Complexity results for factoring univariate polynomials over the rationals (version 0.3)

Mark van Hoeij & Andrew Novocin
Department of Mathematics
Florida State University
Tallahassee, FL 32306-4510, USA
{hoeij,anovocin}@math.fsu.edu

July 18, 2007

1 Introduction

In [6] Zassenhaus gave an algorithm for factoring polynomials $f \in \mathbb{Q}[x]$. In this algorithm one has to solve a combinatorial problem of size r, where r is the number of local factors of f at some suitably chosen prime p. This combinatorial problem consists of selecting the right subsets of the set of local factors. In the worst case, the algorithm [6] ends up trying 2^{r-1} such subsets (if a subset has been tried then one can skip its complement) so this algorithm has an exponential worst case complexity.

Let N be the degree of the polynomial f. Most of the time the number r is much smaller than N, which explains why Zassenhaus' algorithm is often fast despite its exponential worst case complexity. To observe this exponential complexity on a computer, take polynomials of high degree for which the Galois group contains only elements of low order (worst case are the Swinnerton-Dyer polynomials, whose Galois groups have only elements of order 1 and 2, consequently, these polynomials have r as high as $N/2$).

In [4] Lenstra, Lenstra and Lovász introduced a lattice reduction algorithm, which we shall refer to as the LLL algorithm. The paper [4] also gave a factoring algorithm which avoids the above mentioned combinatorial problem by constructing factors of f using LLL. The result was the first polynomial time algorithm for factoring polynomials in $\mathbb{Q}[x]$. Schönhage [5] gave a sharper complexity result for a similar approach.

The paper [3] gave a factoring algorithm that uses LLL as well. What was new is that LLL was not used to construct the factors (constructing factors is a problem whose size depends both on the degree as well as on the size of the coefficients). Instead, LLL was only used to solve the combinatorial problem (a problem whose size depends only on r, because subsets of a set with

r elements can be represented by elements of $\{0,1\}^r$). Since the problem to be solved by LLL has smaller size for the algorithm in [3], it is reasonable to expect that it will run faster than prior factoring algorithms. This expectation is confirmed experimentally, however, actually proving that the algorithm has a better complexity has stubbornly resisted efforts so far.

In [3], the data that is fed to the LLL algorithm (in order to solve the combinatorial problem) consists of p-adic digits of the so-called traces (power sums). This raises the question how many p-adic digits, and of how many traces, need to be fed to LLL in order to successfully solve the combinatorial problem. If one uses too few traces or p-adic digits then one may make only partial progress instead of fully solving the combinatorial problem. If one uses too many, one can end up solving the combinatorial problem using much more CPU time than would have been necessary. From a practical point of view, the latter (using too many traces/digits) is worse because the wasted time can not be recovered, while the former (using too few traces/digits) can be remedied by gradually adding more traces and/or digits.

In fact, even if one knew the exact number of p-adic digits and traces that need to be used in order to solve the combinatorial problem with one call to LLL, then as explained in [3, section 2.3 item 3] it could still be faster to use fewer traces/digits for the first call to LLL despite the resulting increase in the number of calls to LLL. Thus, [3, section 2.3 item 2] proposed to add only few traces at a time, while [3, section 2.3 item 3] suggested to use $O(r^2)$ bits of data in the first call to LLL (a p-adic digit counts as $\log_2(p)$ bits).

The trade-off is that adding many traces/digits at a time will reduce the number of calls to LLL while increasing the cost of each call. Adding few traces/digits at a time reduces the cost of each LLL call, but the number of calls goes up. There are two extreme positions to make concerning this trade-off.

A. Minimize the number of calls to LLL. (In the version described in Theorem 4.3 in [2] this number is brought down to 1.)

B. Minimize the complexity of each individual call. (In the version described by Belabas [1], the cost of each LLL call is bounded by a polynomial that depends only on r, that is, a polynomial that is independent of both N and the coefficient size of f.)

The strategy proposed by Belabas in [1] takes side [B] to the extreme. It uses just one trace at a time, and adds only $O(r)$ bits of data at a time. While this leads to a good complexity bound for each LLL call, one that depends solely on r, it becomes difficult to bound the number of LLL calls if one takes side [B]. We expect the number of LLL calls in Belabas' implementation to be $O(r)$ for typical examples, however, it should be possible to construct examples where this number is significantly higher.

