# Modular Methods in CoCoA

## What are modular methods?

When you have to do a quick calculation on the back of an envelope, you might calculate the sum or product of two (small) polynomials, and you would most likely use a **direct method**, *i.e.* all computations and intermediate results are in the same ring.

A **modular method** is a more indirect way of calculating: to effect a computation in a ring $R$, we do most of the work in one or more quotient rings $R/I_1, R/I_2, \ldots$ and then reconstruct the answer back in $R$. There are two main reasons for doing this: it turns out to be faster than a direct approach; or we do not know any direct way of finding the answer.

An example is a (partial) test of divisibility for $\mathbf{Z}[x]$. To test whether $f$ divides $g$ we could simply apply long division, and check that the remainder is zero. But the long division could be costly if $f$ and $g$ are large. A much quicker partial test would be to see whether $f(0)$ divides $g(0)$; if not then surely $f$ cannot divide $g$. Similarly, we could check the values at $x = 1$ and $x = -1$. Only if all these partial checks pass, do we embark on the lengthy long-division computation. This is a (partly) modular method because evaluating a polynomial at $x = \alpha$ is the same as reducing it modulo $(x - \alpha)$.

Modular methods can be applied to many computations in the rings of integers, rational numbers or algebraic numbers, and to computations involving polynomials or matrices over these rings. They cannot be used with approximate data, nor when the arithmetic ordering on numbers is important.

Some modular methods are **probabilistic**: that is, the answer computed may be correct only with a certain probability. The probabilistic methods offer greater speed than their deterministic counterparts, but the result should be checked. In some cases it is possible to estimate the probability of obtaining a wrong result — if the probability is low enough, you may be willing to skip the verification of the result.

There are two common problems associated with modular methods. One is **bad reduction** when the modular computation is not faithful to the non-modular case, *i.e.* the result obtained is not the modular reduction of the true answer. Typically bad reduction occurs when a test for equality in the modular image gives a different result from the corresponding test in the non-modular computation. The other problem is **reconstruction**, or how to deduce the non-modular result from the modular answer. Since modular reduction discards information, we can achieve reconstruction only by adding in information about the size or nature of the true result — this complementary information must be deduced from the original inputs.

## Motivation

A very basic algebra system might offer only the four fundamental arithmetic operations on polynomials in $\mathbf{Q}[x]$. Before long we would want to extend its repertoire, most particularly to compute GCDs, which are essential if we want to use rational functions. Luckily, Euclid's algorithm adapts readily to $\mathbf{Q}[x]$. Not so luckily, it turns out to be hideously slow.

It is curious how one of the oldest known algorithms became the focus of intense research activity more than 2000 years after its discovery. The villain was "intermediate expression swell": although inputs and output are of modest size, the algorithm generates intermediate results with enormous coefficients. In contrast if we apply the same algorithm to polynomials in $\mathbf{F}_p[x]$, we can be certain that no intermediate result has enormous coefficients: indeed no coefficient can ever exceed $p-1$. In fact, by choosing a suitable $p$ we can indeed calculate the GCD of two polynomials in $\mathbf{Q}[x]$ avoiding the dreaded villain.

Another case where modular methods arise is in polynomial factorization. The simplest case is factorization in $\mathbf{Z}[x]$. Theoretically the problem had been solved several centuries ago, but the method is feasible only for quite small examples — the method is modular and has been ascribed to Kronecker or Newton. The crucial modern breakthrough was a fast algorithm for factorizing in $\mathbf{F}_p[x]$ (due to Berlekamp). Now the only known factorization algorithms feasible for non-trivial cases work via factorization in $\mathbf{F}_p[x]$.

# Summary of examples and non-examples

Later on we shall investigate two particular cases where modular methods work especially well. Here I mention a variety of applications enjoyed by modular methods.

## Some examples

The two uses of modular methods we shall look at more closely are in the computation of GCDs in $\mathbf{Z}[x]$ and for factorization in $\mathbf{Z}[x]$. I have chosen these two particular examples partly because I am familiar with them, and partly because of their historical importance in stimulating the development of modular methods during the early years of computer algebra.

