Practical polynomial factoring in polynomial time

William Hart*
University of Warwick
Mathematics Institute
Coventry CV4 7AL, UK
W.B.Hart@warwick.ac.uk

Mark van Hoeij†
Florida State University
Tallahassee, FL 32306
hoeij@math.fsu.edu

Andrew Novocin
CNRS-INRIA-ENSL
46 Allée d'Italie
69364 Lyon Cedex 07, France
andy@novocin.com

ABSTRACT

State of the art factoring in $\mathbb{Q}[x]$ is dominated in theory by a combinatorial reconstruction problem while, excluding some rare polynomials, performance tends to be dominated by Hensel lifting. We present an algorithm which gives a practical improvement (less Hensel lifting) for these more common polynomials. In addition, factoring has suffered from a 25 year complexity gap because the best implementations are much faster in practice than their complexity bounds. We illustrate that this complexity gap can be closed by providing an implementation which is comparable to the best current implementations and for which competitive complexity results can be proved.

1. INTRODUCTION

Most practical factoring algorithms in $\mathbb{Q}[x]$ use a structure similar to [26]: factor modulo a small prime, Hensel lift this factorization, and use some method of recombining these local factors into integer factors. Zassenhaus performed the recombination step by an exhaustive search which can be made effective for as many as 40 local factors as is shown in [2]. While quite fast for many cases, the exponential complexity of this exhaustive technique is realized on many polynomials.

Polynomial time algorithms, based on lattice reduction, were given in [5, 15, 22]. For a polynomial f of degree N, and entries bounded in absolute value by 2^h, these algorithms perform $\mathcal{O}(N^2(N+h))$ LLL switches. These algorithms are slow in practice, because the size of the combinatorial problem depends only on r, where r is the number of local factors, while the LLL cost depends on N and h, which can be much larger than r. This problem was the motivation for van Hoeij's algorithm [10], but no complexity bound was given. There are several variations on van Hoeij's algorithm, as well as implementations. The most interesting is that of Belabas [4]. His version is designed in such a way that the vectors during LLL have entries with $\mathcal{O}(r)$ bits. In experiments, the number of LLL switches appears to be bounded by $\mathcal{O}(r^3)$. Other implementations, such as NTL and Magma, have a comparable performance in practice.

For a number of years, the practical behavior of these implementations was a complete mystery; there was no complexity bound that came anywhere near the running times observed in practice. There are two ways to reduce the gap between practical performance and theoretical complexity: (a) make the algorithm slower in practice (this was done in [5] because it makes it easier to prove a polynomial time bound), or (b) keep the algorithm at least equally fast in practice and do a more precise analysis.

Note that this $\mathcal{O}(r^3)$ in Belabas' version (and other well tuned implementations) is an observation only, we can not prove it, and suspect that it might be possible to construct counter examples. However, we can prove $\mathcal{O}(r^3)$ after making some modifications to Belabas' version. This was done in [20]. The phrase 'r^3 algorithm' in [20] refers to an algorithm for which we can prove an $\mathcal{O}(r^3)$ bound on the number of LLL switches. The paper [11] explains the lattice reduction techniques in an easier and more general way (more applications) than [20], however, for the application of factoring, the bound in [11] is $\mathcal{O}(Nr^2)$ LLL switches, which is not optimal. So in order to accomplish (b) in the previous paragraph, it remains to show the part "at least equally fast in practice", and this can only be done with an actual implementation.

The goals in this paper are: (1) Implement an algorithm that works well in practice (at least comparable to the best current implementations), that (2) is simpler than the algorithm in [20], and (3) for which the $\mathcal{O}(r^3)$ analysis from [20] still works. In addition, (4) verify the claim in [20] that the so-called early termination strategy can make the implementation faster on common classes of polynomials by doing less Hensel lifting, without hurting the practical performance on the remaining polynomials.

In [20] the metric termed Progress (see Section 2.3) is introduced. The main result of that work is that in order to guarantee $\mathcal{O}(r^3)$ total LLL-switches it is enough to ensure that LLL is only called when a sufficient increase in progress can be made, and that moreover, this is always possible.

*Author was supported by EPSRC Grant number EP/G004870/1

†Supported by NSF Grant numbers 0728853 and 1017880

At every step in our algorithm it is necessary to check that the properties which make the analysis in [20] go through, also hold for the decisions made by our algorithm.

The verification is routine, but it is *not* the aim of this paper to re-cast or simplify the analysis of [20]. Similarly it is not the aim of this paper to report on a highly optimised implementation of the new algorithm. Rather, our goal is to show, with an implementation, that it is possible to have the best theoretical complexity without sacrificing practical performance. A complete complexity analysis of the algorithm we present exists in pre-print format and can be found here [9].

Roadmap Necessary background information will be included in section 2. The algorithm is laid out in an implementable fashion in section 3. Practical notes, including running time and analysis are included in section 4.

2. BACKGROUND

In this section we will outline necessary information from the literature. The primary methods for factoring polynomials which we address can all be said to share a basic structure with the Zassenhaus algorithm of 1969 [26].

2.1 The Zassenhaus algorithm

In some ways this is the first algorithm for factoring polynomials over $\mathbb{Z}$ which properly uses the power of a computer. For background on the evolution of factoring algorithms see the fine treatment in [12].

