Factorization of Univariate Polynomials with Rational Coefficients

Andrew Novocin & Mark van Hoeij
Department of Mathematics, Florida State University

Summary of Zassenhaus' Algorithm

Let $f \in \mathbb{Z}[x]$ be separable and monic with degree N. **Goal:** the factors of f in $\mathbb{Z}[x]$.

Idea 1: If $g \in \mathbb{Z}[x]$ divides f then the coefficients of g are smaller than some computable **bound** L.

Idea 2: If $g \in \mathbb{Z}[x]$ divides f then g can be **reconstructed** when g mod p^a is known for some $p^a > 2L$.

Idea 3: Factor $f = f_1 \cdots f_r$ over $\mathbb{Z}_p$ (the p-adic integers). There are only **finitely many** monic factors of f in $\mathbb{Z}_p[x]$. Each is of the form

$$g_v := \prod f_i^{v_i}$$

for some 0–1 vector $v = (v_1, \ldots, v_r)$.

Idea 4: $f_1, \ldots, f_r$ (and hence g_v) are not known exactly, but are only known mod p^a. That's enough using idea 2. In practice we find $f_1, \ldots, f_r$ mod p and Hensel Lift until they are known mod p^a.

Problem 1: Complexity of the Algorithm

Exponential Search Time: When there are many local factors $f_1, \ldots, f_r$ it can take an exponentially long time to decide which of the local factors combine to form a rational factor g_i. The [van Hoeij] algorithm is a practical solution to this problem but no attempt at a complexity estimate was made.

The [BHKS] paper gave polynomial time complexity results for two versions of the van Hoeij algorithm, a slow version that uses one large lattice reduction, and a fast practical version given in [Belabas] that uses many small lattice reductions. Still, these complexity results are not satisfactory because they did not describe the actual behavior of the algorithm: the fast version received a complexity result that was worse than the complexity result given for the slow version!

Key Theoretical Problem: *The complexity results for the fastest practical algorithm do not describe the actual behavior of that algorithm.*

Our work on this theoretical issue [Novocin 2008] has led to the resolution of a practical problem mentioned below.

Problem 2: Overshooting the Hensel Lifting

Let $g_1, \ldots, g_k$ be the true factors of f in $\mathbb{Z}[x]$ and let $f_1, \ldots, f_r$ be the local factors (over the p-adic integers). Current implementations Hensel lift to determine $f_1, \ldots, f_r$ with a p-adic accuracy a that is guaranteed to be high enough to recover any potential factor of f in $\mathbb{Z}[x]$.

However, the problem is that this p-adic accuracy, a, is often much higher than what was actually necessary to recover all the factors $g_1, \ldots, g_k$. This implies that current implementations often waste CPU time on Hensel Lifting. In practice it frequently happens that f has one large factor, say g_1, and zero or more small factors, say $g_2, \ldots, g_k$. Then, to recover $g_1, \ldots, g_k$ we do not need p^a to be larger than twice the largest coefficient of g_1. All we need is that p^a is larger than twice the largest coefficient in $g_2, \ldots, g_k$. This suffices to reconstruct $g_2, \ldots, g_k \in \mathbb{Z}[x]$ from their modular images, after which the remaining factor g_1 can be determined by a division in $\mathbb{Z}[x]$.

It is easy to give examples where this latter a is ten times smaller than the a used in Zassenhaus' algorithm. Just multiply a small irreducible polynomial by a big one. (Of course a needs to be large enough not only to find $g_2, \ldots, g_k$, but also large enough to prove that $g_1, \ldots, g_k$ are irreducible. More precisely, a needs to be large enough to solve the combinatorial problem. However, using [van Hoeij] this can usually be done with a much smaller a than what is used in Zassenhaus' algorithm.)

