Linear Hensel Lifting for $\mathbb{Z}_p[x, y]$ and $\mathbb{Z}[x]$ with Cubic Cost

Michael Monagan

Department of Mathematics,
Simon Fraser University,
Burnaby, British Columbia V5A 1S6, Canada.
mmonagan@cecm.sfu.ca

Abstract. Hensel lifting is a key tool that is used to factor polynomials and compute polynomial GCDs in $\mathbb{Z}[x]$, $\mathbb{Z}[x_1, \ldots, x_n]$ and $\mathbb{F}_q[x_1, \ldots, x_n]$. There are two versions of Hensel lifting: Linear Hensel Lifting (LHL) and Quadratic Hensel Lifting (QHL). For polynomials in $\mathbb{Z}[x]$, if classical quadratic algorithms for $\times$ and $\div$ are used, LHL and QHL both have a quartic complexity. If asymptotically fast arithmetic is used, up to logarithmic factors, LHL is cubic and QHL is quadratic.

In this work we present cubic algorithms for LHL for $\mathbb{Z}[x]$ and $\mathbb{Z}_p[x, y]$. We present details of C implementations of our cubic algorithms for $\mathbb{Z}[x]$ and $\mathbb{Z}_p[x, y]$. We compare both with Magma implementations of QHL using fast arithmetic. For both cases, we find that our our cubic LHL outperforms Magma's fast QHL for a very wide range of input sizes.

Keywords: Hensel lifting, modular methods, polynomial factorization.

1 Introduction

Hensel lifting is one of the main tools used to factor polynomials. It was first used by Zassenhaus in [20] to factor polynomials in $\mathbb{Z}[x]$. Musser in [12, 13] and Wang and Rothschild in [15, 16] subsequently developed multivariate Hensel lifting (MHL) to factor polynomials in $\mathbb{Z}[x_1, \ldots, x_n]$. In [8], Moses and Yun applied MHL to compute the greatest common divisor of two polynomials in $\mathbb{Z}[x_1, \ldots, x_n]$. They called their algorithm the EZ-GCD algorithm.

In [17] Wang improved his factorization algorithm for $\mathbb{Z}[x_1, \ldots, x_n]$ and in [18], he improved the EZ-GCD algorithm. Wang's MHL codes are the default algorithms used in Macsyma for factorization and for GCDs in $\mathbb{Z}[x_1, \ldots, x_n]$. Wang's algorithms were implemented in Maple by Keith Geddes (see Ch. 6 of [4]) and in Magma by Allan Steel [14].

For sparse multivariate polynomials, Wang's MHL can be exponential in n the number of variables. Random polynomial time algorithms for MHL for the sparse case have been developed by Kaltofen [6] and by Monagan and Tuncer [9]. The latter has been integrated into Maple for Maple version 2019.

In this paper we are interested in the complexity of Hensel lifting when used for Hensel lifting in $\mathbb{Z}[x]$ and also in $\mathbb{Z}_p[x, y]$ for prime p. Our work is motivated by the parallel MHL algorithm of Monagan and Tuncer in [10] which reduces Hensel lifting in $\mathbb{Z}_p[x_1, \ldots, x_n]$ to many bivariate Hensel lifts in $\mathbb{Z}_p[x_1, x_2]$ which are done in parallel. In this multivariate context, the degree of the polynomials is usually under 1000 in practical applications.

Let $a \in \mathbb{Z}[x]$. For simplicity we consider the case of Hensel lifting two factors. The input to Hensel lifting is a, a lifting prime p and power $m > 0$, and two polynomials f_0 and g_0 in $\mathbb{Z}_p[x]$ satisfying (i) $a - f_0 g_0 \equiv 0 \bmod p$ and (ii) $\gcd(f_0, g_0) = 1$. The output is two polynomials $f^{(m)}$ and $g^{(m)}$ in $\mathbb{Z}[x]$ which satisfy $a - f^{(m)}g^{(m)} \equiv 0 \bmod p^m$, $f^{(m)} \equiv f_0 \bmod p$ and $g^{(m)} \equiv g_0 \bmod p$. In some applications we also require that $f^{(m)}$ and $g^{(m)}$ satisfy $a - f^{(m)}g^{(m)} = 0$ in $\mathbb{Z}[x]$. An example is when computing $g = \gcd(a, b)$ where $f = a/g$.

Hensel's Lemma says condition (ii) guarantees the existence of $f^{(m)}$ and $g^{(m)}$ for all $m > 1$. The polynomials $f^{(m)}$ and $g^{(m)}$ are unique up to multiplication by a scalar in $\mathbb{Z}$. "Hensel Lifting" refers to algorithms which compute $f^{(m)}$ and $g^{(m)}$ in a padic (base p) representation

$$f^{(k)} = \sum_{i=0}^{k-1} f_i p^i \text{ and } g^{(k)} = \sum_{i=0}^{k-1} g_i p^k$$

where f_i and g_i are in $\mathbb{Z}_p[x]$. There are two versions of Hensel Lifting, linear Hensel Lifting (LHL) and quadratic Hensel Lifting (QHL). In LHL, we recover $f^{(k+1)}$ and $g^{(k+1)}$ from $f^{(k)}$ and $g^{(k)}$ for $k = 0, 2, \ldots, m-1$. At step k, LHL computes the error $e_k = a - f^{(k)}g^{(k)}$ then computes $c_k = (e^k/p^k) \mod p$ then solves the diophantine equation $f_k g_0 + g_k f_0 = c_k$ in $\mathbb{Z}_p[x]$ for f_k and g_k with $\deg f_k < \deg f_0$. For details and proofs we refer the reader to Ch. 6 of [4].

If we assume (i) p is of bounded size, (ii) classical quadratic $O(k^2)$ algorithms are used for integer $\times$ and $\div$ and (iii) classical quadratic $O(d^2)$ algorithms are used for polynomial $\times$ and $\div$, LHL is in $O(m^3 d^2)$. Miola and Yun [7] reduce the complexity of LHL to $O(m^2 d^2)$ by avoiding recomputation of e_{k-1} at each lifting step. We give details for this in Section 2.

QHL recovers $f^{(2^{k+1})}$ and $g^{(2^{k+1})}$ from $f^{(2^k)}$ and $g^{(2^k)}$ for $k = 0, 1, 2, \ldots, \lceil \log_2 m \rceil - 1$. For details of QHL see Chapter 15 of [3]. Assuming (i), (ii) and (iii) above, the complexity of QHL is $O(m^2 d^2)$. If, however, we use fast polynomial and integer arithmetic, QHL can be done in $M(dm)$ where $M(dm)$ is the cost of multiplying integers of length $O(md)$ words. This leads to an algorithm with complexity $\tilde{O}(md)$. We cite also Bostan et. al. [2] whoe study the complexity of QHL in $\mathbb{Z}_p[x, y]$ for the multi-factor case.

In Appendix 1 we show that when fast arithmetic is used, QHL (Algorithm 15.10 in [3]) does the equivalent of 28 multiplications in $\mathbb{Z}[x]$ mod p^m to compute $f^{(m)}$ and $g^{(m)}$ from f_0 and g_0. This constant 28 means fast QHL will not beat the quartic $O(m^2 d^2)$ LHL until fairly high precision. Our data in Section 6 shows that Magma's fast QHL beats our quartic LHL first at $d = m = 200$. For this reason, both LHL and QHL are in used in practice.

In this paper we present cubic LHL algorithms for $\mathbb{Z}[x]$ and $\mathbb{Z}_p[x, y]$. For Hensel lifting $a = fg$ in $\mathbb{Z}_p[x, y]$, if $d_x = \deg(a, x)$ and $d_y = \deg(a, y)$, our algorithm does $O(d_x d_y^2 + d_x^2 d_y)$ arithmetic operations in $\mathbb{Z}_p$ (Theorem 1). For Hensel lifting of $a = fg$ in $\mathbb{Z}[x]$, if a has degree d and coefficients of size at most 10^m, our algorithm has bit complexity $O(md^2 + m^2 d)$ (Theorem 2). Because the complexities are cubic, our algorithms are not as fast asymptotically as QHL when asymptotically fast arithmetic is used. But they are very practical. The work they do is a little more than multiplying $f \times g$ using a modular method.

We have implemented our cubic algorithm in C for $\mathbb{Z}_p[x, y]$ and for $\mathbb{Z}[x]$. We have also implemented QHL in Magma for both cases (see Appendix 1 and 2). We used Magma because it is the only system with fast $\times$ and $\div$ for both cases. Our cubic algorithm beats Magma (Table 1) for all $d_x = d_y \leq 6,000$ and for

all $m = d \leq 10,000$ (Table 2) which is as big as we could go in Magma before Magma runs out of space on our 64 gigabyte machine.

