GGNFS Documentation

Chris Monico

Department of Mathematics and Statistics
Texas Tech University
e-mail: cmonico@math.ttu.edu

Draft of December 31, 2008

Contents

1 Disclaimer

1. Much of the info here on using `GGNFS` is already outdated! As the software gets closer to a near-final state I will come back and update it. In the meantime, the best idea for learning the usage would be to look at the test/ directories having the most recent timestamps.

2. This is a very preliminary version of this document - I will add the relevent references to the literature and try to fix typos, inaccuracies, and confusing discussions as I find them. It is really not ready to be distributed or read yet; you are only reading it now because, for my own convenience, I am storing it in the source tree. Heck - it's probably not even complete yet.

3. This document is not intended for people who really want to learn the details and many intricicies of the number field sieve. It is intended to give just enough insight to be able to either refer to the literature and learn more, or to be able to run the software and have at least a vague idea of what it's doing.

4. `GGNFS` is not claimed to be lightening fast, robust or even reliable (yet). Before you spend any exorbitant amount of time trying to factor some large number, start with smaller ones and work your way up to find/learn the limitations of the software. There are much better implementations out there. The only reason for writing this one is to have a complete and open source implementation available. It is hoped that, as it becomes more stable, people will begin to make contributions and improvements to the code. At this time, a fast pure (or almost pure) C, `GGNFS`-compatible lattice siever would be more than welcome, as well as improvements to the polynomial selection code. Both of these tasks require serious proficiency in mathematics and expertise in programming. If you are up to the challenge - go for it!

2 Preliminaries

Throughout this document, N will denote the number to be factored (and N is known to be composite). For simplicity, we will also assume N is a product of two distinct primes $N = pq$. For if this is not the case, then there is likely some other algorithm which can more efficiently factor N (for example, if N has 120 digits and 3 prime factors, then at least one of them has fewer than 40 digits, and so ECM would work). On the other hand, if $N = p^2$, then N is easily factored by simply computing the square root.

As is standard, we use the notation $a \equiv b \pmod{N}$ to denote the fact that $a + \mathbb{Z}/N\mathbb{Z}$ and $b + \mathbb{Z}/N\mathbb{Z}$ are the same equivalence class in $\mathbb{Z}/N\mathbb{Z}$ (i.e., $a - b$ is divisible by N). We also use the less standard notation $a \mod N$ to indicate the unique remainder r resulting from an application of the division algorithm. That is, it is the unique value of r so that for some $q \in \mathbb{Z}$,

$$a = qN + r, \qquad 0 \leq r < N.$$

Said differently, $a \mod N$ is the unique *nonnegative* remainder when a is divided by N. We will attempt to be consistent in our use (and abuse) of these two notations. For the reader not familiar with these notions, the difference is really subtle enough that it can be ignored.

The goal of many integer factorization algorithms is to produce two congruent squares. That is, to find nonzero $x, y \in \mathbb{Z}$ so that

$$x^2 \equiv y^2 \pmod{N}. \tag{2.1}$$

For we will then have

$$x^2 - y^2 \equiv (x-y)(x+y) \equiv 0 \pmod{N}.$$

From Equation 2.1, we see that x/y is a square root of 1 modulo N (if y is invertible modulo N; but if not, (y, N) will be a proper divisor of N). By our assumption that $N = pq$, there are exactly 4 distinct square roots of unity, and so if such an (x, y) pair is randomly chosen from the set

$$S = \{(x, y) \,|\, x^2 \equiv y^2 \pmod{N}\}$$

there is about a 50/50 chance that x/y is not one of 1,-1. This means that $x \not\equiv \pm y \pmod{N}$, and so the gcd $(x - y, N)$ is a proper divisor of N. The factorization methods based on this observation try to do exactly that - find a (pseudo) random element of the set S.