What was new about Belabas' strategy is the order in which the p-adic digits are used; this is done in such a way that each call to LLL will maximally benefit from the preceding LLL calls. This way the fact that the number of LLL calls may be large does not hurt the practical performance of the algorithm. Indeed, the number of LLL calls appears to have little effect on the running times of Belabas' implementation; section 2.5.1 in [1] mentions that reducing the value of the parameter BitsPerFactor by a factor 2 (which should double the number of LLL calls) has only a minor impact on the computation timings.

However, having an unknown (and potentially large) number of LLL calls certainly complicates the problem of bounding the complexity. Thus, to get a complexity bound, the paper [2] took side [A]. In Theorem 4.3 in [2] it was shown that the combinatorial problem would be solved with a single LLL call if one applies LLL to a certain lattice (called the all-coefficients lattice in [2]). However, this lattice is as big as the one used in [4] and thus one ends up with the same complexity. The paper [2] also showed (Theorem 4.6 in [2]) how to bound the complexity for [B] (i.e. for Belabas' version, called "one coefficient at a time" in [2]). Although this bound was not spelled out explicitly (Theorem 4.6 in [2] only says "polynomially bounded"), if one follows the steps of the proof one sees that the complexity result for [B] in [2] is much worse than the bound for [A], which is ironic, because [B] runs very much faster than [A].

Our goal in this paper is to address this unsatisfactory situation where version [B] has a worse complexity bound than [A] despite being much faster. Our starting point is the paper [2], the reader is assumed to be familiar with sections 1 through 4 in [2]. Our goal will be to give a complexity result for [B] that matches its actual performance. Our complexity result explains some key features of [B] that were observed in Belabas' implementation. For instance, it explains why doubling the number of LLL calls (by halving the parameter BitsPerFactor) has so little effect on the CPU time. Vice versa, these features hint to our complexity result.

1.1 Overview of Lattice Reduction

The purpose of this subsection is to list notations and known facts (from [4]) that will be needed throughout the paper. A lattice L is a discrete subset of $\mathbb{R}^m$ that is also a $\mathbb{Z}$-module. Let $b_1, \ldots, b_r \in L$ be a basis of L and denote $b_1^*, \ldots, b_r^* \in \mathbb{R}^m$ as the Gram-Schmidt orthogonalization over $\mathbb{R}$ of $b_1, \ldots, b_r$. Let $l_i = \log_{4/3}(\| b_i^* \|^2)$, and denote $\mu_{i,j} = \frac{b_i \cdot b_j^*}{b_j^* \cdot b_j^*}$. Note that $b_i, b_i^*, l_i, \mu_{i,j}$ will change throughout the algorithm sketched below.

Definition 1. *$b_1, \ldots, b_r$ is* LLL-reduced *if $\| b_i^* \|^2 \leq 2 \| b_{i+1}^* \|^2$ for $1 \leq i < r$. (The definition in [4] is slightly stronger, for convenience we only listed what is needed for this paper. See also Remark 2 at the end of this section.)*

efficiency improvements.

Algorithm 1 (Rough sketch of Lattice Reduction Algorithms).
Input: *A basis $b_1, \ldots, b_r$ of a lattice L.*
Output: *An LLL-reduced basis of L.*

1. (Gram-Schmidt over $\mathbb{Z}$). *By subtracting suitable $\mathbb{Z}$-linear combinations of $b_1, \ldots, b_{j-1}$ from b_j make sure that $|\mu_{i,j}| \leq 1/2$ for all $j < i$.*

2. (LLL Switch). *If there is a k such that interchanging b_{k-1} and b_k will decrease l_{k-1} by at least 1 then do so.*

3. (Repeat). *If there was no such k in Step 2, then the algorithm stops. Otherwise go back to Step 1.*

That the above algorithm terminates, and that the output is LLL-reduced was shown in [4]. Step 1 has no effect on the l_i. In step 2 the only l_i that change are l_{k-1} and l_k. independent of the choice of basis. To illustrate step 2 in more detail, suppose that $c_1, \ldots, c_r$ is a basis of L obtained from $b_1, \ldots, b_r$ by applying step 2. So $c_k = b_{k-1}$ and $c_{k-1} = b_k$, and $c_j = b_j$ for the remaining j's.