The algorithm utilizes the fact that the irreducible factors of f over $\mathbb{Z}$ are also factors of f over the p-adic numbers $\mathbb{Z}_p$. So if one has both a bound on the size of the coefficients of any integral factors of f and an irreducible factorization in $\mathbb{Z}_p[x]$ of sufficient precision then one can find the integral factors via simple tests (e.g. trial division). To bound coefficients of integer factors of f we can use the Landau-Mignotte bound (see [7, Bnd 6.33] or [1] for other options). For the p-adic factorization it is common to choose a small prime p to quickly find a factorization over $\mathbb{F}_p$ then use the Hensel lifting method to increase the p-adic precision of this factorization. Due to a comprehensive search of all combinations of local factors the algorithm has an exponential complexity bound which is actually reached by application to the Swinnerton-Dyer polynomials [14].

ALGORITHM 1. *Description of Zassenhaus algorithm*

Input: *Square-free*[1] *and monic*[2] *polynomial* $f \in \mathbb{Z}[x]$ *of degree* N

Output: *The irreducible factors of* f *over* $\mathbb{Z}$

1. Choose a prime, p, such that $\texttt{gcd}(f, f') \equiv 1$ modulo p.
2. **Modular Factorization:** Factor f modulo $p \equiv f_1 \cdots f_r$.
3. Compute the Landau-Mignotte bound $L = \sqrt{(N+1)} \cdot 2^k \cdot \| f \|_\infty$ and $a \in \mathbb{N}$ such that $p^a > 2L$
4. **Hensel Lifting:** Hensel lift $f_1 \cdots f_r$ to precision p^a.
5. **Recombination:** For each $v \in \{0,1\}^r$ (and in an appropriate order) decide if $g_v := \prod f_i^{v[i]}$ mods p^a divides f over $\mathbb{Z}$.

It is common to perform steps 1 and 2 several times to attempt to minimize, r, the number of local factors. Information from these attempts can also be used in clever ways to prove irreducibility or make the recombination in step 5 more efficient, see [2] for more details on these methods. Algorithms for steps 2, 3, and 4 have been well studied and we refer interested readers to a general treatment in [7, Chptrs 14,15]. Our primary interests in this paper lie in the selection of a and the recombination of the local factors in step 5.

2.2 Overview of the LLL algorithm

In 1982 [15] Lenstra, Lenstra, and Lovász devised an algorithm, of a completely different nature, for factoring polynomials. Their algorithm for factoring had a polynomial time complexity bound but was not the algorithm of choice for most computer algebra systems as Zassenhaus was more practical for the majority of everyday tasks. At the heart of their algorithm for factoring polynomials was a method for finding 'nice' bases of lattices now known as the LLL algorithm. The LLL algorithm for lattice reduction is widely applied in many areas of computational number theory and cryptography, as it (amongst other things) gives an approximate solution to the shortest vector problem, which is NP-hard [3], in polynomial time. In fact, the van Hoeij algorithm for factoring polynomials can be thought of as the application of the LLL lattice reduction algorithm to the Zassenhaus recombination step. The purpose of this section is to present some facts from [15] that will be needed throughout the paper. For a more general treatment of lattice reduction see [16].

A lattice, L, is a discrete subset of $\mathbb{R}^n$ that is also a $\mathbb{Z}$-module. Let $\mathbf{b}_1, \ldots, \mathbf{b}_d \in L$ be a basis of L and denote $\mathbf{b}_1^*, \ldots, \mathbf{b}_d^* \in \mathbb{R}^n$ as the Gram-Schmidt orthogonalization over $\mathbb{R}$ of $\mathbf{b}_1, \ldots, \mathbf{b}_d$. Let $\delta \in (1/4, 1]$ and $\eta \in [1/2, \sqrt{\delta})$. Let $l_i = \log_{1/\delta} \| \mathbf{b}_i^* \|^2$, and denote $\mu_{i,j} = \frac{\mathbf{b}_i \cdot \mathbf{b}_j^*}{\mathbf{b}_j^* \cdot \mathbf{b}_j^*}$. Note that $\mathbf{b}_i, \mathbf{b}_i^*, l_i, \mu_{i,j}$ will change throughout the algorithm sketched below.

DEFINITION 1. $\mathbf{b}_1, \ldots, \mathbf{b}_d$ *is* LLL-reduced *if* $\| \mathbf{b}_i^* \|^2 \leq \frac{1}{\delta - \mu_{i+1,i}^2} \| \mathbf{b}_{i+1}^* \|^2$ *for* $1 \leq i < d$ *and* $|\mu_{i,j}| \leq \eta$ *for* $1 \leq j < i \leq d$.

In the original algorithm the values for (δ, η) were chosen as $(3/4, 1/2)$ so that $\frac{1}{\delta - \eta^2}$ would simply be 2.

ALGORITHM 2. *Rough sketch of LLL-type algorithms*

Input: *A basis* $\mathbf{b}_1, \ldots, \mathbf{b}_d$ *of a lattice* L.

Output: *An LLL-reduced basis of* L.

[1] Assumed square-free for simplicity. A standard gcd-based technique can be used to obtain a square-free factorization (see [7, sct 14.6])

[2] We have assumed the polynomial is monic for simplicity. Each algorithm we present can be adapted to handle non-monic polynomials as well.

1. $\kappa := 2$
2. **while** $\kappa \leq d$ **do:**
   (a) *(Gram-Schmidt over $\mathbb{Z}$).* By subtracting suitable $\mathbb{Z}$-linear combinations of $\mathbf{b}_1, \ldots, \mathbf{b}_{\kappa-1}$ from $\mathbf{b}_\kappa$ make sure that $|\mu_{i,\kappa}| \leq \eta$ for $i < \kappa$.
   (b) *(LLL Switch).* If interchanging $\mathbf{b}_{\kappa-1}$ and $\mathbf{b}_\kappa$ will decrease $l_{\kappa-1}$ by at least 1 then do so.
   (c) *(Repeat).* If not switched $\kappa := \kappa + 1$, if switched $\kappa = \max(\kappa - 1, 2)$.