3 Fermat's and Dixon's methods

This section briefly describes Dixon's method, which is the predecessor of the Quadratic Sieve (QS). Since the NFS can be largely considered a horrific generalization of QS, it is crucial to understand the QS before moving on to NFS. Briefly, Dixon's method helps motivate the QS, which in turn helps motivate and understand the basic concepts of the NFS.

Suppose $N = 9017$. We could attempt to find two congruent squares as follows: First compute $\sqrt{N} \approx 94.96$. We can thus start computing $95^2 \mod N, 96^2 \mod N, \ldots$ until we find that one of these numbers is again a perfect square.

a	$a^2 \mod N$
95	8
96	199
97	392
98	587
99	$784 = 28^2$

The last row of this table gives us that $99^2 \equiv 28^2 \pmod{9017}$, and so $(99 - 28)(99 + 28) \equiv 0 \pmod{9017}$. We compute the gcd $(99 - 28, 9017) = (71, 9017) = 71$, and so we have found that 71 is a proper divisor of N. The cofactor is easily found by division and so we obtain $9017 = 71 \cdot 127$.

The method we just described is essentially Fermat's method for factoring integers. However, it is quite a bad thing to do in general. Observe: If $N = pq$ is odd, then we may assume $q = p + k$ for some positive even integer k. It's not hard to see that this method will require

about $k/2$ operations before it succeeds. If p and q each have say 30 digits, then k will typically still be a 30 digit number! That is, asymptotically, this is still a $O(\sqrt{N})$ algorithm in the worst case, and is no better than trial division.

This brings us to Dixon's method. If we had done a little extra work in the previous computation, we could have factored N a little earlier.

a	$a^2 \mod N$
95	$8 = (2^3)$
96	$199 = (199)$
97	$392 = (2^3)(7^2)$

We can already produce two congruent squares from the data in these first three rows. For notice that

$$(95 \cdot 97)^2 \equiv (95^2)(97^2) \equiv (2^3)(2^3 \cdot 7^2) \equiv (2^3 \cdot 7)^2 \pmod{N},$$

whence we obtain $198^2 \equiv 56^2 \pmod{N}$. The gcd computation $(198-56, 9017) = (142, 9017) = 71$ gives a proper divisor of N.

This is perhaps a misleading example: we saved only the computation of two squares modulo N. But asymptotically, it is actually much better than Fermat's method if done right. The general idea is to choose a *factor base* of small primes. We are then interested in only values of a so that $a^2 \mod N$ factors completely over this factor base. That is, we need not try too hard to factor the numbers in the right hand column - we just use trial division with the primes from the factor base, and throw out anything which does not factor completely in this way (i.e., we could have discarded the row corresponding to 96 since 96^2 did not completely factor over small primes). If the factor base has B primes, then it is easy to see that we need at most $B+1$ values of a so that $a^2 \mod N$ factors completely over this factor base. For once we have found at least $B+1$, we are guaranteed that some of them can be multiplied out to produce two congruent squares modulo N.

4 Quadratic sieve

Dixon's method represents a big improvement to Fermat's method. However, the trial division of $a^2 \mod N$ over the small primes is quite time consuming, since many of the $a^2 \mod N$ will not completely factor over the factor base. The net result is that we will waste alot of time doing trial division on numbers which we will only wind up throwing out anyway.

The point of the quadratic sieve is that we can substantially reduce the amount of work by employing a clever *sieve*. Loosely, a *sieve* to a mathematician is exactly the same thing it is to a cook. It is a device for efficiently separating the things we're interested in from the things we're not interested in. A common sieving device is some sort of wire mesh which will hold large objects and allow smaller ones to pass through. For example, a cook might use a sieve to remove chunks of skin and fat from a gravy. Or, perhaps to remove coarse clumps of flour to leave only the fine grains. Manufacturers of abrasives use a sieve to separate coarse grains of particles from fine grains (the finest grains become fine abrasives while the coarse ones left behind become coarse abrasives). The old school miners used to use a sieve to look

for gold within the mud of some creeks and rivers - the sieve would be used to remove the fine particles from the mud leaving only moderate sized rocks which could then be inspected for the presence of gold.