Our paper is organized as follows. Section 2 gives details for Miola and Yun's version of linear Hensel lifting in $\mathbb{Z}_p[x,y]$ and shows that it has a quartic complexity. Section 3 develops a cubic LHL algorithm for $\mathbb{Z}_p[x,y]$. We start from the quartic version of Bernardin [1]. We do this because our cubic algorithm is much easier to see in this setting – this is how we found the cubic algorithm. Section 4 develops a cubic LHL for $\mathbb{Z}[x]$. Here we also give a treatment for the non-monic case. In Section 5 we discuss our C implementation for our cubic algorithm in $\mathbb{Z}_p[x,y]$ and present benchmarks comparing the quartic algorithm, our cubic algorithm, and a Magma implementation of QHL using fast arithmetic. We also present an efficient Lagrange interpolation for $\mathbb{Z}_p[x]$ using $\pm$ points. In Section 6 we discuss an implementation in C for our cubic LHL in $\mathbb{Z}[x]$ and we compare it with our C implementation of the quartic LHL of Miola and Yun [7], and our Magma implementation of QHL using fast arithmetic. In Section 7 we summarize our contribution and discuss two applications.

2 Quartic linear Hensel lifting in $\mathbb{Z}_p[x,y]$

We begin with a presentation of linear Hensel lifting (LHL) for $\mathbb{Z}_p[x,y]$. Let $a \in \mathbb{Z}_p[x,y]$ with $d_x = \deg(a,x) > 1$ and $d_y = \deg(a,y) > 1$. For simplicity we assume $a(x,y)$ is monic in x. The non-monic case is presented for Hensel lifting in $\mathbb{Z}[x]$ in Section 3.

Suppose we are given the factorization $a(x,\alpha) = f_0(x)g_0(x)$ for some $\alpha \in \mathbb{Z}_p$ such that $\gcd(f_0,g_0) = 1$ in $\mathbb{Z}_p[x]$. Suppose we are looking for a factorization $a = fg$ with $f_0 = f(x,\alpha)$ and $g_0 = g(x,\alpha)$. Because $\gcd(f_0,g_0) = 1$, Hensel's Lemma says, for $k > 1$, there exist k'th order approximations

$$f^{(k)} = \sum_{i=0}^{k-1} f_i(x)(y-\alpha)^i \quad \text{and} \quad g^{(k)} = \sum_{i=0}^{k-1} g_i(x)(y-\alpha)^i$$

with $f_i, g_i \in \mathbb{Z}_p[x]$ such that $a - f^{(k)}g^{(k)} \equiv 0 \mod (y-\alpha)^k$. Given $f^{(k)}$ and $g^{(k)}$, to compute f_k and g_k, LHL computes the error $e_k = a - f^{(k)}g^{(k)}$ then $c_k = e_k/(y-\alpha)^k \mod (y-\alpha)$ then solves the polynomial diophantine equation $f_k g_0 + g_k f_0 = c_k$ with $\deg f_k < \deg f_0$. Following Miola and Yun [7] we use

$$\begin{aligned}
\frac{e_k}{(y-\alpha)^k} &= \frac{a - f^{(k)}g^{(k)}}{(y-\alpha)^k} \\
&= \frac{a - (f^{(k-1)} + f_{k-1}(y-\alpha)^k)(g^{(k-1)} + g_{k-1}(y-\alpha)^{k-1})}{(y-\alpha)^k} \\
&= \frac{\dfrac{e_{k-1}}{(y-\alpha)^{k-1}} - f_{k-1}g^{(k-1)} - g_{k-1}f^{(k-1)} - f_{k-1}g_{k-1}(y-\alpha)^{k-1}}{y-\alpha}
\end{aligned}$$

to avoid recomputing e_{k-1}. We present the algorithm as Algorithm 1 below. In Algorithm 1, the value of *error* after Step 5 is $e_k/(y-\alpha)^k$ and the order terms on the right count arithmetic operations in $\mathbb{Z}_p$.

The most expensive step in Algorithm 1 is the computation of the error in Step 10. The worst case occurs when both factors f and g have degree $d_y/2$ in y and degree $d_x/2$ in x. The multiplications $f_k g$ and $g_k f$ cost $O(d_x(kd_x))$

Algorithm 1 Quartic Bivariate Hensel Lifting for $\mathbb{Z}_p[x,y]$: Monic Case.

Input: prime p, $\alpha \in \mathbb{Z}_p$, $a \in \mathbb{Z}_p[x,y]$ and $f_0, g_0 \in \mathbb{Z}_p[x]$ satisfying
(i) a, f_0, g_0 are monic in x, (ii) $a(y=\alpha) = f_0 g_0$ and (iii) $\gcd(f_0, g_0) = 1$.

Output: $f, g \in \mathbb{Z}_p[x,y]$ such that $a = fg$ or FAIL.

1: $d_x \leftarrow \deg(a,x)$; $d_y \leftarrow \deg(a,y)$; $f \leftarrow f_0$; $g \leftarrow g_0$.
2: $error \leftarrow a - f_0\, g_0$. ..... $O(d_x^2 + d_x d_y)$
3: Solve $sg_0 + tf_0 = 1$ using the extended Euclidean algorithm. ..... $O(d_x^2)$
4: **for** $k = 1, 2, 3, \ldots$ **while** $\deg(f,y) + \deg(g,y) < d_y$ **do**
5: $\quad error \leftarrow error/(y-\alpha)$ ..... $O(d_x d_y)$
6: $\quad c_k \leftarrow error(y=\alpha)$ ..... $O(d_x d_y)$
7: $\quad$ **if** $c_k \neq 0$ **then**
8: $\quad\quad$ // *Solve* $f_k g_0 + g_k f_0 = c_k$ *in* $\mathbb{Z}_p[x]$ *for* $f_k, g_k \in \mathbb{Z}_p[x]$ *with* $\deg f_k < \deg f_0$
9: $\quad\quad f_k \leftarrow (sc_k)$ rem f_0; $g_k \leftarrow (c_k - f_k g_0)$ quo f_0. ..... $O(d_x^2)$
10: $\quad\quad error \leftarrow error - f_k g - g_k f - f_k g_k (y-\alpha)^k$ ..... $O(k d_x^2)$
11: $\quad\quad f \leftarrow f + f_k (y-\alpha)^k$; $g \leftarrow g + g_k (y-\alpha)^k$. ..... $O(k d_x)$
12: $\quad$ **end if**
13: **end for**
14: **if** $error = 0$ **then** return f, g **else** return FAIL **end if**

and the multiplication $f_k g_k (y-\alpha)^k$ costs $O(d_x^2 + k d_x)$. In total Step 10 does $\sum_{k=1}^{d_y/2} O(k d_x^2) = O(d_y^2 d_x^2)$ arithmetic operations in $\mathbb{Z}_p$ in the worst case. Note, to establish that the lifted factors f and g satisfy $a = fg$ Hensel lifting must multiply f by g, which, in the classical quadratic model, is also $O(d_x^2 d_y^2)$.

3 Cubic Hensel Lifting in $\mathbb{Z}_p[x,y]$

To obtain an algorithm with cubic complexity we first reorganize the order in which we compute the products $f_i g_j$ in $\mathbb{Z}_p[x]$ in Algorithm 1. Figure 1 shows at which iteration k Algorithm 1 computes $f_i g_j$ and Algorithm 2 computes $f_i g_j$.

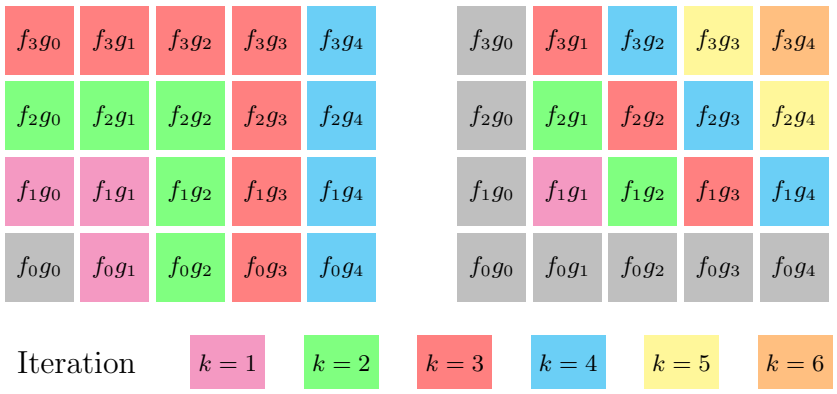

Fig. 1. shows at which iteration k Algorithm 1 (left) and Algorithm 2 (right) multiplies $f_i \times g_j$ for an example where $\deg(f,y) = 3$ and $\deg(g,y) = 4$. Gray squares $f_i g_j$ are not explicitly multiplied.

The key is that in order to compute f_k and g_k at step k, we only need c_k, the coefficient of $a - f^{(k)}g^{(k)}$ in $(y-\alpha)^k$. Let $a = \sum_{k=0}^{\deg(a,y)} a_k(x)(y-\alpha)^k$ be the Taylor representation of $a(x,y)$. One way to compute a_k efficiently is to use the formula $a_k = a^{(k)}(\alpha)/k!$ where $a^{(k)} = \partial^k a(x,y)/\partial y^k$. Alternatively, in Algorithm 2, we use polynomial long division by $(y-\alpha)$ to obtain the a_k.