An easy way (called **Early Termination**) to prevent lifting too far is to do these two steps after each Hensel lift:

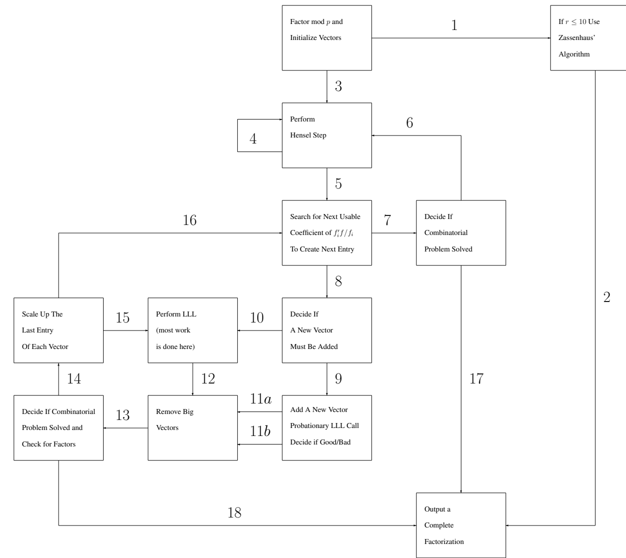
1. Try to solve the combinatorial problem using [van Hoeij]
2. and if this succeeds, try to reconstruct $g_2, \ldots, g_s$ from their modular images (and g_1 with a division).

Suppose that lifting to at least p^{100} was necessary to solve both steps 1 and 2. We use quadratic Hensel lifting, so a doubles each step. This means that we solve the problem once we lifted to p^{128}, which is close to optimal. So compared to Zassenhaus' algorithm we could save much CPU time on Hensel lifting (Hensel lifting often dominates the CPU time).

Key Practical Problem: *But couldn't early termination also be slower in some cases?* After all: What about the time that was spent when step 1 or 2 failed when we lifted to p^{64}, or to p^{32}, etc.? Step 2 costs little, but if step 1 failed by not lifting far enough, couldn't we have wasted CPU time?

This practical problem is the reason that current implementations do not use early termination. The beauty of our work on the theoretical complexity is that it solves this practical problem as well.

The r^3 Algorithm:



Bounding the Dominant Complexity Term: LLL Costs

The number of CPU operations used in an LLL call is dependent on the number of LLL switches plus some small overhead cost for each call. So to bound the LLL cost we will prove a bound for the total number of LLL switches. We achieve this by creating a measure to bound the number of LLL switches which we call Progress P:

$$P := 1 \cdot l_1 + \cdots + (s) \cdot l_s + \frac{3r}{2} \cdot n_{\text{bad}} + \frac{3r}{2} \cdot 2r \log_{\sqrt{4/3}}(2) \cdot n_{\text{rm}}$$

Here l_i is the logarithmic Gram-Schmidt Length of the vectors LLL is working on, while n_{rm} and n_{bad} are counters for removed vectors. The strategy of our switch bound is to show that, throughout the algorithm, the number of LLL switches, n_{switches}, is always $\leq P$, and that $P \leq 68r^3$ at the algorithm's termination [Novocin, 2008]. In order to show this we need some technical properties to hold throughout the algorithm.

Properties true at every exit arrow

Property	Meaning
1. $s < \lfloor \frac{3r}{2} \rfloor$	$s = r + n_{\text{good}} - n_{\text{rm}}$ is the dimension of L
2. $n_{\text{good}} \leq 3r + 2$	n_{good} counts 'good' coefficients (Exit $11a$)
3. $n_{\text{bad}} \leq 3r^2 - 2r + 1$	n_{bad} counts 'bad' coefficients (Exit $11b$)
4. $n_{\text{novec}} \leq 3r + 2$	n_{novec} counts 'no vector' coeffs (Exit 10)
5. $n_{\text{rm}} \leq r + n_{\text{good}} - 1 \leq 4r + 1$	n_{rm} counts the removed vectors
6. $n_{\text{scales}} \leq 3r + 2$	n_{scales} counts successful scalings (Exit 15)
7. $n_{\text{entries}} \leq 3r^2 + 5r + 5$	n_{entries} counts entries in b_i, i.e. $L \subseteq \mathbb{Z}^{n_{\text{entries}}}$
8. $n_{\text{switches}} \leq P$	n_{switches} counts LLL switches throughout
9. $P^{\text{out}} \geq P^{\text{in}}$	Progress, P, is how we bound n_{switches}
10. $W \subseteq \pi(L)$	W is the solution lattice in $\mathbb{Z}^r$
11. $\parallel \pi^{-1}(v_g) \parallel^2 \leq r + 1$ for any irreducible factor g	This is how we ensure $W \subseteq \pi(L)$

The basic strategy for proving these properties throughout the algorithm is to check them inductively at each box of the flow chart. We assume the properties hold true at the beginning of a procedure and show that they still hold at the end of the procedure.