Since $b_1, \ldots, b_r$ and $c_1, \ldots, c_r$ are bases of the same L, they have the same determinant (the product of $\| b_i^* \|$ for $i = 1, \ldots, r$) and hence

$$\| c_{k-1}^* \|^2 \| c_k^* \|^2 = \| b_{k-1}^* \|^2 \| b_k^* \|^2. \tag{1}$$

Step 2 is only taken if it decreases l_{k-1} by at least 1, so $\| c_{k-1}^* \|^2 \leq \frac{3}{4} \| b_{k-1} \|^2$. The vector b_k^* is obtained from b_k by reducing it modulo $\mathbb{R}b_1 + \ldots + \mathbb{R}b_{k-1}$ while c_{k-1}^* is obtained from b_k by reducing it modulo $\mathbb{R}b_1 + \ldots + \mathbb{R}b_{k-2}$. Hence b_k^* can not be longer than c_{k-1}^*. Combining this we find

$$\| b_k^* \|^2 \leq \| c_{k-1}^* \|^2 \leq \frac{3}{4} \| b_{k-1}^* \|^2 \tag{2}$$

which by equation (1) is equivalent to

$$\| b_{k-1}^* \|^2 \geq \| c_k^* \|^2 \geq \frac{4}{3} \| b_k^* \|^2.$$

The equations imply:

Observation 1. *An LLL switch can not increase max$(l_1, \ldots, l_r)$, nor can it decrease min$(l_1, \ldots, l_r)$.*

In summary, an LLL switch reduces l_{k-1} by at least 1, and increases l_k by the same amount (because of equation (1)). This way each LLL switch moves G-S length towards the later vectors, while the sum of the logarithmic G-S lengths $l_1 + \cdots + l_r$ stays the same. Moreover, during the Lattice Reduction Algorithm, the highest G-S length can not increase, and the lowest G-S length can not decrease.

Consider the sum $\sum_i (r-i) l_i$. Each LLL switch reduces this sum by at least 1. Hence, if l_{old} was the value of this sum at the beginning of the computation, and l_{new} was the value at the end, then there can not have been more than $l_{\text{old}} - l_{\text{new}}$ LLL switches during the computation. The same idea was used in the proof of the complexity result given in [4, Proposition (1.26)]. Another useful fact is the following:

Fact 1. *If $\| b_r^* \|^2 > B$ then any vector in L with squared length $\leq B$ is a $\mathbb{Z}$-linear combination of $b_1, \ldots, b_{r-1}$.*

In other words, $b_1, \ldots, b_r$ is a basis of some lattice L, and if the last vector has sufficiently large G-S length, then, in applications (including ours) where one is only interested in elements of L of squared length $\leq B$, one can remove the last basis element.

Fact 1 follows from the proof of (1.11) in [4], and is true regardless of whether $b_1, \ldots, b_r$ is LLL-reduced or not. However, if one chooses an arbitrary basis $b_1, \ldots, b_r$ of some lattice L, then it is unlikely that the last vector has large G-S length (after all, $\parallel b_r^* \parallel$ is the length of b_r reduced over $\mathbb{R}$ modulo all of $b_1, \ldots, b_{r-1}$). The effect of LLL reduction (Algorithm 1) is to move G-S length towards later vectors. So LLL reduction is very useful because if enough of this G-S length arrives at the last vector, then it can be discarded, which brings us one step closer to our target.

Remarks:

1. There are a number of lattice reduction algorithms that are variations of the LLL algorithm sketched above. We would like to present our complexity result in a way that is independent of which variation is used.

   Each of these variations uses (at least asymptotically) the same number of LLL switches. The differences in complexity come from differences in the cost per LLL switch. So we will express our complexity result in terms of the number of LLL switches. This way our result will be compatible with each variation on the LLL algorithm.