At Step 6 let $df = \deg(f^{(k)}, y)$ and $dg = \deg(g^{(k)}, y)$. We have $df < k$ and $dg < k$ and

$$c_k = \text{coeff}\left(a - f^{(k)}g^{(k)}, (y-\alpha)^k\right) = a_k - \sum_{i=\max(0,k-dg)}^{\min(k,df)} f_i g_{k-i}.$$

Now let $n = \deg(a,y)$. When the iteration terminates at $k = n$ we need to know if the error $a - f^{(n)}g^{(n)} = 0$. Let $e_k = \text{coeff}(a - f^{(n)}g^{(n)}, (y-\alpha)^k)$. Then $a - f^{(n)}g^{(n)} = 0 \iff e_k = 0$ for $0 \le k \le n$. We have the following relationship between the c_k which we compute and e_k:

$$e_k = a_k - \sum_{i=0}^{k} f_i g_{k-i} = c_k - (f_0 g_k + g_0 f_k).$$

Thus it appears that we should compute $c_k - (f_0 g_k + g_0 f_k)$ and test if it is zero. But, after computing c_k we solve $f_k g_0 + g_k f_0 = c_k$ for f_k and g_k so we already know that these e_k are zero. So the two multiplications $f_0 g_k$ and $g_0 f_k$, shown in gray in Figure 1, may be omitted. This leads to Algorithm 2 for LHL.

Algorithm 2 Cubic Hensel Lifting for $\mathbb{Z}_p[x,y]$: Monic Case.

Input: prime p, $\alpha \in \mathbb{Z}_p$, $a \in \mathbb{Z}_p[x,y]$ and $f_0, g_0 \in \mathbb{Z}_p[x]$ satisfying
(i) a, f_0, g_0 are monic in x, (ii) $a(y=\alpha) = f_0 g_0$ and (iii) $\gcd(f_0, g_0) = 1$.

Output: $f, g \in \mathbb{Z}_p[x,y]$ such that $a = fg$ or FAIL $\Rightarrow$ no such f, g exist.

1: $d_x \leftarrow \deg(a,x)$; $d_y \leftarrow \deg(a,y)$; $df \leftarrow 0$; $dg \leftarrow 0$.
2: Solve $sg_0 + tf_0 = 1$ using the extended Euclidean algorithm. ..... $O(d_x^2)$
3: $a \leftarrow a$ quo $y - \alpha$ // Note (ii) implies $a - f_0 g_0$ rem $y - \alpha = 0$.
4: **for** $k = 1$ **to** d_y **do**
5: $\quad a_k \leftarrow a$ rem $(y-\alpha)$; $a \leftarrow a$ quo $(y-\alpha)$. ..... $O(d_x d_y)$
6: $\quad c_k \leftarrow a_k - \sum_{i=\max(1,k-dg)}^{\min(k-1,df)} f_i g_{k-i}$ ..... $O(kd_x^2)$
7: $\quad$ **if** $df + dg = d_y$ and $c_k \neq 0$ **then**
8: $\quad\quad$ return FAIL // $e_k \neq 0$ so the error is not zero
9: $\quad$ **end if**
10: $\quad$ // *Solve* $f_k g_0 + g_k f_0 = c_k$ *in* $\mathbb{Z}_p[x]$ *for* $f_k, g_k \in \mathbb{Z}_p[x]$ *with* $\deg f_k < \deg f_0$
11: $\quad f_k \leftarrow (sc_k)$ rem f_0; $g_k \leftarrow (c_k - f_k g_0)$ quo f_0. ..... $O(d_x^2)$
12: $\quad$ **if** $f_k \neq 0$ **then** $df \leftarrow k$; **end if**
13: $\quad$ **if** $g_k \neq 0$ **then** $dg \leftarrow k$; **end if**
14: **end for**
15: $f \leftarrow \sum_{i=0}^{df} f_i (y-\alpha)^i$. ..... $O(df\, d_x^2) \subset O(d_y d_x^2)$
16: $g \leftarrow \sum_{i=0}^{dg} g_i (y-\alpha)^i$. ..... $O(dg\, d_x^2) \subset O(d_y d_x^2)$
17: return f, g.

Although not explicit, Algorithm 2 works in two phases. In the first phase, while $df + dg < \deg(a,y)$, it is computing the coefficients f_k and g_k. In the second phase, after $dg + dg = \deg(a,y)$, it is checking that the coefficients e_k of the error

are 0 for $k \geq \max(df, dg)$. Referring to Figure 1, in the first phase Algorithm 2 computes $c_1 = a_1$ then $c_2 = a_2 - f_1 g_1$ then $c_3 = a_3 - (f_1 g_2 + f_2 g_1)$ then $c_4 = a_3 - (f_3 g_1 + f_2 g_2 + f_1 g_3)$. Now f and g are determined. In the second phase Algorithm 2 computes $c_5 = a_5 - (f_3 g_2 + f_2 g_3 + f_1 g_4)$ then $c_6 = a_6 - (f_3 g_3 + f_2 g_4)$ then finally $c_7 = a_7 - f_3 g_4$.

Observe that in total, Steps 5, 15 and 16 of Algorithm 2 cost $O(d_x d_y^2)$ arithmetic operations in $\mathbb{Z}_p$ and Step 11 costs $O(d_x^2 d_y)$. The most expensive part of Algorithm 2 is the sum

$$\Delta(x) = \sum_{i=\max(1,k-dg)}^{\min(k-1,df)} f_i g_{k-i}$$

in Step 6. The worst case occurs when $\deg(f, y) = \deg(g, y) = d_y/2$, which maximizes the cost of computing the product fg, so that the cost of the polynomial multiplications for computing $\Delta(x)$ is $\sum_{k=1}^{d_y/2}(k-1)O(d_x^2) = O(d_y^2 d_x^2)$. So Algorithm 2 is quartic.

In [1], Bernardin sped up the computation of $\Delta(x)$ by computing the products $f_i g_{k-i}$ in parallel. Bernardin also tried using fast multiplication for each $f_i g_{k-i}$ – he used Karatsuba. Instead, we obtain a natural cubic algorithm by using classical evaluation and interpolation with a memory as follows.

To interpolate $c_k(x)$, since $\deg(c_k, x) < \deg(a, x)$ (this follows from a, f_0, g_0 being monic in x) d_x evaluation points are sufficient. We replace Step 6 with

6 **if** $k > 1$ **then**
6a: $f_{k-1,j} \leftarrow f_{k-1}(x = j)$ for $0 \leq j \leq d_x - 1$.
$g_{k-1,j} \leftarrow g_{k-1}(x = j)$ for $0 \leq j \leq d_x - 1$.
6b: $\Delta_j \leftarrow \sum_{i=\max(1,k-dg)}^{\min(k-1,df)} f_{i,j} \times g_{k-i,j}$ for $0 \leq j \leq d_x - 1$.
6c: Interpolate $\Delta(x)$ from points $(j, \Delta_j) : 0 \leq j \leq d_x - 1$.
6d: $c_k \leftarrow a_k - \Delta(x)$.
else $c_k \leftarrow a_k$.

We must remember the previous evaluations of $f_1, f_2, \ldots, f_{k-2}$ and $g_1, g_2, \ldots, g_{k-2}$ for re-use in Step 6b. The space needed for this is $O(dxdy)$ elements of $\mathbb{Z}_p$ which is of the same order as a dense input $a(x, y)$. Using Horner evaluation in $\mathbb{Z}_p[x]$ Step 6a is $O(d_x^2)$ multiplications and additions in $\mathbb{Z}_p$. Step 6b is $O(kd_x)$ multiplications and additions. For Step 6c we may use either Newton interpolation or Lagrange interpolation both of which are $O(d_x^2)$ arithmetic operations in $\mathbb{Z}_p$. The subtraction in Step 6d is in $\mathbb{Z}_p[x]$ so it costs $O(d_x)$. Summing for $k = 1$ to d_y we have Steps 6a, 6b, 6c and 6d cost $O(d_x^2 d_y)$, $O(d_x d_y^2)$, $O(d_x^2 d_y)$ and $O(d_x d_y)$ respectively. Thus we have the following result.

Theorem 1. *Algorithm 2 does $O(d_x d_y^2 + d_x^2 d_y)$ arithmetic operations in $\mathbb{Z}_p$.*

Remark 1. If $p \geq d_x$ does not hold one may use a small field extension; pick the smallest j such that $q = p^j \geq d_x$ so that the field $\mathbb{F}_q$ with q elements has enough elements for evaluation and interpolation.