In the case of the QS, we will use a sieve to efficiently remove the "coarse" $a^2 \mod N$ values. Specifically, we will remove the ones which most likely do not factor over the small primes. The benefit is that we perform the more expensive trial division step only on numbers which are more likely to be useful (we waste much less time trying to factor numbers which aren't going to factor over the factor base anyway).

The key observation that makes this possible is as follows: If p is a small prime dividing $a_0^2 - N$ for some particular integer a_0, then

$$0 \equiv a_0^2 - N \equiv a_0^2 - N + 2p + p^2 \equiv (a_0 + p)^2 - N \pmod{p}.$$

That is, if $a_0^2 - N$ is divisible by p, then so is $(a_0 + p)^2 - N$.

The astute reader might have noticed that we went from talking about $a^2 \mod N$ to $a^2 - N$. Why? Recall that in Dixon's method we started with an initial value of $a = a_0 = \lceil \sqrt{N} \rceil$. For this value of a and at least several after it, $a^2 \mod N = a^2 - N$. In fact, this continues to hold for values of a until $a_1^2 \geq 2N$. But this means $a_1 \geq \sqrt{2N}$. Since our goal is to have an algorithm which is much better than trial division, we have no intention of performing anywhere near $\sqrt{N}$ operations, so *for all values of a in which we're interested, we can assume* $a^2 \mod N = a^2 - N$. This is also convenient since subtraction is an easier operation than division-and-take-remainder anyway.

Definition 4.1 Let B be a positive integer. An integer n is said to be *B-smooth* if all prime divisors of n are less than or equal to B.

So here is the essence of the quadratic sieve:

1. Choose a positive integer B and a factor base of small primes upto B. Let s denote number of primes in the factor base.

2. Set $a_0 \leftarrow \lceil \sqrt{N} \rceil$. Set $a_1 \ll \sqrt{2N}$ to be a large enough integer that there are at least, say $s + 5$, B-smooth values of $a^2 - N$ with $a_0 \leq a \leq a_1$.

3. Initialize an array of size $a_1 - a_0 + 1$.

4. For each prime p_j in the factor base, do as follows:

   - Find the smallest value of $k \geq 0$ so that $(a_0 + k)^2 - N \equiv 0 \pmod{p_j}$ (if one exists; see the note below).
   - While $a_0 + k \leq a_1$, note in the array entry corresponding to $a_0 + k$ that $(a_0 + k)^2 - N$ is divisible by p_j, then do $k \leftarrow k + p_j$.

5. Scan through the array for entries which are divisible by many primes in the factor base. Do trial division on the $a^2 - N$ corresponding to each such entry, keeping only the ones which are B-smooth.

6. If there were fewer than $s+5$ factorizations kept in the previous step, a_1 was too small - repeat the same procedure over the interval $[a_1+1, a_1+m]$ for a sufficiently large m to produce more.

7. Find a subset of the factorizations that can be multiplied out to produce two congruent squares, $x^2 \equiv y^2 \pmod{N}$. If $(x-y, N) = 1$, try a different subset of the factorizations. (It is easily seen that if we have at least $s+5$ good factorizations over the s primes, then there are at least 16 different ways to produce the congruent squares. One of them will do the job with high probability).

The important difference between QS and Dixon's method comes in Step 4. This is the *sieving step*. The point is that we must do a little work to find some value of k so that $(a_0+k)^2 - N \equiv 0 \pmod{p_j}$. But having done that, we get (almost) for free, all values of a in the interval $[a_0, a_1]$ which are divisible by p_j.