2. Schönhage [5] gives a slightly different definition of reduced, called semi-reduced. This allows him to apply a divide-and-conquer strategy called block-wise reduction that reduces the asymptotic cost per LLL switch. With minor modifications, the results in this paper carry through if one replaces 'reduced' by Schönhage's 'semi-reduced'.

2 Partial Reductions

This section presents a type of partial lattice reduction that we will apply to factoring polynomials.

Definition 2. *Given a lattice L in $\mathbb{R}^m$ and a positive real number B, we call $S = b_1, \ldots, b_k$ a B-reduced sequence for L if S is LLL-reduced, $\parallel b_k^* \parallel^2 \leq B$, and for every $v \in L$ with $\parallel v \parallel^2 \leq B$ we have $v \in SPAN_{\mathbb{Z}}(S)$.*

Algorithm 2 (B-Reduce).
Input: *$V = b_1, \ldots, b_r$ a basis of a lattice L.*
Output: *A B-reduced sequence for L.*
Algorithm: *The same as algorithm 1, except that whenever the last vector has squared G-S length $> B$ at any point during the computation, it is removed.*

Correctness of the algorithm is based on Fact 1 in subsection 1.1. Now consider the following problem. Let $\pi : \mathbb{R}^{m+1} \rightarrow \mathbb{R}^m$ be projection onto the first m coordinates, and $S = b_1, \ldots, b_r \in \mathbb{R}^{m+1}$ with $\pi(b_1), \ldots, \pi(b_r)$ already B-reduced. We would like to B-reduce S in a way that takes advantage of the fact that the first m entries are already B-reduced.

We could just apply algorithm 2 to S, but then the worst-case complexity would be the same as algorithm 1. Specifically, if we applied algorithm 2 to S, then the *switch-complexity* may depend on the size of the last entries of the vectors of S. Here the switch-complexity is defined as the number of LLL-switches (step 2 of algorithm 1) that occur during the computation.

We will give a variant of algorithm 2 with a better switch-complexity, one that only depends on B and r and not the size of the last entries (This does not imply that the complexity itself is independent of the size of last entries because we still work with numbers of that size. But a lower switch-complexity does imply a lower overall complexity because both algorithms have to work with numbers of that size.) The algorithm uses an idea obtained from strategy B in [1].

Algorithm 3 (Gradual B-Reduce).
Input: $S = b_1, \ldots, b_r \in \mathbb{R}^{m+1}$ *with* $\pi(b_1), \ldots, \pi(b_r)$ *already* B*-reduced.*
Output: *A* B*-reduced sequence.*
Algorithm:

```
1. Let $d$ be the smallest nonnegative integer for which $|\frac{b_{1,m+1}}{2^{rd}}| \leq 2^r$, where $b_{i,m+1}$ is the last entry of $b_i$.
2. Scale down the last entry of each vector $b_{i,m+1} := \frac{b_{i,m+1}}{2^{rd}}$ (for $i = 1, \ldots, r$) and set $s := r$.
3. Repeat the following $d$ times:
   (a) If $1 < max(|b_{1,m+1}|, \ldots, |b_{s,m+1}|)$ (where $s$ is the number of remaining vectors) then run algorithm 2 and let $s$ be the number of remaining vectors.
   (b) (Gradually scale back up). Let $b_{i,m+1} := 2^r\, b_{i,m+1}$ (for $i = 1, \ldots, s$).
4. Run algorithm 2 and stop.
```

In order to see why this algorithm also returns a B-reduced basis we need the following:

Lemma 1. *Let* $\sigma : \mathbb{R}^{m+1} \to \mathbb{R}^{m+1}$ *scale up the last entry by some factor* $\delta > 1$*. Let* $S = b_1, \ldots, b_r$ *and* $\sigma(S)$ *the image of* S*. Then* $\| b_i^* \| \leq \| \sigma(b_i)^* \|$.

The Lemma implies that removing a vector (during the call to algorithm 2 in step 3a) before the last entry was scaled back up to original scaling (by repeated calls to step 3b) will not cause removal of a vector whose original squared G-S length (as it was before step 2) had been $\leq B$.