4 Cubic Hensel Lifting in $\mathbb{Z}[x]$

Let $a = \sum_{i=0}^{d} a_i x^i$ be a polynomial in $\mathbb{Z}[x]$ with degree $d > 1$. Let $p > 2$ be a prime that does not divide $\mathrm{lc}(a) = a_d$ the leading coefficient of a. The choice of

p for Hensel lifting depends on the application. For the GCD problem, where we want to compute $\gcd(a,b)$ for $a,b \in \mathbb{Z}[x]$, one would choose a word size prime e.g. 63 bits on a 64 bit computer.

Suppose we are given $f_0, g_0 \in \mathbb{Z}_p[x]$ that satisfy (i) $a - f_0 g_0 \equiv 0 \mod p$ and (ii) $\gcd(f_0, g_0) = 1$ and we are trying to find factors $f, g \in \mathbb{Z}[x]$ such that $a = fg$ and $f \mod p = f_0$ and $g \mod p = g_0$, if they exist.

Let $f^{(k)} = f_0 + f_1 p + \cdots + f_{k-1} p^{k-1}$ and $g^{(k)} = g_0 + g_1 p + \cdots + g_{k-1} p^{k-1}$ be $k'th$ order approximations of f and g, i.e., $f_i, g_i \in \mathbb{Z}_p[x]$ and $a - f^{(k)} g^{(k)} \equiv 0 \mod p^k$. To obtain $k+1$st order approximations we may compute the error $e_k = a - f^{(k)} g^{(k)}$, then compute $c_k = (e_k / p^k) \mod p$ then solve the polynomial diophantine equation $f_k g_0 + g_k f_0 = c_k$ for $f_k, g_k \in \mathbb{Z}_p[x]$ with $\deg(f_k) < \deg(f_0)$. For details see Chapter 6 of [4].

If we need to lift m steps to $f^{(m)}$ and $g^{(m)}$, assuming classical quadratic multiplication in $\mathbb{Z}$ and $\mathbb{Z}_p[x]$, this leads to an algorithm with complexity $O(m^3 d^2)$. To obtain an $O(m^2 d^2)$ algorithm, Miola and Yun [7] reuse e_{k-1} in the computation of e_k to avoid recomputation. For $k > 1$ they use

$$\frac{e_k}{p^k} = \frac{a - f^{(k)} g^{(k)}}{p^k} = \frac{e_{k-1}/p^{k-1} - f_{k-1} g^{(k-1)} + g_{k-1} f^{(k-1)} - f_k g_k p^{(k-1)}}{p}.$$

Observe that we do not need the terms $f_{k-1}(g_2 p^2 + \dots)$ nor $g_{k-1}(f_2 p^2 + \dots)$ nor $f_k p_k p^{k-1}$ for $k > 2$ to determine c_k. To obtain a cubic algorithm, for $k > 1$ we will use

$$\frac{e_k}{p^k} = \frac{e_{k-1}/p^{k-1} - f_{k-1} g_0 - g_{k-1} f_0}{p} - \sum_{i=1}^{k-1} f_i g_{k-i}$$

so that $\frac{e_2}{p^2} = \frac{e_1/p - f_1 g_0 - g_1 f_0}{p} - f_1 g_1$ and $\frac{e_3}{p^3} = \frac{e_2/p^2 - f_2 g_0 - g_2 f_0}{p} - f_1 g_2 - f_2 g_1$. The algorithm is presented as Algorithm 3. As in the $\mathbb{Z}_p[x,y]$ case, the sum $\Delta = \sum_{i=1}^{k-1} f_i g_{k-i}$ is the computational bottleneck and, after f and g have been determined, we must continue to complete the multiplication of fg to show that $a - fg = 0$. In Algorithm 3 we have done this in a separate **while** loop. We also need a bound B on the height of the factors f and g of a. For this one may use the Mignotte bound – see [3]. Unlike the $\mathbb{Z}_p[x,y]$ case, because of carries, we must explicitly multiply $f_{k-1} g_0$ and $g_{k-1} f_0$.

We also include a treatment of the non-monic case in Algorithm 3. We assume the input polynomial $a(x)$ is primitive, i.e., $\mathit{content}(a) = \gcd(a_0, a_1, \dots, a_d) = 1$. Let $\gamma = \mathrm{lc}(a) = a_d$ be the leading coefficient of $a(x)$. Following [4] we use the "replace leading coefficient trick" where we attach γ to f_0 and g_0. This means that if $\alpha = \mathrm{lc}(f)$ and $\beta = \mathrm{lc}(g)$ so that $\gamma = \alpha\beta$, the algorithm computes $f^{(nf)} = \beta f$ and $g^{(ng)} = \alpha g$ for some nf and ng and stops when $\gamma a - f^{(nf)} g^{(ng)} = 0$.

In Algorithm 3 the **mods** operation means use the symmetric remainder, and not the normal remainder. This is necessary to recover negative coefficients in f and g. It is also why the prime p cannot be 2. Given $c \in \mathbb{Z}$, for $p > 2$, the symmetric remainder satisfies $c = pq + r$ for a quotient q and remainder $-\frac{p-1}{2} \le r \le \frac{p-1}{2}$. In the algorithm, where we apply it to polynomials in $\mathbb{Z}[x]$ we mean apply it to each coefficient. For example, for $a = 9x^2 + 40x + 11$ and $p = 7$ we have $\mathbf{mods}(a, p) = 2x^2 - 2x - 3$.

The bottleneck of Algorithm 3 is the computation of Δ in Steps 14 and 20. Here f_i and g_j are in $\mathbb{Z}_p[x]$ with $\deg f_0 + \deg g_0 = d$ and $\deg f_k + \deg g_k < d$ for $k > 0$. Using classical multiplication this costs $O(kd^2)$. The total work is $\sum_{k=1}^{m} O(kd^2) = O(m^2 d^2)$.

Algorithm 3 Cubic Linear Hensel Lifting for $\mathbb{Z}[x]$: General Case.

Input: prime $p > 2$, $a \in \mathbb{Z}[x]$ with $d = \deg a > 1$ and $f_0, g_0 \in \mathbb{Z}_p[x]$ satisfying (i) $content(a) = 1$, (ii) $\gamma \bmod p \neq 0$, (iii) $a - f_0 g_0 \equiv 0 \mod p$ and (iv) $\gcd(f_0, g_0) = 1$. Also a lifting bound $B > \max(\|f\|, \|g\|)$. Let γ be the leading coefficient of a and let $m = \lceil \log_p(2\gamma B) \rceil$. m is the maximum number of lifting steps needed to recover both factors.

Output: $f, g \in \mathbb{Z}[x]$ such that $a = fg$ or FAIL $\Rightarrow$ no such f, g exist.

1: $\gamma \leftarrow \text{lc}(a)$; $a \leftarrow \gamma a$; ..... $O(m^2 d)$
2: $f_0 \leftarrow \mathbf{mods}(\gamma\, \text{lc}(f_0)^{-1} f_0, p)$; Replace $\text{lc}(f_0)$ with γ so that $f_0 = \gamma x^{df} + \ldots$
3: $g_0 \leftarrow \mathbf{mods}(\gamma\, \text{lc}(g_0)^{-1} g_0, p)$; Replace $\text{lc}(g_0)$ with γ so that $g_0 = \gamma x^{dg} + \ldots$
4: Solve $sg_0 + tf_0 = 1$ using the extended Euclidean algorithm ..... $O(d^2)$
5: $T \leftarrow f_0 g_0$; $e \leftarrow (a - T)/p$; ..... $O(d^2 + m^2 + md) + O(md)$.
6: Initialize $k \leftarrow 1$; $nf \leftarrow 0$; $ng \leftarrow 0$;
7: **while** $p^k < 2B\gamma$ and $e \neq 0$ **do**
8: $c_k \leftarrow e \mod p$; ..... $O(md)$
9: // *Solve* $f_k g_0 + g_k f_0 = c_k$ *in* $\mathbb{Z}_p[x]$ *for* $f_k, g_k \in \mathbb{Z}_p[x]$ *with* $\deg f_k < \deg f_0$
10: $f_k \leftarrow (sc_k) \text{ rem } f_0$; $g_k \leftarrow (c_k - f_k g_0) \text{ quo } f_0$; ..... $O(d^2)$
11: **if** $f_k \neq 0$ **then** $nf \leftarrow k$; $f_k = \mathbf{mods}(f_k, p)$; **end if**
12: **if** $g_k \neq 0$ **then** $ng \leftarrow k$; $g_k = \mathbf{mods}(g_k, p)$; **end if**
13: $C \leftarrow f_0 g_k + g_0 f_k$; ..... $O(d^2 + md)$
14: $\Delta \leftarrow \sum_{i=1}^{k} f_i g_{k+1-i}$; ..... $O(kd^2)$
15: $e \leftarrow (e - C)/p - \Delta$; ..... $O(md)$
16: $k \leftarrow k + 1$;
17: **end while**
18: **while** $k < nf + ng$ **do**
19: **if** $e \mod p \neq 0$ **then** return FAIL; **end if**
20: $\Delta \leftarrow \sum_{i=k+1-ng}^{nf} f_i g_{k+1-i}$; ..... $O(kd^2)$
21: $e \leftarrow e/p - \Delta$; ..... $O(md)$
22: $k \leftarrow k + 1$;
23: **end while**
24: **if** $e \neq 0$ **then** return FAIL; **end if**
25: $f \leftarrow \sum_{i=0}^{nf} f_i p^i$; $f \leftarrow f/content(f)$; ..... $O(m^2 d)$
26: $g \leftarrow \sum_{i=0}^{ng} g_i p^i$; $g \leftarrow g/content(g)$; ..... $O(m^2 d)$
27: return f, g