Calculating the determinant of an integer matrix benefits greatly from modular methods. We use Hadamard's bound to limit the size of the result, though it can be quite loose in some cases (implying a certain waste of computation). The waste can be almost entirely eliminated by using a probabilistic termination criterion.

Modular methods work well for solving linear systems with integer or rational entries. Some care is required because the solutions are generally rational numbers — rational numbers can be reconstructed from their modular images, as we shall soon see.

A more complex example is the computation of GCD/factorization in $\mathbf{Z}[x_1, \ldots, x_n]$. We first apply a modular reduction from multivariate to univariate, then use modular methods to calculate the univariate result, and finally reconstruct the multivariate answer from the univariate one.

## Some non-examples

You might be wondering whether modular methods can be applied universally. Unfortunately not. They cannot be applied to algorithms which depend on the arithmetic ordering of integers, for instance Euclid's algorithm on integers. Euclid's algorithm makes use of the ordering when computing the (least positive) remainder. All ordering information is lost when we reduce modulo $p$.

A similar example is the computation of Hermite normal form of a matrix of integers — internally the algorithm computes GCDs.

Currently no one knows of a completely satisfactory way to use modular methods for computing Gröbner bases. The Gröbner trace algorithm is probabilistic, but it is costly to verify the result and difficult to estimate the probability of it being wrong; so its usefulness is limited. Arnold has come up with some promising results regarding the use of modular methods provided the Hilbert function is known.

There are some cases where modular methods bring no benefit. For instance, addition of polynomials in $\mathbf{Z}[x]$ is so simple that the overheads of reduction and reconstruction render modular methods inappropriate. In general, it is often unclear whether modular methods would yield better performance when applied to algorithms whose final result is bigger than any intermediate result: for instance, computing the composition of two polynomials in $\mathbf{Z}[x]$.

# Reconstruction techniques

An obvious limiting factor on the use of modular methods is the existence of an appropriate reconstruction algorithm. Here I present several such algorithms, ranging from very simple to rather involved.

### Reconstruction from $\mathbf{Z}/m$ to $\mathbf{Z}$

The complementary information we need in this case is a range of width at most $m$ containing the true answer. Typically we know that the value to be reconstructed $n \in \mathbf{Z}$ satisfies $-m/2 < n < m/2$. Reconstruction then entails simply taking the residue class representative of least absolute value; the result is obviously unique.

Normally we use theoretical results to determine a bound $B$ for the absolute value of the true answer $n$. We then conduct modular computations sufficient to achieve a "precision" of at least $B$, at which point we are certain that reconstruction will work correctly. For good efficiency we need a bound $B$ which is usually reasonably tight.

### Reconstruction from $\mathbf{Z}/m$ to $\mathbf{Q}$

The problem is to reconstruct $p/q \in \mathbf{Q}$ given its image $r \in \mathbf{Z}/m$. As in the case of reconstructing an integer we also need information about how big $p$ and $q$ could be. If we have available a value $d$ which is a small multiple of $q$ then we can simply apply integer reconstruction to $dr$ to obtain the numerator $n$; finally we cancel any common factors between $n$ and $d$.

If we do not know any small multiple of $q$ then it is sufficient to know upper bounds $q_{max}$ for $q$ and $p_{max}$ for $p$. For the algorithm to work we must have $m > 2p_{max}q_{max}$; this condition guarantees that the reconstructed rational is unique (if it exists at all). In this case we can use the algorithm published by Wang, Guy, and Davenport (the WGD Algorithm).

The algorithm derives from the following observation. Suppose $p/q \equiv r \bmod m$. Equivalently, there is an integer $k$ such that $p = rq - km$ (without modulus). Dividing this by $mq$ we find that

$$\left|\frac{r}{m} - \frac{k}{q}\right| = \left|\frac{p}{mq}\right| < \frac{1}{2q_{max}q} \leq \frac{1}{2q^2}$$

In other words, $k/q$ approximates $r/m$ exceptionally well given the size of its denominator. Such exceptionally good approximations will always appear as a continued fraction convergent for $r/m$ — this is not an obvious result.

