$. The fourth and fifth columns labelled "Fast QHL in Magma" are for our Magma implementation of QHL in $\mathbb{Z}_p[x, y]$. The timings in the column "Timings 1" occur when we set the four factors to have a degree of $d-1$ in the indeterminate y and the timings in the column "Timings 2" are when we set the degrees of both indeterminates to d. This is to avoid any unfair comparisons because QHL takes significantly longer to execute when the degrees of the factors in y are a power of 2. Magma uses fast multiplication in $\mathbb{Z}_p[x, y]$ and fast division in $\mathbb{Z}_p[x, y] \bmod (y - \alpha)^k$.

As the reader can see, our cubic LHL algorithm (column 3) beats Bernardin's quartic LHL algorithm from $d \ge 2$ onward. At $d = 1024$ the cubic LHL algorithm beats the quartic LHL algorithm by a factor of $15877.48/190.13 = 83.5$. In comparison with QHL, the cubic LHL algorithm beats magma's QHL by a factor of $3.16/0.06118 = 51.7$ (Timings 1) and $6.33/0.06118 = 103.5$ (Timings 2) when $d = 64$ and $2974.6/190.13 = 15.6$ (Timings 1) and $5599.3/190.13 = 29.4$ (Timings 2) when $d = 1024$. The cubic LHL algorithm is faster than Magma's fast algorithm at $d = 2048$. Even at $d = 2048$, the largest polynomials we are computing, Magma's fast method hasn't caught up with our cubic LHL algorithm yet. We are unable to get any further timings for Magma as the calculations would use more than 32GB of RAM. Thus, the cubic LHL algorithm is the fastest algorithm for $d \ge 2$.

d	Old LHL $O(d_x^2 d_y^2)$	New LHL $O(d_x^2 d_y + d_x d_y^2)$	Fast QHL in Magma	
			Timings 1	Timings 2
2	0.10ms	0.06ms	1.16ms	3.44ms
4	0.22ms	0.18ms	7.48ms	17.82ms
8	0.84ms	0.54ms	30.30ms	65.61ms
16	4.82ms	2.12ms	125.22ms	247.44ms
32	48.25ms	10.28ms	619.90ms	1,225.6ms
64	505.64ms	61.18ms	3.16s	6.33s
128	6.002s	401.14ms	17.58s	35.21s
256	76.70s	3.41s (0.12gb)	97.46s	204.27s
512	1,073.8s	25.17s (0.46gb)	553.85s	1,137.71s
1024	15,877.5s	190.13s (1.83gb)	2,974.6s	5,599.3s
2048	NA	1,461.5s (7.32gb)	15,583.3s	>32gb
4096	NA	12,614.7s (29.3gb)	NA	NA

Table 1: Hensel lifting timings for $\mathbb{Z}_p[x, y]$ for $p = 2^{31} - 1$ and $n = 4$ factors of degree d in x and y. NA = Not attempted

Table 2 shows the timing breakdown of the individual algorithms used during a single run of our cubic LHL algorithm. We use the same parameters described for the previous benchmarks, except we only consider the $d = 1024$ case. The factors are in the same form as described above.

The first column of Table 2, "Procedure" defines each of the major procedures used in our cubic LHL algorithm. The second column labelled "Time(ms)" describes the time used to perform each procedure. The final column "Percentage" gives the percentage of overall time used by each procedure. As the reader can see, the most expensive operation is "Change of Base" which uses 25% of the overall running time.

Procedure	Time(ms)	Percentage
Change of Base	46,750	24.59
Generate Lagrange Polynomials	890	0.47
CoefficientExtraction	43,350	22.80
Polynomial Interpolation	39,120	20.58
Solving the Diophantine equation	32,250	16.96
Polynomial Evaluation	11,610	6.11
CoefficientUpdate	1,550	0.82
Miscellaneous Operations	14,610	7.68
Overall Time	190,130	100.00

Table 2: Cubic Linear Lift subalgorithm timings for $\mathbb{Z}_p[x, y]$ with $p = 2^{31} - 1$. Compares the execution times for $d = 1024$.

4.1 Optimizations for $\mathbb{Z}_p[x, y]$

We made two notable optimizations to our C implementation. Firstly, we used an accumulator to reduce the number of divisions by p in $\mathbb{Z}_p$. Secondly, we implemented an improved version of Lagrange interpolation. Both changes led to a significant improvement in the overall execution time of our algorithm.