That the above algorithm terminates, and that the output is LLL-reduced was shown in [15]. There are many variations of this algorithm (such as [13, 18, 21, 22, 25]) and we make every effort to use it as a black box for ease of implementation. What we do require is an algorithm which returns an LLL-reduced basis and whose complexity is measured in the number of switches (times step 2a and 2b are called). In fact the central complexity result of [20] is that the r^3 algorithm has $\mathcal{O}(r^3)$ switches throughout the entire algorithm, in spite of many calls to LLL.

Intuitively the Gram-Schmidt lengths of an LLL-reduced basis do not drop as fast as a generic basis (for more on generic bases and LLL see [19]). In practice this property is used in factoring algorithms to separate vectors of small norm from the rest.

2.3 Lattice-based Recombination

Each of [4, 5, 10, 20] use lattice reduction to directly attack the recombination phase of Zassenhaus' algorithm (step 5 in Algorithm 1). The goal of these algorithms is to find **target 0–1 vectors**, $\mathbf{w}_i \in \{0,1\}^r$, which correspond with the true irreducible factors of f over $\mathbb{Z}$, namely $g_i \equiv \prod_{j=1}^{r} f_j^{w_i[j]}$. Each of these algorithms begins with an $r \times r$ identity matrix where each row corresponds with one of the local factors $f_1, \ldots, f_r \in \mathbb{Z}_p[x]$. Then they augment columns and/or rows of data extracted from the corresponding local factors such that:

- The augmented target vectors have boundable norm
- The vectors which do not correspond with factors in $\mathbb{Z}[x]$ can be made arbitrarily large
- This augmented data respects the additive nature of the lattice.

So far, to the best of our knowledge, only traces of the f_i (sums of powers of roots) or so-called CLDs (Coefficients of Logarithmic Derivatives, i.e. $f \cdot f_i'/f_i$) have been used for this purpose. The CLD is important enough that we give a formal definition:

DEFINITION 2. *For a p-adic polynomial $g \in \mathbb{Z}_p[x]$ which divides a polynomial $f \in \mathbb{Z}[x]$ in $\mathbb{Z}_p$, we call the coefficient of x^j in the p-adic polynomial $g' \cdot f/g$ the j^{th} **CLD** of g. This quantity is typically known to some p-adic precision, p^a.*

For example, the rows of the following matrix form the basis of a lattice which could be used:

$$\begin{pmatrix} & & & & p^{a-b_N} \\ & & & \iddots & \\ & & p^{a-b_1} & & \\ 1 & & c_{1,1} & \cdots & c_{1,N} \\ & \ddots & \vdots & \ddots & \vdots \\ & 1 & c_{r,1} & \cdots & c_{r,N} \end{pmatrix}$$

Where $c_{i,j}$ represents the j^{th} CLD of f_i divided by p^{b_j} and p^{b_j} represents $\sqrt{N}$ times a bound of the j^{th} CLD for any factor $g \in \mathbb{Z}[x]$ of f. In this lattice all target vectors have this format: $(\{0,1\}, \ldots, \{0,1\}, \epsilon_1, \ldots, \epsilon_N)$ where ϵ_j is a rational number of absolute value $\leq 1/\sqrt{N}$. These target vectors have a Euclidean-norm $\leq \sqrt{r+1}$, whereas a vector corresponding with a factor in $\mathbb{Z}_p[x] \setminus \mathbb{Z}[x]$ could have an arbitrarily large Euclidean-norm for arbitrarily precise p-adic data.

Brief differences of the algorithms. In [10] the first factoring algorithm to use LLL for the recombination phase was designed and in this case traces were used for the p-adic data. Belabas [4] also used traces, but fine tuned the idea by gradually using this data starting with the most significant bits; this led to more calls to LLL which cost less overall by working on smaller entries. In [5] the idea of using CLDs was introduced, for which we have tighter theoretical bounds than traces. This allowed for an upper bound on the amount of Hensel lifting needed before the problem is solved. Also lattices using CLDs instead of traces tend to have a greater degree of separation between the target vectors and non-targeted vectors at the same level of p-adic precision.

In [20] an approach is outlined which mimics the practical aspects of Belabas while making measures to ensure that the behavior is not harmed when attempting to solve the recombination phase 'too early'. The primary complexity result in [20] is a method of bounding and amortizing the cost of LLL throughout the entire algorithm. This was done by introducing a metric called **Progress** which was to never decrease and which was increased by at least 1 every time any call to LLL made a switch (step 2b of Algorithm 2).

The Progress metric mimicked an energy-function with an additional term to deal with 'removed' vectors, namely:

$$P := 0 \cdot l_1 + 1 \cdot l_2 + \cdots + (s-1) \cdot l_s + (r+1) \cdot n_{\text{rv}} \cdot \log\left(2^{3r}(r+1)\right)$$

Where l_i is the log of the norm of the i^{th} Gram-Schmidt (from here on G-S) vector, s is the number of vectors at the moment, 2^{3r} was the bound on the norm of any vector throughout the process, and n_{rv} is the number of vectors which have been removed from the basis so far.

The factoring algorithm in [20] is then shown to terminate before the progress can cross some threshold of $\mathcal{O}(r^3)$, where r is the number of p-adic factors of f. The Progress metric gives us a method for determining that a call to LLL will be guaranteed to move the algorithm forward. Every decision made by the r^3 algorithm and the algorithm we present here

is tied to ensuring that Progress is never decreased by too much and is increased every time LLL is called.