The algorithm is very simple. Take the last continued fraction convergent for $r/m$ with denominator not exceeding $q_{max}$; call its denominator $q$. Compute $p = rq \bmod m$. If $|p| \leq p_{max}$ return the answer $p/q$, otherwise return *non-existant.*

Recall that the continued fraction convergents can be computed easily using a form of Euclid's algorithm. A slight improvement to the WGD algorithm was found by Encarnacion.

### Reconstruction of algebraic numbers from modular images

Let $\mathbf{Q}(\alpha)$ be an algebraic number field, and let $m_\alpha$ be the minimal polynomial of $\alpha$ over $\mathbf{Q}$. Let $p$ be a prime which does not divide any denominator in $m_\alpha$. Suppose that $m_\alpha$ is squarefree modulo $p$, and let $m_\beta$ be a factor of $m_\alpha$ modulo $p$. Then there is an algorithm to reconstruct elements of $\mathbf{Q}(\alpha)$ from their images modulo $(p^k, m_\beta^*)$ where $m_\beta^*$ is the $p$-adic factor corresponding to $m_\beta$. The algorithm was published by Lenstra, uses LLL lattice reduction, and is too complicated to describe here.

### Reconstruction of univariate polynomials from their values

The classical method for reconstructing a univariate polynomial from its values $(y_1, \ldots, y_n)$ at distinct points $(x_1, \ldots, x_n)$ may be summarized in a single formula:

$$\sum_{i=1}^{n} y_i \frac{\prod_{j \neq i}(x - x_j)}{\prod_{j \neq i}(x_i - x_j)}$$

There is one particular case worthy of special mention: if $n$ is a power of two and the $x_i$ are all the $n$-th roots of unity then the reconstruction can be effected by an (inverse) fast fourier transform

There is an even simpler method in the case of $f \in \mathbf{Z}[x]$ where we know good bounds on the coefficients of $f$: we can simply read off the polynomial from its value at a single large point. For example, let $f$ be a polynomial with all coefficients bounded in absolute value by 49; then given $f(100) = 2999907$ we can deduce immediately that $f = 3x^3 - x + 7$, and the result is obviously unique. This approach works best if all coefficients have more or less the same size, and we have a good bound for their size.

**Ben-Or and Tiwari's method for sparse multivariate polynomials**

Ben-Or and Tiwari's method reconstructs an element $f \in R[x_1, \ldots, x_n]$ from its evaluations in $R$. It is particularly well suited to the reconstruction of very sparse polynomials: it presupposes only knowledge of an upper bound on the number of terms in the polynomial. In particular, it does not require any knowledge of the degree of the polynomial nor the size of its coefficients. Frankly, I was amazed when I first heard of the existence of this algorithm.

In the presentation of the algorithm I shall assume $R = \mathbf{Z}$; we shall see why later. Let $t$ be the number of terms in $f$; so we may assume that $f = \sum_{i=1}^{t} c_i \pi_i$ where the $c_i \in R \setminus \{0\}$ and the $\pi_i$ are distinct power products. Let $p_1, \ldots, p_n$ be the first $n$ prime numbers, and perform the following $2t+1$ evaluations: $v_k = f(p_1^k, \ldots, p_n^k)$ for $k = 0, 1, \ldots, 2t$. These are the only evaluations of $f$ we shall need.