It is well known that hardware integer division instructions are much slower than hardware integer multiplication instructions. In [8], Granlund and Montgomery show how to speedup division by using two multiplications and several additions, shifts and bitwise logical operations to divide by p. For the "Old LHL" timings in Table 1, we are using Moller and Granlund's improved algorithm

from [14]. For convolutions of the form $\sum_{i=0}^{n} a_i b_i$ and $\sum_{i=0}^{n} a_i b_{n-i}$ in $\mathbb{Z}_p$ we obtain a significant speedup by using a double precision accumulator to reduce the number of divisions by one.

We have implemented our code to store integers that are 64 bits in size, and we implemented a 128 bit accumulator. Our algorithm works for primes of size up to 63 bits. We have implemented our algorithm to use an accumulator whenever $p < 2^{50}$. As a result of this, the product of any two integers will be at most 98 bits in size. This means we can add at least $2^{128}/2^{98} = 2^{30}$ products together before we risk overflowing the accumulator. Implementing an accumulator reduces the number of divisions for the convolutions $\sum_{i=0}^{n} a_i b_i$ and $\sum_{i=0}^{n} a_i b_{n-i}$ from $n+1$ to 1.

The timings for the column "Old LHL" in Table 1 are for a reorganization of the evaluation and interpolation algorithm so that we can use an accumulator method. We also use $0, \pm 1, \pm 2, ..., \pm \frac{d_x}{2}$ for evaluation points which allows us to speed up evaluation and interpolation by a further factor of 2.

Let $c(x) = \sum_{i=0}^{d} c_i x^i$ be the polynomial we wish to interpolate. In the following, we assume d is even; if not, we add 1 to d and use an additional evaluation point.

It is easy to evaluate $c(x)$ in $\mathbb{Z}_p[x]$ twice as fast using $\pm$ points. We can write $c(x) = a(x^2) + xb(x^2)$ where $a(x) = \sum_{i=0}^{d/2} c_{2i} x^i$ and $\sum_{i=0}^{d/2-1} c_{2i+1} x^i$. If we have already evaluated $c(\alpha) = a(\alpha^2) + \alpha b(\alpha^2)$, we can compute $c(-\alpha) = a(\alpha^2) - \alpha b(\alpha^2)$ using one additional subtraction. To also use the accumulator trick, we can compute $a(\alpha^2)$ via the dot product $[a_0, a_2, a_4, ..., a_d] \cdot [1, \alpha^2, ..., \alpha^d]$ and $b(\alpha^2)$ via the dot product $[a_1, a_3, ..., a_{d-1}] \cdot [1, \alpha^2, ..., \alpha^{d-2}]$. For $\alpha = i$, the arrays $[1, \alpha^2, \alpha^4, ..., \alpha^d]$ for $\alpha = 1, 2, ..., d/2$ are computed before the main Hensel loop so they can be reused.

Let $c(x) = \sum_{i=0}^{d} c_i x^i$ and assume we have computed $c(0)$ and $c(\pm i)$ for $1 \le i \le d/2$. We will use Lagrange interpolation to interpolate $c(x)$. Let

$$L(x) = \prod_{i=-d/2}^{d/2} (x - i) \text{ and } L_i(x) = \frac{L(x)}{(x-i)} \text{ for } -\frac{d}{2} \le i \le \frac{d}{2}.$$

The polynomials $L_i(x)$ are the Lagrange basis polynomials so we may write $c(x) = \sum_{i=-d/2}^{d/2} \alpha_i L_i(x)$ for some unique α_i. To determine the Lagrange coefficients α_i, since $L_j(i) = 0$ for $j \ne i$, we have $\alpha_i = c(i)/L_i(i)$.

In [15] Monagan shows how to efficiently calculate the coefficients $c_0, c_1, ..., c_d$ using $(\frac{d}{2}+1)\frac{d}{2}$ multiplications and using the double precision accumulator, so that only $O(d)$ divisions are done.

5 CONCLUSION

In this paper, we have developed a new algorithm for Linear Hensel Lifting (LHL) for $R[x]$ where $R = \mathbb{Z}_p[y]$ which works for $n \ge 2$ factors and has cubic time complexity. Our work generalizes the work of Monagan [15] for $n = 2$ factors. We have implemented our new algorithm in C for $p < 2^{63}$. Our C implementation beats our Magma implementation of fast Quadratic Hensel Lifting for a very wide range of input sizes.

Our C code is being used by Chen and Monagan [4] to factor multivariate polynomials in $\mathbb{Z}[x_1, \ldots, x_n]$. In this context our cubic LHL algorithm beats Bernardin's quartic LHL algorithm early enough to be useful.