Let $a = \sum_{i=0}^{d} a_i x^i$ with $a_i \in \mathbb{Z}$. Let $\|a\|$ denote the height of a, that is, $\|a\| = \max_{i=0}^{d} |a_i|$. Since the coefficients of f_i and g_i are in the range $[-\frac{p-1}{2}, +\frac{p-1}{2}]$, we have $\|\Delta\| < k\frac{d}{2}(\frac{p}{2})^2 \leq \frac{mdp^2}{8}$. We propose to compute Δ modulo primes $q_1, q_2, \ldots$ and use Chinese remaindering. If q is the smallest prime $\lceil \log_q \frac{mdp^2}{4} \rceil$ primes are sufficient. If $p < 2^{64}$ and $md < 2^{60}$, which will be the case in practice, then three 63 bit primes are sufficient. For each prime we use the same evaluation/interpolation strategy that we used to make Algorithm 2 have cubic complexity. So at step k we compute Δ modulo q_i in $O(kd)$ arithmetic operations modulo q_i.

Note that the products $f_0 g_k$ and $g_0 f_k$ in Step 13 are of the form $(\gamma x^{df} + \Delta_{f0}) g_k$ and $(\gamma x^{dg} + \Delta_{g0}) f_k$ where the coefficients of $\Delta_{f0}, f_k, \Delta_{g0}$ and g_k are mod p. Since γ may be the largest coefficient of a then $\log_p(\gamma) \in O(m)$ so these two products cost $O(md + d^2)$. Summing $\sum_{k=1}^{m}$ this is $O(m^2 + md^2)$. Thus we have established the following result.

Theorem 2. *If* $\deg a = d$ *and* p *is of bounded size and the primes used for the Chinese remaindering are at least* q *then we can compute* $f = f^{(m)}$ *and* $g = g^{(m)}$ *in* $O(m^2 d \log_q(md) + md^2)$. *Moreover, if* $q^2 > md$, *which will always be the case in practice if we use primes of 50 bits or more, then we can compute* f *and* g *in* $O(m^2 d + md^2)$.

5 Implementation and Benchmarks for $\mathbb{Z}_p[x,y]$

Table 1 shows timings for Hensel lifting in $\mathbb{Z}_p[x,y]$ for three algorithms for $p = 2^{31}-1$. The timings were obtained using a sever with 64 gigabytes of RAM and two Intel Xeon E5 2680 processors running at 2.2GHz base and 3.0GHz turbo.

In Table 1, d is the degree of both factors in both variables, i.e., d_x=d_y=d. The factors f and g have the form $x^d + \sum_{i=0}^{d-1} \sum_{j=1}^{3} c_{ij} y^{e_{ij}} x^i$ where the coefficients c_{ij} are chosen at random from $[0,p)$ and the exponents e_{ij} are chosen at random from $[0,d]$. We then input $\alpha = 3$, $a = f \times g$, $f_0 = f(x,3)$ and $g_0 = g(x,3)$ to Hensel lifting.

The second column labelled "Linear Lift" is for the quartic $O(d_x^2 d_y^2)$ algorithm. The third column labelled "New1" is for our new cubic $O(d_x^2 d_y + d_x d_y^2)$ algorithm where we used Horner evaluation and Newton interpolation to compute the sum in Step 6. The fourth column labelled "New2" is also for our new cubic algorithm but we have made several optimizations discussed later in this section. The fifth column labelled Magma is for a Magma implementation of QHL in $\mathbb{F}_p[x,y]$. The Magma code used is given in Appendix 2. Magma uses fast multiplication in $\mathbb{F}_p[x,y]$ and fast division in $\mathbb{F}_p[x,y] \mod (y-\alpha)^k$.

	Linear Lift	New $O(d^3)$ Linear Lift		Fast Quadratic	
d	$O(d^4)$	New1	New2	in Magma	(#steps)
10	0.14ms(.04)	0.22ms(0.14)	0.17ms(0.07)	10.9ms	(4)
15	0.35ms(.19)	0.57ms(0.37)	0.34ms(0.11)	35.7ms	(4)
20	0.75ms(.46)	1.23ms(0.85)	0.63ms(0.25)	48.9ms	(5)
40	6.58ms(5.09)	8.57ms(5.49)	3.22ms(0.98)	244ms	(6)
60	26.7ms(22.2)	27.7ms(22.2)	8.70ms(2.56)	464ms	(6)
100	166ms(148)	126ms(103)	34.2ms(10.2)	1.59s	(7)
200	2.15s(2.03)	992ms(834)	230ms(65ms)	8.73s	(8)
400	29.5s(28.5)	7.91s(6.71)	1.63s(0.49)	45.7s	(9)
800	425s(418)	63.4s(53.8)	13.9s(3.87)	273.8s	(1.13gb) (10)
1000	1017s(1003)	125s(109)	26.7s(7.41)	391.7s	(1.48gb) (10)
1500	5135.4s	382.6s	87.2s (0.39gb)	1195.8s	(4.17gb) (11)
2000	NA	1000s	207s (0.69gb)	1808.5s	(5.81gb) (11)
3000	NA	3348s	709s (1.55gb)	6230.7s	(17.1gb) (12)
4000	NA	NA	1704s (4.35gb)	9255.7s	(22.8gb) (12)
6000	NA	NA	5438s (6.20gb)	36500.s	(68.2gb) (13)
8000	NA	NA	13035s (11.0gb)	NA	

Table 1. Hensel lifting timings for $\mathbb{Z}_p[x,y]$ with $p = 2^{31}-1$. NA = Not attempted.

As the reader can see our cubic algorithm (column New1) does not beat the quartic algorithm until $d > 60$. At degree $d = 1000$ the New1 beats the quartic algorithm by a factor of 1017/125=8.1. In comparison with QHL, New1 beats Magma's QHL by a factor of $1.59/0.126 = 12.6$ at $d = 100$ and $391.7/125 = 3.1$

at $d = 1000$. This is a good result but we had hoped that we would beat the $O(d^4)$ method earlier. Shown in Table 1 in () in columns New1 and New2 is the time spent evaluating f_k and g_k and interpolating c_k. For example, for $d = 100$ we spent 103/126=82% doing this. In what follows we will reduce this to obtain the timings in column New2. Here at $d = 100$ we spent 10.2/34.2=30% in evaluation and interpolation. New2 beats the classical method at $d = 15$ which is an excellent result. New2 beats Magma by a factor of 1.59/0.034=47 at $d = 100$ and a factor of 391.7/26.7=14.7 at $d = 1000$. At $d = 6000$, which is as far as we could go with Magma because of the space it is using, Magma's fast method has still not caught our cubic method. Thus New2 is the fastest method for $d \geq 15$.

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

It is well known that hardware integer division instructions are much slower than hardware integer multiplication instructions. In [11] 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 New1 timings in Table 1 we are using Möller and Granlund's improved algorithm from [5]. We find that the time to multiply $f \times g$ in $\mathbb{Z}_p[x]$ for $p = 2^{62} - 57$ and $\deg f = \deg g = 10000$ on an Intel Core i7 2600 was 2.46s using the hardware division, and 0.407s using Möller and Granlund for a speedup of a factor of 6. 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 further significant speedup of a factor of 3.5 by using a double precision accumulator to reduce the number of divisions to one. To illustrate, the C code below does this for a 31 bit prime using a 64 bit accumulator.

```
# define I64 long long int
int dotproductp ( int *A, int *B, int n, int p ) {
   // compute ( A[0] B[0] + A[1] B[1] + ... + A[n-1] B[n-1] ) mod p
   // assumes 1 < p < 2^31  and 0 <= A[i], B[i] < p
   int i; I64 m,z;
   m = p; m = m << 32; // m = 2^32 p
   z = m;
   for( i=0; i<n; i++ ) {
      z -= (I64) A[i] * B[i];
      z += (z >> 63) & m; // if( z<0 ) z += m;
   }
   z = (-z) % p;
   if( z<0 ) z += p;
   return z;
}
```

For a 63 bit prime we have implemented this in assembler using the 64 bit by 64 bit multiply and 128 bit addition. The time for the above polynomial multiplication was reduced to 0.115s, for a speedup of another factor of 3.5.

The timings in column New2 in Table 1 are for a reorganization of the evaluation and interpolation algorithm so that we can use the accumulator option. We also $0, \pm 1, \pm 2, \ldots, \pm\frac{d}{2}$ for the evaluation points which allows us to speed up evaluation and interpolation by a further factor of 2. Because the prime $p = 2^{31} - 1$ used is smaller than the biggest possible, we can unroll the loop yielding a further improvement. The total improvement is a factor of 103/10.2=10 for $d = 100$.