Our careful choice of evaluation points allows us to write the $v_k$ in an alternative manner: $v_k = c_1 m_1^k + \cdots + c_t m_t^k$ where each $m_j$ is the value of $\pi_j$ at the point $(p_1, \ldots, p_n)$; observe that the $m_j$ are distinct. This particular form indicates that the sequence of the $v_k$ satisfies a recurrence of order $t$. That is, there exist integers $\alpha_1, \ldots, \alpha_r$ for which:

$$v_k = \alpha_1 v_{k-1} + \alpha_2 v_{k-2} + \cdots + \alpha_t v_{k-t} \quad \text{for} \quad k \geq t$$

Moreover, the $\alpha_i$ are related to the $m_j$ in the following way:

$$x^t + \alpha_1 x^{t-1} + \alpha_2 x^{t-2} + \cdots + \alpha_t = \prod_j (x - m_j)$$

We can use polynomial factorization to obtain the $m_j$ once we know the $\alpha_i$. And the $\alpha_i$ we can find via linear algebra: they are the unique solution to this matrix equation

$$\begin{pmatrix} v_0 & v_1 & \cdots & v_t \\ v_1 & v_2 & \cdots & v_{t+1} \\ \vdots & \vdots & \ddots & \vdots \\ v_t & v_{t+1} & \cdots & v_{2t} \end{pmatrix} \begin{pmatrix} \alpha_t \\ \alpha_{t-1} \\ \vdots \\ \alpha_1 \\ -1 \end{pmatrix} = \underline{0}$$

Now we have the values of the $m_j$; the corresponding power product can be found simply by counting how many times each prime $p_i$ divides $m_j$. We find the value of the coefficients in $f$ by solving this linear system

$$\begin{pmatrix} m_1^0 & m_2^0 & \cdots & m_t^0 \\ m_1^1 & m_2^1 & \cdots & m_t^1 \\ \vdots & \vdots & \ddots & \vdots \\ m_1^{t-1} & m_2^{t-1} & \cdots & m_t^{t-1} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_t \end{pmatrix} = \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{t-1} \end{pmatrix}$$

Let me reiterate that we have reconstructed $f$ from its values at only $2t+1$ points regardless of the degree of $f$. One point which is not very evident from this presentation is that the values of the $v_k$, the $m_j$ and the $\alpha_i$ tend to be rather large.

**Zippel's method for sparse multivariate polynomials**

Zippel's method reconstructs $f \in R[x_1, \ldots, x_n]$ from various evaluations of $f$ in $R$. It has several merits including general applicability and good speed. In contrast, its main weak point is that it is probabilistic and could produce a wrong result; so it should be used only where the result can be verified in some way.

The prerequisites for using the algorithm are:

- a procedure for evaluating $f$ at a point of our choice in $R^n$
- a univariate reconstruction algorithm (reconstructs $g \in R[x]$ from its values in $R$)
- an upper bound for the degree of $f$ in each indeterminate
- a plentiful supply of elements of $R$ (problems can arise if $R$ is small and finite)

The probabilistic aspect derives from the fact that we make an initial choice of a point $(a_1, \ldots, a_n) \in R^n$ and we will assume thereafter that (for $s = 2, \ldots, n-1$) if $f(x_1, \ldots, x_s, a_{s+1}, \ldots, a_n) = \sum c_i \pi_i$ where $c_i \in R \setminus \{0\}$ and $\pi_i$ are distinct power products then $f = \sum C_i(x_{s+1}, \ldots, x_n)\pi_i$ for some polynomials $C_i$.

For simplicity we present the method for the case $n = 2$; the extension to the general case appears as one of the exercises.

Let $d_1$ be the bound for the degree of $f$ in $x_1$, and $d_2$ the bound for $x_2$.

Pick the starting point $(a_1, a_2) \in R^2$; in fact, we will never use $a_1$.

Pick distinct values $p_0, p_1, \ldots, p_{d_1} \in R$, evaluate $u_i = f(p_i, a_2)$ for each $i$. Apply univariate lifting to obtain $f(x_1, a_2) \in R[x_1]$.

Extract the support $S$ of $f(x_1, a_2)$. That is $S = \{\pi_1, \ldots, \pi_s\}$ where $f(x_1, a_2) = \sum_{i=1}^{s} c_i \pi_i$ with $c_i \in R \setminus \{0\}$ and $\pi_i$ distinct power products in the indeterminate $x_1$.

Here we make the implicit probabilistic assumption that $f(x_1, x_2) = \sum_{i=1}^{s} C_i(x_2)\pi_i$ and proceed to calculate values of all the $C_i$ simultaneously.