Proof. Let $V_i = \{b_i - (a_{i-1}b_{i-1} + \cdots + a_1 b_1) \,|\, a_1, \ldots, a_{i-1} \in \mathbb{R}\}$, then b_i^* is just the shortest vector in V_i. Now the claim is that the shortest vector in V_i is not longer than the shortest vector in $\sigma(V_i)$. So let w be the shortest vector in $\sigma(V_i)$. There is some $v \in V_i$ with $\sigma(v) = w$. Let $w = (c_1, \ldots, c_m)$, then $v = (c_1, \ldots, c_m/\delta)$. Now $\| b_i^* \| \leq \| v \| \leq \| w \| = \| \sigma(b_i)^* \|$. $\square$

Lemma 2. *In algorithm 3 every vector has squared G-S length $\leq 2^{3r}B$ at any time during steps 3 and 4. In particular, this holds for every removed vector at the time of its removal.*

Proof. Since running B-reduce cannot increase G-S lengths (by Observation 1) we need to decide how large a vector can possibly be just after scaling up by 2^r. But just before the scaling we know that $b_1, \ldots, b_s$ is B-reduced which implies that $\| b_1^* \|^2 \leq 2\| b_2^* \|^2 \leq \cdots \leq 2^{s-1}\| b_s^* \|^2 \leq 2^{s-1}B$, since the last vector is no larger than B and the definition of a reduced basis. So just after scaling the largest G-S length can have squared length no greater than $(2^{2r})(2^{s-1}B) \leq 2^{3r}B$. $\square$

Definition 3 (Value). *Suppose $b_1, \ldots, b_s$ is the current set of vectors at some point in the algorithm. We will define*

$$\mu(b_1, \ldots, b_s) = 0 \cdot l_1 + 1 \cdot l_2 + \cdots + (s-1) \cdot l_s + r(r-s)\log_{4/3}(2^{3r}B)$$

which we call the value *at that point in the algorithm. It is a weighted sum (with weights $0, 1, \ldots, s-1$ and r) of the logarithmic G-S lengths, where the $r - s$ removed vectors are counted as if they had squared G-S lengths $2^{3r}B$.*

Theorem 1. *Gradual B-reduce has switch-complexity $O(r^3 + r^2\log(B))$ if we assume that the Gram-Schmidt lengths of $\pi(b_2), \ldots, \pi(b_r)$ are at least 1.*

Proof. The assumption on the Gram-Schmidt lengths implies that the value of μ at the end of step 2 is nonnegative. Substituting $s = 0$ gives the highest possible value for μ after step 2 (see lemma 2). Hence $\mu \leq r^2\log_{4/3}(2^{3r}B) = O(r^2(r + \log(B))$ will hold at any time during steps 3 and 4. Because each LLL-switch increases μ by at least 1, we can prove the theorem by showing that the value μ never decreases during steps 3 and 4.

Removing a vector can only increase μ by Lemma 2. The only steps that change G-S lengths are scalings and LLL-switches. Lemma 1 shows that scaling up (step 3b) can only increase G-S lengths and hence can not decrease μ. $\square$

3 Partial Reduction of a Special Matrix

We want to bound the switch-complexity of B-reducing the rows of the following type of matrix:

$$\begin{pmatrix} & & & & & d_N \\ & & & & \iddots & \\ & & & d_1 & & \\ 1 & & & * & \cdots & * \\ & \ddots & & \vdots & \ddots & \vdots \\ & & 1 & * & \cdots & * \end{pmatrix}$$ where the lower left hand corner is an $r \times r$

identity matrix, $*$ represents any number, and ${d_i}^2 > 2^{((r+1)^2-(r+1))/2}B^{(r+1)}$. In addition we require $|d_i| > 1$ and that d_i has the largest absolute value in its column.