There are several important points we've neglected to keep the above description simple: First of all, the reader may have noticed that the equation $x^2 - N \equiv 0 \pmod{p_j}$ in Step 4 actually has *two* distinct solutions modulo p_j in general. This is true, and in fact, we should use them both! Also, it is not hard to see that there are certain primes which can be omitted from the factor base - those for which $\left(\frac{N}{p_j}\right) = -1$ can be trivially omitted, since there will be no solution for k in Step 4. Also, we haven't mentioned anything about how to choose B; on the one hand, if it's too small, it will be very hard to find B-smooth values of $a^2 - N$. On the other hand, if it's too large, we will need to find way too many of them! Also, we need not construct a really enormous array and try to find them all at once - in practice, one partitions the very large array into pieces small enough to fit into the CPU cache, and does each piece separately. We also did not specify exactly how to "note in the array that $(a_0+k)^2 - N$ is divisible by p_j. This can be done, for example, by initializing the array entries to zero and simply adding $\log p_j$ to the corresponding entry. Then, at the end of the day, if an entry of the array is x, we will know that it has a divisor of size at least e^x which is B-smooth.

There are many significant improvements to the algorithm that we won't cover here. Indeed, the above description is only enough to implement a very rudimentary version of QS which could barely handle 40 digit numbers. With the improvements (multiple-polynomials, large prime variations, Peter Montgomery's block Lanczos algorithm,...) it is possible to factor numbers of upto say 130+ digits with quadratic sieving.

The most important point is that the overview of the algorithm is very similar to the overall view of the NFS. It can be viewed as having four major steps:

1. Parameter selection.

2. Sieving (produce relations).

3. Find dependencies among relations.

4. Produce final factorization.

The NFS will proceed in essentially the same way - only the specifics for each step will change.

5 About the number field sieve

The number field sieve in its current form is a very complicated beast. John Pollard had the interesting idea to (re-)factor the seventh Fermat number $F_7 = 2^{2^7} + 1$ by doing some very clever computations in the number field $\mathbb{Q}(\alpha) = \mathbb{Q}[x]/\langle x^3 + 2\rangle$. It was clear immediately that his idea could be used to factor any number of the form $r^e \pm s$ for small e and s. Subsequently, many people contributed to making the basic algorithm work for integers without such a special form.

The number field sieve is still more efficient for numbers of the special form that Pollard first considered. However, such numbers can be factored with the same algorithm - only the first step becomes much easier and the algorithm runs with much 'better parameters' than for more general numbers. However, factoring such numbers is still referred to as 'Special Number Field Sieving (SNFS)' to highlight the fact that the factorization should proceed much more quickly than for arbitrary numbers of the same size. From an aesthetic point of view, it is very pleasing that the NFS should factor special-form numbers faster than more general numbers. After all, from a complexity-theoretic point of view, these numbers can be passed to an algorithm with a smaller input size. That is to say, in some sense, such numbers have lower (Kolmogorov) complexity since they admit a shorter description, and so it is aesthetically pleasing (to me, at least) that the NFS can take advantage of this.

Similarly, the number field sieve applied to numbers which have no such special form is sometimes referred to as the General Number Field Sieve (GNFS) to highlight the fact that it is being applied to a general number. But again - the algorithm proceeds in exactly the same way after the first step.

The quadratic sieve was a huge leap forward in factoring integers. However, it does have something of a bottleneck - the values $a^2 - N$ which are required to be B-smooth grow quadratically as a increases. They very quickly become so large that they are less and less likely to be B-smooth. This problem is somewhat alleviated by the multiple polynomial version of the quadratic sieve (MPQS), but it is still the major hindrance to factoring larger numbers.

The number field sieve gains its (asymptotic) efficiency over QS by moving it's search for smooth numbers to a set of numbers which tend to be much smaller. This does come at a price - the numbers in question must actually be simultaneously smooth over two different factor bases: a *rational factor base (RFB)* and an *algebraic factor base (AFB)*. Even still, it turns out that the gain is significant enough to make NFS asymptotically faster (already faster by 120 digits or so - perhaps lower or higher, depending on implementations).

It is not our intent to give here a complete and thorough description of the algorithm. For that, the reader is referred to the many papers on the subject. Instead, we wish to give just an essential overview of the major steps and a basic insight as to how each is performed.