Pick $s$ distinct points $\alpha_1, \ldots, \alpha_s \in R^1$ for which the matrix with $(i, j)$ entry being the value of $\pi_i(\alpha_j)$ is non-singular. Here we need $R$ to be sufficiently large.

Pick distinct values $p_0, p_1, \ldots, p_{d_2} \in R$.

For each $j = 0, \ldots, d_2$ evaluate

$$\begin{aligned} f(\alpha_1, p_j) &= C_1(p_j)\pi_1(\alpha_1) + \cdots + C_s(p_j)\pi_s(\alpha_1) \\ \cdots & \quad \cdots \\ f(\alpha_s, p_j) &= C_1(p_j)\pi_1(\alpha_s) + \cdots + C_s(p_j)\pi_s(\alpha_s) \end{aligned}$$

and determine the values of $C_i(p_j)$ by solving a linear system.

For each $i = 1, \ldots, s$ apply univariate lifting to $C_i(p_0), \ldots, C_i(p_{d_2})$ to find $C_i(x_2)$.

The final lifted result is then $\sum_{i=1}^{s} C_i(x_2)\pi_i$.

## Case Study 1: GCD in $\mathbf{Z}[x]$

It is too easy to take GCD computation for granted. We all know Euclid's algorithm. Maybe you were mildly surprised when you first heard that it could be applied to polynomials too. Now, because you know in principle how to compute polynomial GCDs, you expect the computer to calculate GCDs *quickly.* Thanks to modular methods your expectations can be fulfilled.

Using modular methods to compute the GCD of $f, g \in \mathbf{Z}[x]$ is easy provided we are careful. There are three points to note:

1. we must avoid primes which divide LC($f$) or LC($g$);
2. $\mathbf{F}_p$ is a field, so a GCD in $\mathbf{F}_p[x]$ is defined only up to multiplication by a unit;
3. applying Euclid's algorithm in $\mathbf{F}_p[x]$ may produce an answer of degree strictly greater than that of the true GCD in $\mathbf{Z}[x]$.

It is not immediately clear how we can overcome these difficulties. We know that there are only finitely many bad primes, *i.e.* ones which yield a modular GCD of too high a degree. But it is not clear how to tell if a prime is bad. Even if the prime is good, the GCD computed is only an unknown constant multiple of the faithful modular image.

Since we cannot use modular methods to compute the GCD of integers, we have to determine the purely integer factor of the GCD in a separate way. So before applying the algorithm we precondition $f$ and $g$ by removing

their contents; the GCD of these contents will be the integer factor of the answer — here we are using Gauss's lemma implicitly. To find the polynomial part of the result we intend using Chinese remaindering: we shall compute the GCD modulo various primes $p_1, p_2, \ldots, p_k$, and combine them to obtain a GCD modulo $\prod p_i$.

### Chinese remaindering

Recall that the Chinese remainder theorem says that $\mathbf{Z}/m_1 \times \mathbf{Z}/m_2 \equiv \mathbf{Z}/(m_1 m_2)$ provided $m_1$ and $m_2$ are coprime; the theorem trivially generalizes to a product $\mathbf{Z}/m_1 \times \cdots \times \mathbf{Z}/m_k$. By **Chinese remaindering** we mean the computation of the element of $\mathbf{Z}/(m_1 \cdots m_k)$ corresponding to given elements of $\mathbf{Z}/m_1, \ldots, \mathbf{Z}/m_k$. The calculations for performing Chinese remaindering are quite simple. The simplest is a sequential algorithm; it is often a good choice. There are more advanced "parallel" algorithms which are faster if integer multiplication and GCD are fast. We can be clever if there are many values to be Chinese remaindered from the same set of quotient rings.

Chinese remaindering extends in a natural way to polynomials with integer coefficients, or to matrices with integer entries. Chinese remaindering lends itself naturally to highly parallel processing.