Our Hensel lifting algorithm does not handle the general non-monic case. In current work we are studying how to modify our algorithm so we can factor $A = \prod_{i=1}^{n} f_i$ in $R[x]$ when $\deg(A, y) > 0$. In a multivariate factorization context, if we use Wang's leading coefficient pre-determination algorithm from [22], then we may assume we know $\text{lc}(f_i)$. In this case, if we "attach" $\text{lc}(f_i)$ to $f_i^{(0)}$ (see [7]) we need to modify our construction to compute $\text{coeff}(\prod_{i=1}^{n} f_i^{(k)}, (y-\alpha)^k)$ for $k \ge 1$.

REFERENCES

[1] Bernardin, L. On bivariate Hensel lifting and its parallelization. In *Proceedings of ISSAC '98* (1998), ACM, pp. 96–100.

[2] Berthomieu, J., and Lebreton, R. Relaxed p-adic hensel lifting for algebraic systems. In *Proceedings of ISSAC '12* (2012), ACM, p. 59–66.

[3] Bostan, A., Lecerf, G., Salvy, B., Schost, E., and Wiebelt, B. Complexity Issues in Bivariate Polynomial Factorization. In *Proceedings of ISSAC '04* (2004), pp. 42–49.

[4] Chen, T., and Monagan, M. Parallel algorithms for factoring multivariate polynomials represented by black boxes. *To appear in the post conference proceedings of CASC 2021* (2021).

[5] Gao, S. Factoring multivariate polynomials via partial differential equations. *Mathematics of Computation* ***72***, 242 (2003), 801–822.

[6] Gathen, J. v. z., and Jürgen, G. *Modern Computer Algebra*, 3 ed. Cambridge University Press, 2013.

[7] Geddes, K. O., Czapor, S. R., and Labahn, G. *Algorithms for Computer Algebra*. Kluwer Academic, 1992.

[8] Granlund, T., and Montgomery, P. L. Division by invariant integers using multiplication. *ACM SIGPLAN Notices* ***29***, 6 (1994), 61–72.

[9] Hilbert, D. Ueber die irreducibilität ganzer rational functionen mit ganzzahigen koeffizienten. *J. Renie Agewandte Mathematik* ***110*** (1982), 104–129.

[10] Kaltofen, E. A polynomial reduction from multivariate to bivariate integral polynomial factorization. In *Proceedings of STOC '82* (1982), ACM, p. 261–266.

[11] 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. *J. Symb. Comput.* ***9***, 3 (1990), 301–320.

[12] Lebreton, R. Relaxed hensel lifting of triangular sets. *J. Symb. Comput.* ***68***, 2 (2015), 230–258.

[13] Lecerf, G. Improved dense multivariate polynomial factorization algorithms. *J. Symb. Comput.* ***42***, 4 (2007), 477–494.

[14] Moller, N., and Granlund, T. Improved Division by Invariant Integers. *IEEE Trans. Computers* ***60*** (2011), 165–175.

[15] Monagan, M. Linear Hensel Lifting for $\mathbb{Z}_p[x, y]$ and $\mathbb{Z}[x]$ with Cubic Cost. In *Proceedings of ISSAC '19* (2019), pp. 299–306.

[16] Monagan, M., and Tuncer, B. Sparse Multivariate Hensel Lifting: A High-Performance Design and Implementation. In *Proceedings of ICMS '18, LNCS* ***10931*** (2018), Springer, pp. 359–368.

[17] Moses, J., and Yun, D. Y. Y. The EZ GCD Algorithm. In *Proceedings ACM '73* (1973), ACM, p. 159–166.

[18] Musser, D. Multivariate polynomial factorization. *J. ACM* ***22*** (1975), 291–308.

[19] Paluck, G. A new bivariate Hensel lifting algorithm for n factors. *Masters Thesis, Simon Fraser University*, www.cecm.sfu.ca/CAG/theses/garrett.pdf (2019).

[20] Shaw, M., and Traub, J. F. On the Number of Multiplications for the Evaluation of a Polynomial and Some of Its Derivatives. *J. ACM* ***21***, 1 (1974), 161–167.

[21] van der Hoeven, J. Faster relaxed multiplication. In *Proceedings of ISSAC '14* (2014), ACM, p. 405–412.

[22] Wang, P. An Improved Multivariate Polynomial Factoring Algorithm. *Mathematics of Computation* ***32*** (1978), 1215–1231.