Let $c(x)$ be the polynomial we wish to interpolate and let $d = \deg(c(x))$. In what follows we assume d is even; if not, we add 1 to d and use an additional evaluation point.

It is easy to evaluate a polynomial $c(x)$ in $\mathbb{Z}_p[x]$ twice as fast using $\pm$ points. Let $c(x) = \sum_{i=0}^{d} c_i x^i$. Write $c(x) = a(x^2) + xb(x^2)$ where $a(x) = \sum_{i=0}^{d/2} c_{2i} x^i$ and $b(x) = \sum_{i=0}^{d/2-1} c_{2i+1} x^i$. 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 more subtraction. To also use the accumulator trick we compute $a(\alpha^2)$ via the dot product $[a_0, a_2, a_4, \ldots, a_d] \cdot [1, \alpha^2, \ldots, \alpha^d]$ and $b(\alpha^2)$ via the dot product $[a_1, a_3, \ldots, a_{d-1}] \cdot [1, \alpha^2, \ldots, \alpha^{d-2}]$. For $\alpha = i$, the arrays $[1, i^2, i^4, \ldots, i^d]$ for $i = 1, 2, \ldots, d/2$ 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

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

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 $\alpha_i = c(i)/L_i(i)$. Furtherore, since $L_i(i) = (d-i)!i!(-1)^{(d-i)}$ we can compute α_i with 2 multiplications assuming we have already computed the inverses of $i!$ for $1 \le i \le d$.

Let $L_i(x) = \sum_{j=0}^{d} L_{ij} x^j$ and $L_i = [L_{i0}\ L_{i1}\ L_{i2} \ldots L_{id}]$. In matrix vector form we can compute the coefficients c_i of $c(x)$ using

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_d \end{bmatrix} = \begin{bmatrix} | & | & | & & | & | \\ L_0 & L_1 & L_{-1} & \ldots & L_{d/2} & L_{-d/2} \\ | & | & | & & | & | \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \alpha_{-1} \\ \vdots \\ \alpha_{d/2} \\ \alpha_{-d/2} \end{bmatrix}$$

First note that $L_i(0) = 0$ for all $i \ne 0$ so $c_0 = \alpha_0 L_{00}$. Because d is even we have $L_{-i}(x) = L_i(-x)$ so we only need compute $L_i(x)$ for $0 \le i \le d/2$. For even i we can compute c_i using $(\frac{d}{2}+1)\frac{d}{2}$ multiplications as follows.

$$\begin{bmatrix} c_2 \\ c_4 \\ c_6 \\ \vdots \\ c_d \end{bmatrix} = \begin{bmatrix} L_{02} & L_{12} & L_{22} & \ldots & L_{\frac{d}{2}2} \\ L_{04} & L_{14} & L_{24} & \ldots & L_{\frac{d}{2}4} \\ L_{06} & L_{16} & L_{26} & \ldots & L_{\frac{d}{2}6} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ L_{0\frac{d}{2}} & L_{1\frac{d}{2}} & L_{2\frac{d}{2}} & \ldots & L_{\frac{d}{2}\frac{d}{2}} \end{bmatrix} \begin{bmatrix} \alpha_1 + \alpha_2 \\ \alpha_3 + \alpha_4 \\ \alpha_5 + \alpha_6 \\ \vdots \\ \alpha_{\frac{d}{2}-1} + \alpha_{\frac{d}{2}} \end{bmatrix}$$

Similarly for odd i we can compute c_i using

$$\begin{bmatrix} c_1 \\ c_3 \\ c_5 \\ \vdots \\ c_{d-1} \end{bmatrix} = \begin{bmatrix} L_{01} & L_{11} & L_{21} & \ldots & L_{\frac{d}{2}1} \\ L_{03} & L_{13} & L_{23} & \ldots & L_{\frac{d}{2}3} \\ L_{05} & L_{15} & L_{25} & \ldots & L_{\frac{d}{2}5} \\ \ldots & \ldots & \ldots & \ldots & \ldots \\ L_{0\frac{d}{2}-1} & L_{1\frac{d}{2}-1} & L_{2\frac{d}{2}-1} & \ldots & L_{\frac{d}{2}\frac{d}{2}-1} \end{bmatrix} \begin{bmatrix} \alpha_1 - \alpha_2 \\ \alpha_3 - \alpha_4 \\ \alpha_5 - \alpha_6 \\ \vdots \\ \alpha_{\frac{d}{2}-1} - \alpha_{\frac{d}{2}} \end{bmatrix}$$

Thus we can determine $c_1, c_2, \ldots, c_d$ using $d(\frac{d}{2}+1)$ multiplications. Crucially, if we do all these multiplications as dot products of vectors, we can use our accumulator optimization if the matrices are constructed in row major order.

We compute $L_i(x)$ for $0 \le i \le d/2$ by first computing $L(x)$ then dividing $L(x)$ by $x-i$ using polynomial division. The two matrices formed from $L_i(x)$ for $0 \le i \le d/2$ and the inverses of the factorials are computed once before the main Hensel lifting loop and reused in the loop.

6 Implementation and Benchmarks for $\mathbb{Z}[x]$

Our implementation of Algorithm 3 assumes the lifting prime $p < 2^{63}$. To compute Δ in Steps 14 and 20 we use three 63 bit primes q_1, q_2, q_3. For evaluation of f_k and g_k and interpolation of $\Delta \mod q_i$, since we may choose q_1, q_2, q_3 we choose them of the form $q_i = 2^{30}s_i + 1$ so that we can use an FFT for evaluation and interpolation since this is faster.

In Algorithm 3, since e is initialized to include the input polynomial a, all computations involving e in Steps 5, 8, 13, 14, 15, 19, 20, and 21, will involve large integers. If we use a package like GMP, this would complicate memory management and cause a slowdown. In Algorithm 4 we have reorganized Algorithm 3 for the monic case to eliminate all multi-precision arithmetic with e. First we compute an explicit padic representation of $a = \sum_{i=0} a_i(x)p^k$ (see Steps 3 and 4) and include the coefficient a_k in e when needed to determine c_k in Step 8.

We seek to bound $\|e\|$, the height of e at Step 9 and 20. The heights $\|C\|$ and $\|\Delta\|$ are maximized when $\deg(f,x)=\deg(g,x)=\frac{d}{2}$. We have $\|C\| < 2\frac{d}{2}(\frac{p}{2})^2 = \frac{2dp^2}{4}$ and $\|\Delta\| < k\frac{d}{2}(\frac{p}{2})^2 \le \frac{mdp^2}{8}$ where m is the maximum number of lifting steps. Since in Step 16, we divide $e - C$ by p before subtracting Δ, after Step 16 the height of e will be a little larger than $\|\Delta\|$, and certainly no larger than twice $\frac{mdp^2}{8}$. If $p < 2^{63}$ then three 63 bit primes q_1, q_2, q_3 will be sufficient in most cases to recover the coefficients of e using Chinese remaindering at Step 9 and 20. Our idea is to do all computations involving C, Δ and e with values. If we use an FFT we need n values where $n = 2^k$ and $n \ge d$.

Let ω_j have order $n \bmod q_j$. We need the values $f_i(\omega_j^r)$, $g_i(\omega_j^r)$ and $a_i(\omega_j^r)$ for $0 \le i \le k$, $0 \le r < n$, and $1 \le j \le 3$. Using these values we can compute values of $C(\omega_j^r)$ in Step 13, $\Delta(\omega_j^r)$ in Step 14 and 20, and $e(\omega_j^r)$ in Steps 15 and 21. Using the values $e(\omega_j^r)$ we interpolate $e(x) \bmod q_i$ then use Chinese remaindering to recover the integer coefficients of e for Steps 9 and 20. The most expensive of these is computing $\Delta(\omega_j^r)$ in Step 15 which costs $\sum_{k=1}^{m} kn \in O(m^2 n) = O(m^2 d)$. Thus the modifications do not change the asymptotic complexity.

We mention a space optimization. At the k'th step, after computing $f_k(x)$ and $g_k(x)$, it would be natural to store the values of $f_k(\omega_j^m)$ and $g_k(\omega_j^m)$ in six two-dimensional arrays F_1, F_2, F_3 and G_1, G_2, G_3. But, to compute values of $\Delta(\omega_j^r)$ we need to compute $\sum_{i=1}^{k} f_i(\omega_j^r) g_{k+1-i}(\omega_j^r)$ for $0 \le r < n$ and $1 \le j \le 3$. To do this in a cache friendly manner, after computing $f_k(\omega_j^r)$ and $g_k(\omega_j^r)$, we store them transposed. For example, after step k the r'th row of F_j is

$f_0(\omega_j^r)$	$f_1(\omega_j^r)$	$\ldots$	$f_k(\omega_j^r)$	0	0	0	$\ldots$	0

The total storage for the six arrays is $6n(m+1)$ words of memory. It is more than six times the size of a dense input polynomial $a(x)$ so it limits the size of the problems we can solve. If $\|f\|\,\|g\| \le B$, which is usually the case, then more

Algorithm 4 Cubic Linear Hensel Lifting for $\mathbb{Z}[x]$: Monic Case.