6 NFS overview

This section is for the mathematically sophisticated, and may be skipped if desired. The next section will still be understandable for the purposes of running the `GGNFS` code - all that will be lost by skipping this section is any hint as to how it actually works and what

it really does. The number field sieve is, however, a rather deep algorithm and difficult to describe precisely. This section is intended to be just a broad overview; perhaps I will add another section to this document in the future containing a more precise description. The problem is that there are so many details to be filled in that a very precise description could easily fill ten pages and look more like computer code than mathematics.

As usual, suppose N is the composite number to be factored. Let $c_0 + c_1 x + \cdots c_d x^d = f(x) \in \mathbb{Z}[x]$ be a (not necessarily monic) polynomial of degree d so that $f(m) = cN$ for some small nonzero integer c and some $m \in \mathbb{Z}$. We assume f to be irreducible - for in the unlikely scenario where $f = gh$ splits, then we would have $g(m)h(m) \equiv 0 \pmod{N}$ likely giving a factorization of N.

Set $\mathbb{K} = \mathbb{Q}(\alpha) = \mathbb{Q}[x]/\langle f \rangle$, where $f(\alpha) = 0$. If we let $\hat{\alpha} = c_d \alpha$ there is a ring homomorphism from the order $\mathbb{Z}[\hat{\alpha}]$ to $\mathbb{Z}/N\mathbb{Z}$ given by

$$\begin{aligned} \psi : \mathbb{Z}[\hat{\alpha}] &\longrightarrow \mathbb{Z}/N\mathbb{Z} && (6.2) \\ h(\hat{\alpha}) &\longmapsto h(m) && (6.3) \end{aligned}$$

In fact, either this homomorphism extends to a homomorphism from the full ring of integers $\mathbb{Z}_{\mathbb{K}}$, or we can factor N by simply computing $\mathbb{Z}_{\mathbb{K}}$. Usually the former case is true (in fact, `GGNFS` does not even check for the latter case as it is extremely unlikely for numbers N with even 40 digits). The reason for taking $\hat{\alpha}$ instead of α is quite simply that, in general, α will not be an algebraic integer, and later computations are most easily done with respect to an integral basis - so we can avoid fractional arithmetic by considering $\hat{\alpha}$ instead of α. But, of course, $\mathbb{Q}(\alpha) = \mathbb{Q}(\hat{\alpha})$, and so it doesn't matter which we use in principle.

Choose bounds $B_1, B_2 > 0$ and let $\mathcal{R}$ denote the set of prime integers below B_1. $\mathcal{R}$ is called the *Rational Factor Base (RFB)*. Recall that an ideal in the ring of integers $\mathfrak{p} \subseteq \mathbb{Z}_{\mathbb{K}}$ is called a *first degree prime ideal* if $\mathfrak{p}$ is a prime ideal with norm $N(\mathfrak{p}) = |\mathbb{Z}_{\mathbb{K}}/\mathfrak{p}| = p$ for some prime integer p. The point is that an arbitrary prime ideal may have norm p^e for any $1 \leq e \leq d$, but the first degree prime ideals have norm p^1. The *Algebraic Factor Base (AFB)* $\mathcal{A}$ will consist of all first degree prime ideals in $\mathbb{Z}_{\mathbb{K}}$ with norm at most B_2. It is easy to see that, with at most finitely many exceptions (in fact, a very small number of exceptions), these ideals may be represented uniquely as the set of pairs of integers (p, r) with $p \leq B_2$ a prime and $f(r) \equiv 0 \pmod{p}$ (to see this, consider the canonical projection from $\mathbb{Z}_{\mathbb{K}}$ to $\mathbb{Z}_{\mathbb{K}}/\mathfrak{p} \cong \mathbb{Z}/p\mathbb{Z}$). Explicitly, we represent these internally as

$$\begin{aligned} \mathcal{R} &= \{(p, r) \in \mathbb{Z} \,|\, 2 \leq p \leq B_1 \text{ is prime}, r = m \mod p\} && (6.4) \\ \mathcal{A} &= \{(p, r) \in \mathbb{Z} \,|\, 2 \leq p \leq B_1 \text{ is prime}, f(r) \equiv 0 \pmod{p}\} && (6.5) \end{aligned}$$

The reason for keeping track of $m \mod p$ for the primes in the RFB will become clear in a moment; it will allow us to sieve identically with both factor bases (it also saves some computation during sieving).