[23] Wang, P. S. The EEZ-GCD Algorithm. *SIGSAM Bull.* ***14***, 2 (1980), 50–60.

[24] Wang, P. S., and Rothschild, L. P. Factoring Multivariate Polynomials Over the Integers. *Mathematics of Computation* ***29***, 131 (1975), 935–950.

[25] Zassenhaus, H. On Hensel Factorization, I. *Journal of Number Theory* ***1***, 3 (1969), 291–311.

A APPENDIX 1: MAGMA CODE FOR QHL IN $\mathbb{Z}_p[x, y]$

Reference: Algorithm 15.17 "Multifactor Hensel lifting" from [6]. We have modified it to stop when the error is 0 and to lift the solutions to the Diophantine equation to half the precision. Magma is using fast multiplication for multiplications in $\mathbb{Z}_p[x, y]$, fast division for the two divisions QuotRem(...) in $Z_p[x, y]$ and fast division for the integer divisions by m in reduce(...).

```
p := 2^31-1;
n := 1;
alpha := 3;

Fp := GaloisField(p);
Zpy<y> := PolynomialRing(Fp);
Zpxy<x> := PolynomialRing(Zpy);

function getexpons(dy,n)
   L := [];
   while #L lt Minimum(dy+1,n) do
      e := Random(dy);
      if not( e in L ) then
        L := Append(L,e); end if;
   end while;
   return L;
end function;

function getcoeff(dy,p)
   E := getexpons(dy,dy);
   C := 0;
   for e in E do C := C +
     Random(1,p-1)*y^e; end for;
   return C;
end function;

function getpoly(dx,dy,p)
   C := [ getcoeff(dy,p) :
          i in [0..dx-1] ];
   f := Zpxy!C;
   return x^dx+f;
end function;

function reduce( f, m )
   return Zpxy![ c mod m : c
               in Coefficients(f) ];
end function;

function convertytox( f )
g := Zpxy!Coefficients(f);
return g;
end function;

function DivideModm(f,g,m)
   I := ideal<Zpy|m>;
   Q := quo<Zpy|I>;
   Rx := PolynomialRing(Q);
   F := Rx!f;
   G := Rx!g;
   Q,R := Quotrem(F,G);
   q := Zpxy!Q;
   r := Zpxy!R;
   return q,r ;
end function;
```

```
function BivariateHensel(f,F,n,m,L)

   if n eq 1 then
      return [reduce(f,m^L)];
   end if;

   k := Floor(n/2);

   g := 1;
   F1 := [];
   for i := 1 to k do
      g := g*F[i];
      F1 := Append(F1,F[i]);
   end for;
   h := 1;
   F2 := [];
   for i := k+1 to n do
      h := h*F[i];
      F2 := Append(F2,F[i]);
   end for;

   G0 := Zpy!g; H0 := Zpy!h;
   gcd,S0,T0 := XGCD(G0,H0);
   s := convertytox(S0); t := convertytox(T0);

   e := f-g*h;
   K := 1;
   M := m;
   while Degree(M) le L do
      //print "Step", K, "Degree", Degree(m);
      M := M^2;

      e := reduce(e,M);
      q,r := DivideModm(s*e,h,M);
      u := t*e+q*g; u := reduce(u,M);
      g := g + u;
      h := h + r;

      e := f-g*h;
      if e eq 0 then break; end if;

      // Lift diophantine equation solutions
      b := s*g + t*h - 1; b := reduce(b,M);
      c,d := DivideModm(s*b,h,M);
      u := t*b+c*g; u := reduce(u,M);
      s := s - d;
      t := t - u;

      K := K+1;
   end while;

   F1 := BivariateHensel(g,F1,k,m,L);
   F2 := BivariateHensel(h,F2,n-k,m,L);

   H := [];
   for i := 1 to k do
      H := Append(H,F1[i]);
   end for;
   for i := 1 to (n-k) do
      H := Append(H,F2[i]);
   end for;

   return H;

end function;
```

```
alpha := 3;
m := Zpy!(y-alpha);

n := 4;
dx := 20;
dy := 20;

f1 := getpoly(dx,dy,p);
f2 := getpoly(dx,dy,p);
f3 := getpoly(dx,dy,p);
f4 := getpoly(dx,dy,p);

f := f1*f2*f3*f4;
F := [reduce(f1,m),reduce(f2,m),
      reduce(f3,m),reduce(f4,m)];

time G := BivariateHensel(f,F,n,m,2*dy);

G[1] - f1; G[2] - f2; G[3] - f3; G[4] - f4;
```