Algorithm 4. *Matrix Reduce*
Begin with the vectors $S = b_1, \ldots, b_r$ which are the rows of the $r \times r$ identity matrix in the bottom left corner, throughout the algorithm we will let s denote the current number of vectors in S and c denote the number of removed vectors. Then perform the following:

1. *Add a row and a column. By this we mean expand the corner of the matrix we will deal with, which includes adjoining a new entry for each current row-vector and then adding the next row-vector (properly truncated). Note that the new vector is added as the* first *entry in S (this will simplify the proof of lemma 5 below).*

2. *Call Algorithm 3 on S.*

3. *Unless the matrix has been exhausted go back to step 1.*

If $b_1, \ldots, b_s$ is the current collection of vectors and c is the number of vectors which have been removed, then we define the Value of the current vectors as:

$$\mu(b_1, \ldots, b_s) = 0 \cdot l_1 + 1 \cdot l_2 + \cdots + (s-1) \cdot l_s + (r+1)c \log_{4/3}(2^{3(r+1)}B)$$

Lemma 3. *In step 2 at least one vector gets removed so that s is never more than $r+1$ in Algorithm 4.*

Proof. If no vector gets removed then the determinant squared must be the same both before and after step 2. But before step 2 the determinant squared is $\geq B^{(r+1)}2^{((r+1)^2-(r+1))/2}$ (because of the d_i). After step 2 we have $\| b_s^* \|^2 \leq B$ and $\| b_i^* \|^2 \leq 2^{s-i} \| b_s^* \|^2$ which imply that our determinant squared must be $\leq B^s 2^{(s^2-s)/2}$. Since the first time step 2 is called we have $s = r+1$ we know that at least one vector must have been removed. But now by repeating this logic we have $s \leq r+1$ for every other time step 2 is called. Therefore every time step 2 is called at least one vector is removed. $\|^2$. Also the product of the G-S lengths is the determinant of S. So if we were to keep all vectors when adding a column we would need that $\| b_s^* \|^2 \leq B$. These facts imply that the determinant squared of the starting lattice must be below $B^s 2^{(s^2-s)/2}$ but ${d_i}^2 > B^{(r+1)}2^{((r+1)^2-(r+1))/2}$ for all i, so that the determinant of any of the lower left hand corners we take is large enough to guarantee at least one vector is removed. $\square$

Corollary 1. *No removed vector (at the time of its removal) can have l_i larger than $\log_{4/3}(2^{3(r+1)}B)$, and when lattice reduction (Algorithm 1) is called no remaining vector has l_i larger than $\log_{4/3}(2^{3(r+1)}B)$.*

The proof is the same as the proof for Lemma 2.

Lemma 4. *The G-S lengths of any vector in Algorithm 4 are always* ≥ 1*. In other words* $l_i \geq 0$*.*

Proof. The initial vectors have G-S length ≥ 1 ($d_i \geq 1$ above). So the lemma follows from Observation 1. $\square$

Lemma 5. *Value* μ *never decreases in Algorithm 4.*

Proof. We have already shown that Algorithm 3 will not decrease value. It remains to show that step 1 will not decrease value.

Step 1 adds the vector $(0, \ldots, 0, d_i)$ to the beginning of $b_1, \ldots b_s$. This way the G-S lengths of $b_1, \ldots, b_s$ are not changed by the addition of an extra entry. They will only have their weights increased by one in μ, which can only increase μ (because of lemma 4). The length of $(0, \ldots, 0, d_i)$ has no impact on μ since it is counted with weight 0. $\square$

Lemma 6. *Each LLL switch increases Value* μ *by at least 1.*

This is the same as in the previous section.

Lemma 7. μ *is always* $\leq N(r+1)(\log_{4/3}(2^{3(r+1)}B)$*, so that* $\mu = \mathcal{O}(Nr(r + \log(B)))$*.*

Proof. As seen in the previous section when Gradual B-Reducing no l_i can become larger than $\log_{4/3}(2^{3(r+1)}B)$. Thus a remaining vector contributes less to value than a removed vector. In this application however we do allow a large G-S length but only when the new vector is added in step 1. However by adding this vector as the first vector in our set it contributes nothing to value and when algorithm 3 is called its size is immediately scaled back down. Thus the value is always less than it would be if all vectors were removed. $\square$

This implies that the number of LLL switches is bounded by $N(r+1)(\log_{4/3}(2^{3(r+1)}B) = \mathcal{O}(Nr(r + \log(B)))$, since each switch ensures an increase of at least 1.

4 New Bounds for Factoring in $\mathbb{Q}[x]$

Now we simply observe that factoring over $\mathbb{Q}$ can be accomplished by B-reducing a matrix with the same format as our special matrix in the previous section. over $\mathbb{Q}$.