The goal is to find a set of pairs $\{(a_1, b_1), \ldots, (a_s, b_s)\}$ such that

1. $\prod(a_i - b_i m) \in \mathbb{Z}$ is a perfect square, say x^2.
2. $\prod(a_i - b_i \alpha) \in \mathbb{Z}_{\mathbb{K}}$ is a perfect square, say β^2.

For if we can find such (a, b) pairs satisfying this, we then have

$$x^2 = \prod(a_i - b_i m) \equiv \prod \psi(a_i - b_i \alpha) \equiv \psi(\prod(a_i - b_i \alpha)) \equiv \psi(\beta)^2 \pmod{N},$$

the congruent squares we seek.

The way we will find such (a, b) pairs is similar to the way we did it in the quadratic sieve: we will find many such pairs that are smooth over the factor bases and then find a subset which gives the necessary perfect square products. On the rational side, this is a straightforward operation, facilitated by the fact that the size of the numbers involved is much smaller than for the QS (about $O(N^{1/(d+1)})$.

The situation on the algebraic side is somewhat different, complicated by the fact that neither $\mathbb{Z}_{\mathbb{K}}$ nor $\mathbb{Z}[\alpha]$ are UFDs in general. Without unique factorization, the entire factor base idea seems to fall apart. On the other hand, $\mathbb{Z}_{\mathbb{K}}$ is a Dedekind domain and so we have unique factorization of ideals! This is the fact that will save us. To be now a little more precise than above, we will find a set of pairs $\{(a_1, b_1), \ldots, (a_s, b_s)\}$ such that

1. $\prod(a_i - b_i m) \in \mathbb{Z}$ is a perfect square, say x^2.
2. $\prod \langle a_i - b_i \alpha \rangle \subset \mathbb{Z}_{\mathbb{K}}$ is a perfect square ideal, say I^2.
3. The product $\prod(a_i - b_i \alpha)$ is a quadratic residue in all of the fields $\mathbb{Z}[x]/\langle q, f(x) \rangle$ for a fixed set of large primes q not occurring anywhere else, and for which $f(x)$ is still irreducible mod q. This base of primes is called the Quadratic Character Base (QCB). It consists of a relatively small number of primes (60 in `GGNFS`).

The idea is that conditions 2 and 3 together make it extremely likely that, in fact, $\prod(a_i - b_i \alpha) \in \mathbb{Z}_{\mathbb{K}}$ is a perfect square. Loosely, it can be thought of this way: if the product is a perfect square, it will necessarily be a quadratic residue modulo in the specified fields. If it is not a perfect square, it will be a quadratic residue with about a 50/50 chance. So, if it is a quadratic residue modulo sufficiently many primes, it becomes very likely that it is itself a perfect square.

Let us now describe the procedure by which we find such (a, b) pairs. We will again use a sieving procedure. There are several types of sieves, but we will discuss only the classical sieve here.

First observe that if $(p, r) \in \mathcal{R}$, then $p|(a - bm)$ iff $a \equiv bm \equiv br \pmod{p}$. This is one reason for storing the RFB in the way we did, having precomputed $r = m \mod p$. More importantly, the required congruence has exactly the same form as on the algebraic side: if $\mathfrak{p}$ is the first degree prime ideal corresponding to the pair $(p, r) \in \mathcal{A}$, then $\mathfrak{p}|\langle a - b\alpha \rangle$ iff $a \equiv br \pmod{p}$ (this follows by considering the image of $\langle a - b\alpha \rangle$ under the canonical projection $\pi_{\mathfrak{p}} : \mathbb{Z}_{\mathbb{K}} \longrightarrow \mathbb{Z}_{\mathbb{K}}/\mathfrak{p}$).