3. THE MAIN ALGORITHM

In this section we present a modified version of the algorithm presented in Novocin's thesis, [20], but for which the same complexity analysis holds. We divide the algorithm into several sub-algorithms to give a top-down presentation. The sub-algorithms are as follows:

- Algorithm 3 is the top layer of the algorithm. We choose a suitable prime, perform local factorization, decide if the Zassenhaus algorithm is sufficient, perform Hensel lifting, and call the recombination process.
- Algorithm 4 creates a knapsack lattice, processes the local factors (extracts information from CLDs) and tests how much impact on progress they will have. If the progress will be sufficient according to the results of [20] then the CLD information is added to a lattice and LLL is called. We test to see if the algorithm has solved the problem and decide if more Hensel lifting will be needed.
- Algorithm 5 takes CLD data and decides whether or not a call to LLL will make enough progress to justify the cost of the call to LLL. This step guarantees that the complexity analysis of [20] holds.
- Algorithm 6 is a practical method for bounding the size of CLDs arising from 'true factors'. This bound is the analogue of the trace bounds from [10] and gives us some idea of how much Hensel lifting will be needed before a call to LLL can be justified (via the Progress metric).
- Algorithm 7 gives a heuristic 'first' Hensel bound.
- Algorithm 8 decides whether or not we have found a true factorization of f.

3.1 Main Wrapper

The input to the algorithm is a polynomial, $f \in \mathbb{Z}[x]$, (for simplicity, we assume it is monic and square-free). The output is the irreducible factorization of f. The strategy at this level is the same as that of van Hoeij's algorithm and indeed that of Zassenhaus and its variants.

We select a prime, p, for which f is square-free in $\mathbb{Z}_p[x]$ and find the factorization of f modulo p. This step is well understood (see for example [7, Chpts.14,15]). Standard heuristics are used for selecting a 'good' prime.

Next, we perform Hensel lifting of the factors to increase their p-adic precision. We then call a recombination procedure to attempt a complete factorization at the current level of p-adic precision. If this process fails then we Hensel lift again and re-attempt recombination, etc.

ALGORITHM 3. *The main algorithm*

Input: *Square-free, monic, polynomial $f \in \mathbb{Z}[x]$ of degree N*

Output: *The irreducible factors of f over $\mathbb{Z}$*

1. Choose a prime, p, such that $\texttt{gcd}(f, f') \equiv 1$ modulo p.
2. **Modular Factorization:** Factor f modulo $p \equiv f_1 \cdots f_r$.
3. **if** $r \leq 10$ **return** Zassenhaus(f)
4. Compute first target precision a with `Algorithm 7`
5. **until solved**:
   (a) **Hensel Lifting:** Hensel lift $f_1 \cdots f_r$ to precision p^a.
   (b) **Recombination:** $\texttt{Algorithm 4}(f, f_1, \ldots, f_r, p^a)$
   (c) **if** not solved: $a := 2a$

3.1.1 Choosing an initial Hensel precision

Step 4 provides a starting p-adic precision, p^a, by calling Algorithm 7. Other standard algorithms choose this value such that $p^a \geq 2L$ where L is the Landau-Mignotte bound (see [7, sct 14.6]). To the best of our knowledge, this is the lowest known upper-bound precision for which the true factors can be provably reconstructed for every possible input.

Our algorithm attempts recombination at a lower level of p-adic precision, noting that the Landau-Mignotte bound is designed for the worst-case inputs. As section 4 shows our reduced precision is often sufficient, and is substantially lower than other methods.

An example of where this strategy pays dividends is when f can be proven irreducible in the recombination phase at our reduced precision. In this case there is no need to Hensel lift to a higher precision. Another case is when we can reconstruct low-degree factors of f at the current precision and prove the irreducibility of the remainder when f is divided by these small factors. The precision needed to solve the recombination problem and the precision needed to reconstruct integer factors once the recombination is solved are unrelated. Further the precision needed to solve the recombination is not well understood, there is a theoretical worst-case in [5] which has never been reached in practice.

The worst-case for our algorithm is when either the recombination requires the same or more p-adic precision than the Landau-Mignotte bound or when the true factorization has two or more factors of large degree with large coefficients (in which case they each require precision near the Landau-Mignotte bound and they cannot all be discovered by division). We do not know, a priori, which case we will be in, so we design the algorithm to ensure that in the worst case we do no worse than other algorithms and in the best case we minimize Hensel lifting.

We designed our Hensel lifting procedure for the case that we need to increase the precision frequently. Our implementation uses a balanced factor tree approach as presented in [7, Sect. 15.5]. To minimize overhead in Hensel lifting multiple times our implementation caches the lifting tree, intermediate modular inverses computed by the extended gcd, and the intermediate products of the lifting tree itself. This way there is little difference between Hensel lifting directly to the end precision or lifting in several separate stages.

3.2 Recombination

The next several sub-algorithms form the core the new approach. In Algorithm 4 we are given a local factorization at a new p-adic precision and (except for the first call to this sub-algorithm) we are also given an LLL-reduced lat-

tice. This is the layer of the algorithm which organizes all of the lattice decisions, including the creation of new columns and/or rows from the p-adic factorization, the decision as to when lattice reduction is justified, the lattice reduction itself, and the extracting of factors from the information in the reduced lattice.