Input: prime $p > 2$, $a \in \mathbb{Z}[x]$ with $d = \deg a > 1$ and $f_0, g_0 \in \mathbb{Z}_p[x]$ satisfying (i) a, f_0, g_0 are monic in x, (ii) $a - f_0 g_0 \equiv 0 \mod p$ and (iii) $\gcd(f_0, g_0) = 1$. Also a lifting bound $B > \max(\|f\|, \|g\|, \|a\|)$.

Output: $f, g \in \mathbb{Z}[x]$ such that $a = fg$ or FAIL $\Rightarrow$ no such f, g exist.

$f_0 \leftarrow \mathbf{mods}(f_0, p)$; $g_0 \leftarrow \mathbf{mods}(g_0, p)$;
Solve $sg_0 + tf_0 = 1$ using the extended Euclidean algorithm ..... $O(d^2)$
$a_0 \leftarrow \mathbf{mods}(a, p)$; $a \leftarrow (a - a_0)/p$ ..... $O(md)$
$e \leftarrow (a_0 - f_0 g_0)/p$ ..... $O(d^2)$
$k \leftarrow 1$; $nf \leftarrow 0$; $ng \leftarrow 0$.
while $p^k < 2B$ **do**
    $a_k \leftarrow \mathbf{mods}(a, p)$; $a \leftarrow (a - a_k)/p$ ..... $O(md)$.
    $e \leftarrow e + a_k$
    $c_k \leftarrow e \mod p$ ..... $O(d)$
    // *Solve* $f_k g_0 + g_k f_0 = c_k$ *in* $\mathbb{Z}_p[x]$ *for* $f_k, g_k \in \mathbb{Z}_p[x]$ *with* $\deg f_k < \deg f_0$
    $f_k \leftarrow (sc_k) \text{ rem } f_0$; $g_k \leftarrow (c_k - f_k g_0) \text{ quo } f_0$ ..... $O(d^2)$
    if $f_k \neq 0$ **then** $nf \leftarrow k$; $f_k = \mathbf{mods}(f_k, p)$ **end if**
    if $g_k \neq 0$ **then** $ng \leftarrow k$; $g_k = \mathbf{mods}(g_k, p)$ **end if**
    $C \leftarrow f_0 g_k + g_0 f_k$. ..... $O(d^2)$
    $\Delta \leftarrow \sum_{i=1}^{k} f_i g_{k+1-i}$; ..... $O(kd^2)$
    $e \leftarrow (e - C)/p - \Delta$ ..... $O(kd)$
    $k \leftarrow k + 1$.
end while
while $k < nf + ng$ **do**
    if $e \mod p \neq 0$ **then** output FAIL; **end if**
    $\Delta \leftarrow \sum_{i=k+1-ng}^{nf} f_i g_{k+1-i}$; ..... $O(kd^2)$
    $e \leftarrow e/p - \Delta$; ..... $O(kd)$
end while
if $e \neq 0$ **then** output FAIL **end if**
$f \leftarrow \sum_{i=0}^{nf} f_i p^i$; $g \leftarrow \sum_{i=0}^{ng} g_i p^i$. ..... $O(m^2 d)$
output f, g.

than half of the entries in these matrices will be zero. But we don't know which of f or g has the larger coefficients, so we don't know how big the rows of F_j and G_j should be in advance. To save half the space we create one array of size $n \times (m+1)$ and, for each j, store the values of f and g together as follows

$f_0(\omega_j^r)$	$f_1(\omega_j^r)$	...	$f_k(\omega_j^r)$	0	0	...	0	0	$g_k(\omega_j^r)$	...	$g_1(\omega_j^r)$	$g_0(\omega_j^r)$

If the length of the coefficients of f and g would collide, we simply make a copy of the array F_j and set G_j to point to the copy, and continue lifting.

6.1 Benchmarks for $\mathbb{Z}[x]$

Table 2 presents timings for Hensel lifting in $\mathbb{Z}[x]$ using $p = 2^{50} - 27$. The polynomials f and g have degree d with coefficients chosen uniformly at random from $(-p^m, p^m)$ and set $a = fg$. In Table 2, the timings in column C quartic are for our C implementation of LHL using an $O(m^2 d^2)$ algorithm for computing Δ. The time spent computing Δ is shown in parentheses. The timings in column Maple is for a Maple implementation of LHL. Maple beats our C code at $d = m = 800$ because it is using a fast multiplication for multiplication in $\mathbb{Z}_p[x]$.

	Old LHL		Fast QHL	Cubic LHL
d / m	C quartic(Δ)	Maple($error$)	Magma	C cubic(Δ)
25 / 25	.0012(.0005)	0.016(0.004)	0.015	0.0025(.0005)
50 / 50	.0101(.0072)	0.134(0.043)	0.062	0.0106(.0028)
100 / 100	0.124(0.108)	1.157(0.607)	0.294	0.051(0.015)
200 / 200	1.774(1.685)	6.745(4.656)	1.683	0.263(0.080)
300 / 300	8.598(8.342)	22.03(16.84)	7.53	0.850(0.400)
400 / 400	26.66(26.13)	53.16(42.67)	10.70	1.500(0.650)
600 / 600	138.2(136.1)	143.7(123.3)	50.44	6.310(3.900)
800 / 800	429.4(424.7)	370.2(320.6)	65.98(0.31gb)	10.93(6.350)
1000/ 1000	1052.(1042.)	674.5(582.4)	74.92(0.37gb)	17.16(9.070)(0.10gb)
2000/ 2000	NA	5875.(5256.)	425.1(1.35gb)	119.2(69.46)(0.40gb)
4000/ 4000	NA	NA	2345.(5.28gb)	880.6(531.3)(1.61gb)
8000/ 8000	NA	NA	12287.2(21.01gb)	6748.(4170.)(6.42gb)
10000/10000	NA	NA	46646.8(47.30gb)	18315.(13333.)(16.1gb)
12000/12000	NA	NA	NA	27775.(19195.)(19.3gb)
10 / 100000	1988.(1997.)	NA	233.8	1244.8(1238.8)
100 / 10000	1218.(1216.)	NA	234.7	105.1(101.1)
1000 / 1000	1052.(1042.)	NA	74.9	17.17(9.68)
10000 / 100	1091.(1004.)	NA	106.9	57.00(3.48)
100000 / 10	1972.(858.8)	NA	155.7	777.3(1.46)

Table 2. Timings in CPU seconds for Hensel lifting in $\mathbb{Z}[x]$ using $p = 2^{50} - 27$. LHL = linear Hensel lifting, QHL = quadratic Hensel lifting, NA = Not Attempted.

The timings in column Magma are for our Magma implementation of QHL in Appendix 1. The multiplication in $\mathbb{Z}[x]$ and division in $\mathbb{Z}_M[x]$ where $M = p^{2^k}$ are both done using fast arithmetic. The Magma timings in the last 5 rows verify this. The timings in column C cubic are for our C implementation of our new $O(d^2m + m^2d)$ algorithm. The time spent computing Δ (up to $m = d = 800$) is shown in parentheses. Shown in parentheses for $m = d \geq 1000$ (and for Magma) is the total space used in gigabytes.

The data in Table 2 shows Magma's fast QHL lift beats the $O(d^2m^2)$ LHL algorithm at $m = d = 200$. Our first result is that our $O(d^2m + m^2d)$ LHL algorithm beats the $O(d^2m^2)$ algorithm just over $m = d = 50$. Our second result is that our cubic LHL beats Magma's fast QHL for all values $d = m$ in Table 2. It also uses about one third of the space of Magma. This is a very good result. The only cases where it loses to Magma are $d = 10, m = 10^5$ and $d = 10^5, m = 10$.

7 Conclusion

In this paper we have introduced a cubic algorithm for linear Hensel lifting (LHL). We have shown that our C implementation beats a Magma implementation of fast quadratic Hensel lifting (QHL) in $\mathbb{Z}_p[x, y]$ and $\mathbb{Z}[x]$ for a wide range of input sizes. Thus we believe our cubic algorithm stands a chance of becoming the algorithm of choice in practice.

In [10] Monagan and Tuncer reduce multivariate Hensel lifting to many bivariate Hensel lifts in $\mathbb{Z}_p[x, y]$. In that context, our cubic algorithm beats the quartic LHL early enough to be useful. In Table 1, for $d = 100$, we obtain a speedup of a factor of 166/34.2=4.8.

In [7] Miolo and Yun used Hensel lifting to compute $g = \gcd(a, b)$ in $\mathbb{Z}[x]$ and they compared it with the modular GCD algorithm (see Algorithm 6.38 of [3]).