It is also possible to perform fault-tolerant Chinese remaindering: we reconstruct the correct answer even when some of the modular images are wrong. For this to work correctly the redundancy must exceed twice the maximum possible error.

We note that the classical way of interpolating a univariate polynomial is merely one way of effecting Chinese remaindering in a univariate polynomial ring.

## The algorithm

The algorithm accepts as input two polynomials $f, g \in \mathbf{Z}[x]$, ideally they should be primitive. The result is the primitive part of their GCD.

The algorithm chooses a succession of primes $p_1, p_2, \ldots$, avoiding any which divide $\mathrm{LC}(f)$ or $\mathrm{LC}(g)$. It applies Euclid's algorithm to $f$ and $g$ modulo each prime. It discards all primes where the GCD has non-minimal degree as they are surely bad; after trying enough primes, all surviving primes must be good. To deal with the unknown factors one technique is to impose a suitable leading coefficient $C$ on the GCD: we choose $C = \mathrm{GCD}(\mathrm{LC}(f), \mathrm{LC}(g))$ which must be a multiple of the true leading coefficient of $\mathrm{GCD}(f, g)$. The algorithm forces each modular gcd to have leading coefficient equivalent to $C$; so we know it is a faithful image of our chosen multiple of $\mathrm{GCD}(f, g)$, and can now apply Chinese remaindering directly.

Finally, the resulting polynomial is made primitive — forcing the leading coefficient may have introduced some artifical content.

The only remaining issue is deciding how many different primes to use so that reconstruction can succeed. This is dictated by the size of the largest coefficient which could possibly appear in the GCD. There are formulas for bounding the largest coefficient, though they often seem rather too generous. Note that there are cases where the GCD has coefficients larger than any coefficient in the input polynomials.

Rather than using loose bounds, we might prefer a probabilistic strategy: we keep adding primes until Chinese remaindering gives the same polynomial twice in succession. That polynomial is probably the right answer, but we cannot simply return it because it might be wrong. If the answer is right, it will divide both $f$ and $g$; if it is wrong, it cannot divide both $f$ and $g$ because either its degree is too high or it has a wrong coefficient — it cannot have degree less than that of the true GCD.

There are other modular methods for computing polynomial GCDs. Hensel lifting works well once a GCD free basis has been found. The polynomial GCD can also be found by computing a single (large) integer GCD (see publications about `HeuGCD` and `GCDHeu`).

# Case Study 2: Factorization in $\mathbf{Z}[x]$

Having seen how well Chinese remaindering works for calculating GCDs, we are tempted to try the same approach for factorization. Very quickly we discover that it is not so simple: good primes can be hard to find, and it is not always obvious which factors to combine using Chinese remaindering. These difficulties turn out to be essentially insurmountable, and a different approach is needed.

Bad primes are a fact of life in factorization: indeed for some polynomials, such as $x^4 + 1$, there are no good primes at all. Even worse: for the Swinnerton-Dyer polynomials all primes are extremely bad — the polynomials are irreducible but split into factors of degree 1 or 2 modulo every prime. Nonetheless, we can be certain of one fact: the modular factorization is a refinement of the true factorization.

A viable alternative to Chinese remaindering was found by Zassenhaus: **Hensel lifting**, an effective application of Hensel's Lemma. Instead of using many different primes, we use just one. Hensel lifting obtains a factorization modulo $p^k$ starting from one modulo $p$. We can think of it as an alternative to Chinese remaindering.

Analogously to the case of computing GCDs, the modular factors are defined only up to multiplication by a unit, and this ambiguity may be eliminated by imposing a leading coefficient.

If the prime we chose is good then reconstructing the true factors is a trivial matter. If not, we must find out how to combine the modular factors appropriately. One way is just to try all possible combinations, and check divisibility by each one: if the candidate divides the original polynomial then it is a true factor. A faster, more sophisticated technique was found recently by van Hoeij, and works via LLL lattice basis reduction.

The question of how much Hensel lifting to perform is answered by estimating how large the coefficients of factors could be. Several suitable bounds are known; they are all invariably quite loose for "normal" polynomials. Nonetheless, there do exist some rare polynomials in $\mathbf{Z}[x]$ all of whose irreducible factors have coefficients bigger than the biggest coefficient of the original polynomial.