ALGORITHM 4. *Attempt Reconstruction*

Input: *f, $f_1, \ldots, f_r$ the lifted factors, their precision p^a, and possibly $M \in \mathbb{Z}_{s\times(r+c)}$.*

Output: *If solved then the irreducible factors of f over $\mathbb{Z}$ otherwise an updated M.*

1. If this is the first call let $M := I_{r\times r}$
2. Choose a k heuristically (see below for details)
3. For $j \in \{0, \ldots, k-1, N-k-1, \ldots, N-1\}$ do:
   (a) Compute CLD bound, X_j, for x^j using Algorithm 6
   (b) If $\sqrt{N} \cdot X_j \leq p^a/2^{1.5r}$ then compute new column vector $\mathbf{x}_j := (x_{1,j}, \ldots, x_{r,j})^T$ where $x_{i,j}$ is the coefficient of x^j in $f \cdot f_i'/f_i$
4. For each computed $\mathbf{x}_j$ do:
   (a) `justified`:= True; While `justified` is True do:
      i. Decide if LLL is justified using Algorithm 5 which augments M
      ii. If so then run LLL(M)
      iii. If not then `justified` := False
      iv. Compute G-S lengths of rows of M
      v. Decrease the number of rows of M until the final Gram-Schmidt norm $\leq \sqrt{r+1}$
      vi. Use Algorithm 8 to test if solved

This algorithm provides the basic framework of our attack. The rows of the matrix M provide the basis of our lattice-based recombination (see section 2.3). We compute bounds for the $2k$ CLDs, $\{0, \ldots, k-1, N-k-1, \ldots, N-1\}$, from these bounds we can determine if p^a is a sufficient level of p-adic precision to justify computing any of the $2k$ actual CLDs.

For each CLD which is actually computed we call Algorithm 5 to decide what to do. Details are given in the next section. Steps 4(a)iv and 4(a)v are the same as both [10] and [20], but we note that step 4(a)iv can be done with a well-chosen floating-point algorithm since M is LLL-reduced.

The heuristic k. In practice we need not compute all of the coefficients of the r p-adic polynomials $f_i' \cdot f/f_i$. Often only a few coefficients are needed to either solve the problem or decide that more precision will be needed. The value of k provides a guess at the number of coefficients which will be needed and can be determined experimentally. In our implementation we found that a value of $5 \leq k \leq 20$ was usually sufficient. A fail-safe value of $k = N/2$ can be used for the cases when the p-adic precision is close to the theoretical bound and where the problem is yet to be solved (which did not occur for us in practice).

It is best to compute the highest k coefficients and/or the lowest k coefficients of each logarithmic derivative. There are two reasons for this:

- To compute the bottom k coefficients of $f_i' \cdot f/f_i$ mod p^a, power series techniques can be used. Thus only the bottom k coefficients of f_i and f are needed (same for the top k coefficients) rather than all coefficients.
- The heuristic we use in Algorithm 7, for initial p-adic precision, only Hensel lifts far enough to guarantee that either the leading CLD or trailing CLD can justify a call to LLL.

The soundness of these heuristics is checked by examining the CLD bounds for true factors and comparing them with p^a. If our heuristics are well-adjusted then some of the computed $2k$ CLD bounds will be smaller than p^a. These CLD bounds are cheap to compute and a method is provided in Algorithm 6. Of course other choices are possible for each of our heuristics, and a more sophisticated heuristic could be designed.

3.3 Determining if a potential column justifies LLL

The following algorithm is given both a column vector whose i^{th} entry is the j^{th} CLD i.e. the coefficient of x^j in the p-adic polynomial $f_i' \cdot f/f_i$ (which is known to a precision p^a) and a matrix M. The rows of M form a reduced basis of a lattice which contains small vectors corresponding to irreducible factors of f over $\mathbb{Z}$.

The algorithm decides if augmenting M by an appropriate transformation of the given column vector would increase the norm of the rows of M by enough to justify a call to LLL on the augmented M. The metric used is the Progress metric of [20]. This algorithm also performs scaling in the style of [4], to prevent entries of more than $\mathcal{O}(r)$ bits in M.

This sub-algorithm is important to the proven bit-complexity of the algorithm and not to the practical complexity. The purpose of this sub-algorithm is to bound the timings of all potential worst-cases via the theoretical analysis of [20]. One of the important contributions of this paper is to show, via implementation, that this sub-algorithm does not harmfully impact the performance of our algorithm.

ALGORITHM 5. *Decide if column is worth calling LLL*

Input: *$M \in \mathbb{Z}_{s\times(r+c)}$, data vector $\boldsymbol{x}_j$, p^a, X_j the CLD bound for x^j*

Output: *A potentially updated M and a boolean `justified`*

1. Let $B := r+1$ and s be the number of rows of M
2. If $p^a < X_j \cdot B \cdot \sqrt{N} \cdot 2^{(1.5)r}$ `justified` := *False* and exit
3. Find U the first r columns of M
4. Compute $\mathbf{y}_j := U \cdot \mathbf{x}_j$
5. If $\| \mathbf{y}_j \|_\infty < X_j \cdot B \cdot \sqrt{N} \cdot 2^{(1.5)r}$ then `justified` := *False* and exit
6. If $p^a - Bp^a/2^{(1.5)r} > \| \mathbf{y}_j \|_\infty \cdot (2(3/2)^{s-1} - 2)$ then no_vec :=True otherwise False
7. Find new column scaling 2^k either $\frac{\|\mathbf{y}_j\|_\infty}{X_j \cdot B \cdot \sqrt{N} \cdot 2^{(1.5)r}}$ if no_vec is True or $\frac{p^a}{X_j \cdot B \cdot \sqrt{N} \cdot 2^{(1.5)r}}$ if False