The quartic LHL algorithm that Miola and Yun developed is not competitive with the modular GCD algorithm which has a cubic complexity of $O(md^2+m^2d)$. Because of this disadvantage, in 1989, the author replaced Maple's GCD code for $\mathbb{Z}[x]$, which was using Miola and Yun's LHL with the modular GCD algorithm. Magma also uses the modular GCD algorithm for gcd computation in $\mathbb{Z}[x]$. Our new cubic LHL means the modular GCD algorithm no longer has this asymptotic advantage over LHL. Indeed Hensel lifting has an advantage: if one input, a say, is much smaller than b (the degree is smaller or the size of the coefficients is smaller), Hensel lifting can be applied to the smaller polynomial a whereas the modular GCD algorithm must always compute with both a and b.

In the paper we only considered the two factor case. We are currently working with G. Paluck to develop cubic LHL for $a \in \mathbb{Z}_p[x]$ for more than two factors.

References

1. Bernardin, L.: On Bivariate Hensel Lifting and its Parallelization. *Proc. of ISSAC 1998*, pp. 96–100, ACM (1998).
2. A. Bostan, G. Lecerf, B. Salvy, E. Schost, B. Weibelt. Complexity Issues in Bivariate Polynomial Factorization. *Proc. of ISSAC 2004*, 42–49, ACM (2004).
3. von zur Gathen J., Gerhard J.: *Modern Computer Algebra*. 3rd ed. Cambridge University Press (2013).
4. Geddes K.O., Czapor S.R., Labahn G.: *Algorithms for Computer Algebra*. Kluwer Academic (1992).
5. Niels Möller and Torbjörn Granlund. Improved division by invariant integers IEEE Transactions on Computers, **60**, 165–175, 2011.
6. Erich Kaltofen. Sparse Hensel lifting. Proc. of EUROCAL '85, LNCS **204**: 4–17. Springer (1985)
7. Alfonso Miola and David Y.Y. Yun. Computational aspects of Hensel-type univarate polynomial greatest common divisor algorithms. SIGSAM Bulletin **8**(3): 46–54. ACM (1974).
8. Moses J., Yun D.Y.Y. The EZ-GCD algorithm. *Proc. ACM annual conference*, 159-166, ACM, (1973).
9. Monagan M., Tuncer B.: Using Sparse Interpolation in Hensel Lifting. In *Proc. of CASC 2016*, LNCS **9890**: 381–400, 2016.
10. Monagan M., Tuncer B.: Sparse multivariate Hensel lifting: A high-performance design and implementation. In *Proc. of ICMS 2018*, LNCS **10931**, 359–368, 2018.
11. Torbjörn Granlund and Peter Montgomery. Division by Invariant Integers using Multiplication In *Proceedings of PLDI '94*, 61–72, ACM (1994).
12. David Musser. Algorithms for Polynomial Factorization. Computer Science Technical Report No. 134. (Ph.D. Thesis), Univ. of Wisconsin, 1971.
13. David R. Musser. Multivariate Polynomial Factorization. *J. ACM.*, **22**: 291–308, (1975).
14. Allan Steel. Private communication.
15. Wang, P.S., Rothschild, L.P. Factoring Multivariate Polynomials over the Integers. SIGSAM Bulletin No. 28, 1973.
16. Paul S. Wang and Linda Preiss Rothschild. Factoring Multivariate Polynomials over the Integers. *Math. Comp.* **29**(131): 935–950 (1975).
17. Paul S. Wang. An improved multivariate polynomial factoring algorithm. *Math. Comp.* **32**(144): 1215–1231 (1978).
18. Paul S. Wang. The EEZ-GCD Algorithm. SIGSAM Bulletin **14**(2): 50–60 (1980).
19. Yun D.Y.Y. The Hensel Lemma in algebraic manipulation. Ph.D. Thesis, MIT, (1974).
20. Zassenhaus H. On Hensel factorization, I. *J. Number Theory* **1**: 291–311 (1969).

Appendix 1: Magma code for QHL in $\mathbb{Z}[x]$.

Reference: Algorithm 15.10 "Hensel step" from [3]. 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 $\mathbb{Z}[x]$, fast division for the two divisions QuotRem(...) in $\mathbb{Z}_m[x]$ and fast division for the integer divisions by m in reduce(...).

```
Z := IntegerRing();
Zx<x> := PolynomialRing(Z);

height := function(f)
    m := 0;
    for c in Coefficients(f) do m := Max(Abs(c),m); end for;
    return m;
end function;

FactorBound := function(f)
    h := height(f);
    d := Degree(f);
    return 2^(d-1)*h*Ceiling(Sqrt(d));
end function;

getpoly := function(d,m,p)
    n := p^m;
    C := [ Random(-n,n) : i in [0..d-1] ];
    g := Zx!C;
    return x^d+g;
end function;

DivideModm := function(f,g,m)
   Zm := ResidueClassRing(m);
   Zmx<x> := PolynomialRing(Zm);
   F := Zmx!f;
   G := Zmx!g;
   Q,R := Quotrem(F,G); // divide in Zm[x]
   q := Zx!Q;
   r := Zx!R;
   return q,r;
end function;

reduce := function( f, m )
    return Zx![ c mod m : c in Coefficients(f) ];
end function;
mods := function(a,p,p2)
   if a ge p2 then return a-p; else return a; end if;
end function;
mapmods := function(f,p,p2)
   return Zx![ mods(c,p,p2) : c in Coefficients(f) ];
end function;


HenselLift := function( f, g0, h0, p )

  Fp := GaloisField(p);
```

```
  Fpy<y> := PolynomialRing(Fp);

  g0 := mapmods(g0,p,p/2);  g := g0;
  h0 := mapmods(h0,p,p/2);  h := h0;
  G0,S0,T0 := XGCD( Fpy!g0, Fpy!h0 );
  s := Zx!S0; t := Zx!T0;

  B := FactorBound(f); // height bound on factors of f
  m := p;
  e := f-g*h;

  while m lt 2*B do

    m := m^2; m2 := (m-1)/2;

    e := reduce(e,m);
    q,r := DivideModm(s*e,h,m);
    u := t*e+q*g; u := reduce(u,m);
    g := g + u; g := mapmods( g, m, m2 );
    h := h + r; h := mapmods( h, m, m2 );
    e := f-g*h;

    if e eq 0 then break; end if;
    if m gt 2*B then break; end if;

    // Lift diophantine equation solutions s and t
    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;

  end while;
  return  e, g, h  ;
end function;

d := 400; m := 400; p := 2^31-1;
g := getpoly(d,m,p);
h := getpoly(d,m,p);
f := g*h;
g0 := reduce(g,p);
h0 := reduce(h,p);
printf "Start Hensel lift:  d=%m  m=%m  p=%m\n", d,m,p;
time e, G, H := HenselLift(f,g0,h0,p);
e; G-g; H-h; // should all be 0
```

The reader can see that the algorithm does 4 multiplications at precision 2^k (mod $m = p^{2^k}$), namely, `s*e`, `t*e`, `q*g` and `f*g` and one division of $se \div h$ in $\mathbb{Z}_m[x]$. Fast division (see Algorithm 9.5 of [3]) reduces division to 5 multiplications of precision 2^k, three to compute the inverse, one to obtain the quotient and one more for the remainder. Thus 9 multiplications at precision 2^k.

The algorithm also must lift the solutions to the diophantine equation to precision $2^{(k-1)}$. The reader can see that there are 5 multiplications `s*g`, `t*h`, `s*b`, `t*b` and `c*g` and one division $sb \div h$ which is equivalent to another 10 multiplications at precision 2^{k-1}.

Adding up the work to lift the factors from mod p^1 to mod p^{2^k} we have $9\sum_{i=1}^{k} 2^i = 18 \times 2^k - 18$ plus $10\sum_{i=1}^{k-1} 2^i = 10 \times 2^k - 20$ for a total of 28 multiplications at precision 2^k.

Appendix 2: Magma code for QHL in $\mathbb{Z}_p[x, y]$.

```
dx :=  80;
dy :=  80;
p := 2^31-1;
alpha := 3;
n := 1;

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

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

getcoeff := function(dy,p)
    E := getexpons(dy,3); // E has 3 distinct integers
    C := 0;
    for e in E do C := C + Random(p)*y^e; end for;
    return C;
end function;

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

G := getpoly(dx,dy,p);
H := getpoly(dx,dy,p);
time f := G*H;

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

g0 := reduce(G,y-alpha); G0 := Zpy!g0;
h0 := reduce(H,y-alpha); H0 := Zpy!h0;
gcd,S0,T0 := XGCD( G0, H0 );

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

```
DivideModm := function(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;

time for i := 1 to n do

  s := convertytox(S0);
  t := convertytox(T0);

  g := g0;
  h := h0;
  m := Zpy!(y-alpha);
  e := f-g*h;

  k := 1;
  while Degree(m) le 2*dy 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;
end for;

e; G-g; H-h; // should be 0
```