Notation: Let $f \in \mathbb{Q}[x]$ be a polynomial of degree N, and p a prime such that $f \equiv f_1 \cdots f_r \bmod p$ is the factorization of f in $\mathbb{F}_p[x]$. Let $f \equiv \tilde{f}_1 \cdots \tilde{f}_r$ be the factorization of $f \bmod p^a$ for some positive integer a.

In [2] we see that solving the factorization of f over $\mathbb{Q}$ can be accomplished by finding a B-reduction of the rows of the following matrix, for a B value of $r+1$ (by scaling the final entries so that our target vectors have final entries below $1/\sqrt{N}$ rather than 1) and a sufficiently large value for a:

$$\begin{pmatrix} & & & & & p^a \\ & & & & \iddots & \\ & & & p^a & & \\ 1 & & & * & \cdots & * \\ & \ddots & & \vdots & \ddots & \vdots \\ & & 1 & * & \cdots & * \end{pmatrix}$$ where the $*$ represent coefficients of the logarithmic derivative (multiplied by f) for each of the local factors.

We will make some minor changes to produce a matrix that looks like:

$$\begin{pmatrix} & & & & & p^{a-b_N} \\ & & & & \iddots & \\ & & & p^{a-b_1} & & \\ 1 & & & * & \cdots & * \\ & \ddots & & \vdots & \ddots & \vdots \\ & & 1 & * & \cdots & * \end{pmatrix}$$ where p^{b_i} represents a bound on the i^{th} coefficient of the logarithmic derivative (up by $\sqrt{N}$), and we have Hensel lifted high enough to ensure that $p^{a-b_i} > 2^{((r+1)^2-(r+1))/2}B^{(r+1)}\sqrt{n}$ for all i. A B-reduction of this matrix will also solve the recombination problem by a similar argument. Thus the switch complexity of factoring over $\mathbb{Q}$ is bounded by $N(r+1)(\log_{4/3}(2^{3(r+1)}B) = \mathcal{O}(Nr^2)$ (our previous scaling makes B only depend on r).

5 Conclusion and Future Improvements

This new switch-complexity of $\mathcal{O}(Nr^2)$ is an improvement over [5] which has switch complexity $\mathcal{O}(N^3)$ if the coefficients are small (more if they are big). Note that our switch complexity is independent of coefficient size.

We tried to create an algorithm that would allow us to bound the LLL switches while not deviating too far from the fastest current implementations like [1]. One major deviance that we will fix in future versions is that we used precise computations over $\mathbb{Q}$ when scaling down the last entries. In practice these numbers are rounded for efficiency reasons to the nearest element of $2^{-d}\mathbb{Z}$ for some d. This could allow our value μ to decrease, and we have to find a value for d that will not hurt the complexity.

We are currently writing down a proof that the switch complexity can be bounded by $\mathcal{O}(r^3)$, independent of both degree and coefficient size. This should be included in the next version of this preprint. information and a proof that only $\mathcal{O}(r)$ traces/coefficients are ever needed. We will work on these proofs for a later version of this paper. Examples indicate that $\mathcal{O}(r^3)$ is asymptotically sharp.

References

[1] K. Belabas *A relative van Hoeij algorithm over number fields*, J. Symbolic Computation, **37** (2004), pp. 641–668.

[2] K. Belabas, M. van Hoeij, J. Klüners, and A. Steel, *Factoring polynomials over global fields*, preprint arXiv:math/0409510v1 (2004).

[3] M. van Hoeij, *Factoring polynomials and the knapsack problem*, J. Number Theory, **95** (2002), pp. 167–189.

[4] A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász, *Factoring polynomials with rational coefficients*, Math. Ann. **261** (1982), pp. 515–534.

[5] A. Schönhage, *Factorization of univariate integer polynomials by Diophantine approximation and an improved basis reduction algorithm*, Proc. ICALP 84, Springer Lec. Notes Comp. Sci. **172**, (1984), pp. 436–447.

[6] H. Zassenhaus, *On Hensel factorization I*, Journal of Number Theory (1969), pp. 291–311.