8. Embed $\mathbf{x}_j$ and $p^a/2^k$ into $\mathbb{Z}/2^r$ by rounding and denote results as $\tilde{\mathbf{x}}_j$ and $\tilde{P}$
9. If no_vec is True then augment M with new column $\tilde{\mathbf{y}}_j = U \cdot \tilde{\mathbf{x}}_j$
   If no_vec False then also adjoin a new row so $M := \left[\begin{array}{c|c} \mathbf{0} & \tilde{P} \\ \hline M & \tilde{\mathbf{y}}_j \end{array}\right]$
10. `Justified` :=True; return M

The most significant change of this algorithm from the algorithm in [20] is that we round the new column after scaling. We keep $\log r$ bits after the decimal for the sake of numerical stability. Consider this change a practical heuristic which we can prove does not impact the $\mathcal{O}(r^3)$ bound for the number of LLL switches (see [9]). This proof uses the bounds we have on the size of unimodular transforms encountered throughout the algorithm.

As some implementations of LLL prefer to work on matrices with integer entries we note that a virtual decimal place can be accomplished using integers by scaling up the entries in U (the first r columns) by 2^r. Such a scaling requires replacing $\sqrt{r+1}$ with $\sqrt{2^{2r}(r+1)}$ in step 4(a)v of Algorithm 4.

3.4 Obtaining practical CLD bounds

The goal of this sub-algorithm is to quickly find a bound for the absolute value of the coefficient of x^j in any integer polynomial of the form $g' \cdot f/g$ where $g \in \mathbb{Z}[x]$ divides $f \in \mathbb{Z}[x]$. This bound, which we will frequently call the j^{th} **CLD bound**, is the CLD equivalent of the Landau-Mignotte bound. Although in the CLD case we have a different bound for each possible j.

The following method (an analogous bound for the bivariate case is given in [5, Lemma 5.8]) quickly gives fairly tight bounds in practice. The method is based on the fact that $g'f/g = \prod_{\alpha \mid g(\alpha)=0} \frac{f}{x-\alpha}$ summed over all roots of the potential factor.

ALGORITHM 6. *CLD bound*

Input: $f = a_0 + \cdots + a_N x^N$ *and* $c \in \{0, \ldots, N-1\}$

Output: X_c*, a bound for the absolute value of the coefficient of* x^c *in the polynomial* fg'/g *for any* $g \in \mathbb{Z}[x]$ *dividing* f*.*

1. Let $B_1(r) := \frac{1}{r^{c+1}}(|a_0| + \cdots + |a_c| r^c)$
2. Let $B_2(r) := \frac{1}{r^{c+1}}(|a_{c+1}| r^{c+1} + \cdots + |a_N| r^N)$
3. Find $r \in \mathbb{R}^+$ such that $\text{MAX}\{B_1(r), B_2(r)\}$ is minimized to within a constant.
4. return $X_c := N \cdot \text{MAX}\{B_1(r), B_2(r)\}$

In this method, for any positive real number r, either $B_1(r)$ or $B_2(r)$ is an upper bound for the coefficient of x^c in $\frac{f}{x-\alpha}$ for any possible complex root α (because of the monotonicity of B_1 and B_2, if $r \leq |\alpha|$ then $B_1(r)$ is the upper bound and if $r \geq |\alpha|$ then $B_2(r)$ will be). Thus for every positive real number r the quantity $\text{MAX}\{B_1(r), B_2(r)\}$ is an upper bound for the coefficient of x^c in $\frac{f}{x-\alpha}$. The task is then to find an r for which $\text{MAX}\{B_1(r), B_2(r)\}$ is minimized. Since the CLD is summed over every root of g we use N as a bound for the number of roots of g to give a CLD bound of $N \cdot \text{MAX}\{B1(r), B2(r)\}$.

For finding an r which minimizes $\text{MAX}\{B1(r), B2(r)\}$ our floating-point method is as follows (where sign(x) is 1 if x positive, -1 if x negative, and 0 otherwise).

1. Let $r := 1$, `scaling` $:= 2$, `cur_sign` $:= \text{sign}(B1(r) - B2(r))$, and `pos_ratio` $:= \left(\frac{B1(r)}{B2(r)}\right)^{\texttt{cur\_sign}}$
2. Until `cur_sign` changes or `pos_ratio` ≤ 2 do:
   (a) $r := r \cdot \texttt{scaling}^{\texttt{cur\_sign}}$
   (b) `cur_sign`:= $\text{sign}(B1(r) - B2(r))$
   (c) `pos_ratio` $:= \left(\frac{B1(r)}{B2(r)}\right)^{\texttt{cur\_sign}}$
3. If `pos_ratio` > 2 then `scaling` $:= \sqrt{\texttt{scaling}}$ and Go to step 2 Otherwise Return r.

Another method of finding r is simply solving $B_1(r) - B_2(r) = 0$ for which many Computer Algebra Systems have efficient implementations. Our method is a quick-and-dirty method which is good enough in practice.

3.5 The new starting precision heuristic

We now outline our suggested heuristic for selecting an initial p-adic precision, a. This heuristic is designed so that we lift just far enough to warrant at least one call to LLL from either the 0^{th} CLD or the $(N-1)^{\text{st}}$ CLD, which can be enough to solve the problem.

ALGORITHM 7. *Heuristic for initial precision*

Input: $f \in \mathbb{Z}[x]$, p

Output: *Suggested target precision* a

1. Use Algorithm 6 to compute b, the minimum of (CLD bound for x^0) and (CLD bound for x^{N-1}).
2. return $a := \left\lceil \frac{2.5r + \log_2 b + (\log_2 N)/2}{\log_2 p} \right\rceil$

This heuristic is focused on either the trailing coefficient or the leading coefficient of f and guarantees that at least one CLD is computed in step 3b of Algorithm 4 and will be used by Algorithm 5.