This Complexity Result Explains The Algorithm's Behavior

As already mentioned, [BHKS] gave the worst complexity result for the faster version of the algorithm. The explanation is that the complexity work in [BHKS] does not reflect some important features of the algorithm. These features are the following previously unexplained observations about this algorithm:

- According to Section 2.5.1 in [Belabas], if the parameter BitsPerFactor in Algorithm 2.3 is reduced a factor 2, it has little impact on the CPU time. However, this would double the number of LLL calls. So the observation is that the number of LLL calls has little impact on the CPU time. This observation is not reflected in the [BHKS] complexity result, but is reflected in our complexity result since our bound of $68r^3$ LLL switches is independent of how many LLL calls are made.
- In section 2.5.3 of [Belabas] there is an example which uses 62 LLL calls, most of the CPU time for this example is spent during only 11 of the calls to LLL. This illustrates that a small subset of the LLL calls can dominate the LLL cost. This observation can now be understood with our complexity result; the switch complexity bound for each individual LLL call, which is $\mathcal{O}(r^3)$, is up to a constant the same as our bound for all LLL calls combined. In our bound, we do not know a priori which LLL calls will perform the bulk of the work, all we can do is bound the total cost.

The example in section 2.5.3 in [Belabas] helped us because it illustrated that in order to get a good complexity result, we should not calculate "bound # LLL calls" times "bound cost of each LLL call". The above two observations gave a strong clue that we needed to search for a switch complexity bound (for all LLL calls combined) that is independent of the number of LLL calls. Finding such a bound explains both observations. So the theoretical complexity work gives practical insight about how the [van Hoeij] style factoring algorithms (specifically the fastest versions) really work!

Further, the design of our algorithm allows for the **Early Termination** feature to be included without harming the switch complexity.

Why This is Interesting

There used to be a gap between the best factoring algorithm in theory, and the best algorithm in practice. Before 2001, the Zassenhaus algorithm performed best in practice, while LLL/Schönhage had the best theoretical complexity. This gap between theory and practice grew even wider with [van Hoeij 2002] because this algorithm was even faster in practice, while even worse in theory ([van Hoeij 2002] contains no complexity bound).

Then in [BHKS] the gap was made smaller; polynomial time complexity bounds for two versions of the [van Hoeij 2002] algorithm were given. However, the gap between theory and practice remained very large because the version that was faster in practice received the worse theoretical bound!

We can now resolve this unfortunate situation. We have made the (in practice) fastest version even faster, in practice, by saving time on Hensel lifting. For this (in practice) fastest version we have proved a bound for the switch-complexity that is asymptotically sharp. This bound perfectly captures the actual behavior of the algorithm, in fact, the progress P could even be used to give a realistic progress bar!

We do not merely reduce the gap between theory and practice; we eliminate the gap altogether. *The algorithm that is best in practice, and the algorithm that is best in theory, are now the same algorithm.*

Our theoretical work resulted in more than just a better bound for the complexity of factoring. It also allowed us to solve the key practical problem (mentioned earlier) in designing an efficient early termination algorithm. Recall that the key practical problem was wasting time on attempts that were "unsuccessful" because we had not lifted far enough. We can now solve this problem by designing the algorithm with the measure P in mind, we just have to ensure that P increases with every LLL switch, and that we take no steps that could decrease P.

References

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

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

M. van Hoeij, *Factoring polynomials and the knapsack problem*, J. Number Theory, **95** (2002).

M. van Hoeij and A. Novocin, *Complexity results for factoring univariate polynomials over the rationals*, preprint (2007).

A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász, *Factoring polynomials with rational coefficients*, Math. Ann. **261** (1982).

A. Novocin, *Factoring Univariate Polynomials over the Rationals*, Florida State University, PhD Thesis (2008).

H. Zassenhaus, *On Hensel factorization I*, Journal of Number Theory (1969).