Hensel lifting, when applicable, tends to be more efficient than Chinese remaindering, but appears to be inherently sequential.

## $p$-adic numbers and Hensel lifting

In contrast to Chinese remaindering, Hensel lifting uses only a single prime. Hensel lifting is not as widely applicable as Chinese remaindering, but it does work very well for polynomial factorizations, and for solving linear systems.

We are all familiar with the usual norm on the integers: the *absolute value.* It turns out that it is not the only possible norm: for each prime $p$ there is the associated $p$-adic norm, written $|\cdot|_p$. We can define it simply: $|n|_p = p^{-k}$ where $p^k$ is the highest power of $p$ dividing $n$; it is related to the concept of $p$-adic *valuation,* which we shall not need.

We use the $p$-adic norm to define the ring of $p$-adic integers $\mathbf{Z}_p$. Analogously to the construction of the reals $\mathbf{R}$ by completion of $\mathbf{Q}$ with respect to the usual norm, we build $\mathbf{Z}_p$ as the completion of $\mathbf{Z}$ with respect to $|\cdot|_p$. We can think of the elements of $\mathbf{Z}_p$ as being "power series" in $p$ with coefficients in $\mathbf{Z}/p\mathbf{Z}$: a typical element looks like

$$\alpha = \alpha_0 p^0 + \alpha_1 p^1 + \alpha_2 p^2 + \cdots \qquad \alpha_i \in \{0, 1, \ldots, p-1\}$$

One pretty result about $p$-adic numbers is Hensel's Lemma which shows how a solution modulo $p$ can be lifted to a $p$-adic solution. We illustrate it by showing Hensel lifting of a polynomial factorization. We start with a factorization $f \equiv gh \pmod{p}$ where $g$ and $h$ are coprime (modulo $p$); for convenience we suppose that $f, g, h$ are monic. Hensel lifting is a process which then determines a refined factorization $f \equiv g_k h_k \pmod{p^k}$ where $g_k \equiv g \pmod{p}$ and $h_k \equiv h \pmod{p}$, and $g_k$ and $h_k$ are monic.

The first step is to find the polynomials $g_2 = g + p\delta_g$ and $h_2 = h + p\delta_h$; substituting and expanding we get

$$f \equiv g_2 h_2 \equiv gh + p(\delta_g h + \delta_h g) \pmod{p^2}$$

Since the $p^0$ parts match in the factorization, we need only choose $\delta_g$ and $\delta_h$ to make the $p^1$ parts match. Now, by Bezout's Theorem, the coprimality of $g$ and $h$ implies the existence of polynomials $g^*$ and $h^*$ such that

$gg^* + hh^* \equiv 1 \pmod{p}$. We can compute $g^*$ and $h^*$ using the extended euclidean algorithm. Multiplying Bezout's equation by $(f-gh)/p$ we deduce the existence of polynomials $\delta_g$ and $\delta_h$ such that $\delta_g h + \delta_h g \equiv (f-gh)/p \pmod{p}$. In fact there are many possble choices for $\delta_g$ and $\delta_h$: we may add a multiple of $g$ to $\delta_g$ and subtract the same multiple of $h$ from $\delta_h$. To preserve monicness, we choose the unique $\delta_g$ satisfying $\deg(\delta_g) < \deg(g)$; this implies that also $\deg(\delta_h) < \deg(h)$. These conditions fix $\delta_g$ and $\delta_h$ uniquely, and we have found $g_2$ and $h_2$.

In a similar way we obtain $g_3$ and $h_3$ starting from $g_2$ and $h_2$; eventually we reach $g_k$ and $h_k$. This method is known as **linear lifting.** There is a more involved variant which goes straight from $g_2, h_2$ to $g_4, h_4$ to $g_8, h_8$, and so on: it is called **quadratic lifting,** and is usually faster than linear lifting to the same $p$-adic precision.