3.6 Checking if problem solved

Finally we briefly mention the new method in which we check for true factors. One of the central novelties to the algorithm is a reduced level of Hensel lifting when attempting to solve the problem. It has been observed that the Landau-Mignotte bound is often too pessimistic and that even Zassenhaus' algorithm could potentially terminate at a lower level of Hensel lifting. This is also true of our lattice based attack, as we can often prove the irreducibility of a potential factor before we can fully reconstruct each of its coefficients. This is seen most frequently in polynomials which turn out to have one large degree factor and zero or more smaller degree factors. In these cases we must check for true factors in a way that will recover the large factor by dividing away any small irreducible factors.

We will begin by using a short-cut for detecting a 0-1 basis of our lattice. Such a basis, if it exists, could potentially solve the recombination problem.

ALGORITHM 8. *Check if solved*

Input: *M, f, $f_1, \ldots, f_r$ to precision p^a*

Output: *A Boolean,* `solved`*, and possibly the irreducible factors of f in $\mathbb{Z}[x]$*

1. Sort the first r columns of M into classes of columns which are identical
2. If there are not more classes than there are rows of M then we have a potential solution otherwise `solved` := $False$ and exit
3. For each class multiply the p-adic polynomials corresponding with the columns in that class and reduce with symmetric remainder modulo p^a to find the potential factors
4. In order, from the lowest degree to the highest degree, perform trial divisions of f
5. If any two polynomials fail to divide f then `solved` := $False$ and exit
6. `solved` := $True$ if there is one failed polynomial then recover it by division of f by the successful factors

The goal of lattice-based recombination is to find the target 0–1 vectors shown in section 2.3. It is possible that we have a basis whose echelon form gives these 0–1 vectors. Since we know that a solution can only use each local factor once then any echelon form solution will have a unique 1 in each of the first r columns. We detect this by examining which columns are identical. The symmetric remainder of step 3 is required to capture polynomials with negative coefficients. By moving from the lowest degree to the highest degree we maximize the chances of solving the problem with less Hensel lifting than the Landau-Mignotte bound used in Zassenhaus.

4. RUNNING TIMES AND PRACTICAL OBSERVATIONS

In this section we illustrate that our algorithm is useful in practice and give direct evidence that our approach can factor successfully at a lower level of p-adic precision than the previous standard based on the Landau-Mignotte bound. Here we make progress on the complexity gap by illustrating that an algorithm for which a good complexity result exists (see [9]) can match (and sometimes exceed) the performance of the best current implementations for polynomial factoring. We do this by providing running times of our implementation side by side with highly polished implementations in NTL version 5.5.2 as well as the implementation in MAGMA version 2.16-7. We also provide the level of p-adic precision at which the routines completed the factorization. This is done for a collection of benchmark polynomials from the literature [4] which was collected by Paul Zimmerman and Mark van Hoeij. It is tempting to judge a new algorithm on its performance on the diabolical Swinnerton-Dyer polynomials (which are amongst our benchmarks), however the practical bottleneck for standard polynomials tends to be the cost of Hensel lifting (see [4, pg.11]). Because of this we wish to emphasize that our algorithm successfully terminates with far less Hensel lifting than the others.

Poly	MAG	NTL	Z-bnd	FLINT	N-bnd
P1	.03	.068	29^{311}	.096	89^{33}
P2	.08	.204	11^{437}	.104	11^{44*}
P3	.16	.312	11^{629}	.164	11^{62*}
P4	1.87	1.956	13^{745}	.940	7^{160*}
P5	.11	.088	19^{51}	.044	23^{26}
P6	.11	.12	19^{152}	.108	23^{76*}
P7	1.07	1.136	37^{78}	.524	19^{74}
P8	2.18	3.428	13^{324}	1.532	11^{84}
M12_5	9.54	12.429	13^{1171}	2.88	11^{180}
M12_6	21.49	21.697	13^{1555}	5.18	13^{380*}
S7	.42	.340	29^{78}	.20	47^{41}
S8	4.62	3.752	47^{140}	2.06	53^{79}
S9	165.2	71	137^{228}	20.7	149^{125}
T1	2.54	3.848	7^{495}	1.23	7^{40}
T2	2.06	3.18	7^{200}	1.25	7^{43}
T3	19.7	24.27	7^{984}	7.35	7^{82}
P7*M12_6	240m	–	19^{1438}	53m	29^{265}
M12_5*P7	145m	–	19^{1114}	78m	29^{236}

These timings are measured in seconds (unless otherwise stated) and were made on a 2400MHz AMD Quad-core Opteron processor, using gcc version 4.4.1 with the -O2 optimization flag, although the processes did not utilize all four cores.

These timings show that our algorithm, a simplification of the r-cubed algorithm of [20], can be comparable in practice to [10] on benchmark polynomials (and in most of the cases a bit faster). Also of interest are the columns labeled 'Z-bnd' (for Zassenhaus bound) compared with 'N-bnd' (for New bound) which give the p-adic precision at which the algorithms solved the factoring problem (MAGMA and NTL chose the same primes and precisions). In the case of the five polynomials which include a $*$, the level of Hensel lifting shown was sufficient to reconstruct all of the factors but the combinatorial problem was actually solved at half of the shown precision. In every case we were able to solve the factorization with significantly less Hensel lifting than the Landau-Mignotte bound used in NTL and MAGMA. We stopped the NTL factorizations of P7*M12_6 and P7*M12_5 after some time.

We designed the degree 900 polynomials T1 and T2 and degree 2401 polynomial T3 to illustrate 'everyday' factoring problems which a user of a number theory library might naturally face. They are taken from factorizations inside of Trager's Algorithm, and the Hensel lifting dominates the running times. Such examples are plentiful (see the discussion in section 3.1.1) and arise naturally in many applications. Observe the large gap in p-adic precisions between our method and the others. Our implementation is open source and publicly available as of FLINT version 1.6 [8].

4.1 A new floating point LLL trick

We implemented a floating-point based LLL, in FLINT, using the template of [18]. In the process we noted that the performance of our algorithm was devastated when multi-limb floating-point computations were needed. In order to avoid this we implemented a new wrapper of floating-point LLL which avoids needing multi-precision floating points, we

call it `U-LLL`. We present it here as this was the application it was developed for.

ALGORITHM 9. `U-LLL`

Input: *M whose rows form a lattice in $\mathbb{R}^n$,* `step_size`

Output: *An updated M which is LLL-reduced in place*

1. **loop**:
   (a) `max_size` $=\| M \|_\infty$
   (b) Find smallest integer $k \geq 0$ such that $\| M/2^k \|_\infty \leq 2^{\texttt{step\_size}}$
   (c) if $k == 0$ then **break**
   (d) Let $M_{\text{temp}} := \lfloor M/2^k \rceil$ rounded
   (e) $B := [M_{\text{temp}}|I]$ augment an identity matrix
   (f) $\text{LLL}(B) = [UM_{\text{temp}}|U]$
   (g) $M := UM$
   (h) if $\| M \|_\infty >$ `max_size`$/2^{\texttt{step\_size}/4}$ then **break**
2. $M :=\text{LLL}(M)$

The algorithm will always output an LLL-reduced M and if each iteration of truncation continues to decrease the bit-length of M then this algorithm will loop until the bit-size of M is small. The advantage of this approach is that the augmented identity on a lattice of small bit-size tends to be numerically stable as the matrix is well conditioned. This allowed us to avoid needing expensive multi-precision floating point computations in our lattice reductions by choosing a sensible `step_size`.

5. REFERENCES

[1] J. Abbott *Bounds on Factors in Z[x]* arXiv:0904.3057

[2] J. Abbott, V. Shoup and P. Zimmermann, *Factorization in $\mathbb{Z}[x]$: The Searching Phase*, ISSAC'2000 Proceedings, 1–7 (2000).

[3] M. Ajtai *The shortest vector problem in L2 is NP-hard for randomized reductions*, STOC 1998, pp. 10–19.

[4] K. Belabas *A relative van Hoeij algorithm over number fields*, J. Symb. Comp. **37** 2004, pp. 641–668.

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

[6] J. J. Cannon, W. Bosma (Eds.) *Handbook of Magma Functions*, Edition 2.13 (2006)

[7] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra.* Cambridge University Press, 1999.

[8] W. Hart *FLINT*, open-source C-library. `http://www.flintlib.org`

[9] W. Hart, M. van Hoeij, A. Novocin, *Complexity analysis of factoring polynomials*, work-in-progress, `http://andy.novocin.com/pro/complexity.pdf`

[10] M. van Hoeij, *Factoring polynomials and the knapsack problem*, J. Num. The., **95** 2002, pp. 167–189.

[11] M. van Hoeij and A. Novocin, *Gradual sub-lattice reduction and a new complexity for factoring polynomials*, accepted for proceedings of LATIN 2010.

[12] E. Kaltofen, *Polynomial factorization.* In: Computer Algebra, 2nd ed„ editors B. Buchberger et all, Springer Verlag, 95–113 (1982).

[13] E. Kaltofen,*On the complexity of finding short vectors in integer lattices*, EUROCAL'83,LNCSv.162,pp.235–244

[14] E. Kaltofen, D.R. Musser, and B.D. Saunders *A generalized class of polynomials that are hard to factor*, SIAM J. Comput.,12(2),pp.473–485 (1983).

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

[16] L. Lovász, *An Algorithmic Theory of Numbers, Graphs and Convexity*, SIAM 1986

[17] I. Morel, D. Stehlé, and G. Villard, *H-LLL: Using Householder Inside LLL*, ISSAC'09, pp. 271–278.

[18] P. Nguyen and D. Stehlé, *Floating-point LLL revisited*, Eurocrypt 2005, v. 3494 LNCS, pp. 215–233.

[19] P. Nguyen and D. Stehlé, *LLL on the Average*, ANTS VII 2006, v. 4076 LNCS, pp. 238–256.

[20] A. Novocin, *Factoring Univariate Polynomials over the Rationals*, Ph.D. Thesis Flor. St. Univ. 2008, `http://andy.novocin.com/pro/dissertation.pdf`.

[21] C. P. Schnorr, *A more efficient algorithm for lattice basis reduction*, J. of Algo. **9** 1988, pp. 47–62.

[22] A. Schönhage, *Factorization of univariate integer polynomials by Diophantine approximation and an improved basis reduction algorithm*, ICALP'84, LNCS **172**, pp. 436–447.

[23] V. Shoup *NTL*, open-source C++ library. `http://www.shoup.net/ntl/`

[24] W. Stein *SAGE Mathematics Software.* `http://www.sagemath.org`

[25] A. Storjohann, *Faster algorithms for integer lattice basis reduction*, Technical report 1996, ETH Zürich.

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