Finally we remark that we need consider only (a, b) pairs such that a and b are coprime. The reason for this will be clear when we discuss finding dependencies shortly. Briefly, if we have two (a, b) pairs with one a multiple of the other, and both appearing in the subset which generates the final squares, then we could remove both and still have a square. That is, we can do the job of both of them using at most one of them - so for simplicity, we can simply ignore such multiples.

The facts above suggest the classical sieve:

1. Fix an integer $b > 0$, and a range of a values $[a_0, a_1]$ (typically chosen symmetrically, $[-A, A]$). We will produce values of $a_0 \leq a \leq a_1$ so that $(a - bm)$ is smooth over the RFB and $\langle a - b\alpha \rangle$ is smooth over the AFB.

2. Initialize an array of size $a_1 - a_0 + 1$.

3. (rational sieve) For each $(p, r) \in \mathcal{R}$ do as follows:

   - Find the least nonnegative value of k so that $a_0 + k \equiv rb \pmod{p}$.
   - While $a_0 + k \leq a_1$ do as follows: note in the k-th entry of the array that $(a - bm)$ is divisible by (the prime corresponding to) the pair (p, r), then $k \leftarrow k + p$.

4. Scan through the array and indicate elements which are not divisible by enough primes from $\mathcal{R}$ to be smooth. For the elements which are (or could be) smooth over $\mathcal{R}$, initialize the corresponding entries for the algebraic sieve.

5. (algebraic sieve) For each $(p, r) \in \mathcal{A}$ do as follows:

   - Find the least nonnegative value of k so that $a_0 + k \equiv rb \pmod{p}$.
   - While $a_0 + k \leq a_1$ do as follows: note in the k-th entry of the array that $(a - bm)$ is divisible by (the prime corresponding to) the pair (p, r), then $k \leftarrow k + p$.

6. Output any pairs (a, b) with a and b coprime and which were divisible by sufficiently many primes from both factor bases to be (probably) smooth over both FB's.

To determine whether or not entries are "divisible by enough primes to be smooth", we do as we did for the QS. Initialize the entries to zero. Then when an entry is divisible by some prime (p, r), we simply add $\log p$ to the entry. After one such sieve, then, if an entry contains the value x, we know that it has at least e^x prime divisors from the corresponding factor base. It is clear that such a procedure will necessarily miss some values of (a, b) which are smooth (i.e., perhaps because it is divisible by a moderate power of one of the primes). For this reason, we allow for 'fudge factors' in both sieves. We may thus wind up missing some smooth (a, b) pairs and outputting some which are not smooth; so long as we generate many pairs which are smooth and not too many 'false reports', the situation is good.

Remark 6.1 The code has been modified, as of version 0.33, so that the siever now does the bulk of the factoring of relations. Thus, it outputs only good relations and the number it reports as `total: xxxxx` is an accurate total number of good relations found.

Okay - assume we have generated many coprime pairs (a, b) which are simultaneously smooth over $\mathcal{R}$ and $\mathcal{A}$. Our goal is to find a subset so that

1. $\prod(a_i - b_i m) \in \mathbb{Z}$ is a perfect square, say x^2.

2. $\prod \langle a_i - b_i \alpha \rangle \subset \mathbb{Z}_{\mathbb{K}}$ is a perfect square ideal, say I^2.

3. The product $\prod(a_i - b_i\alpha)$ is a quadratic residue in all of the fields $\mathbb{Z}[x]/\langle q, f(x)\rangle$ for a fixed set of large primes q not occurring anywhere else, and for which $f(x)$ is still irreducible mod q. This base of primes is called the Quadratic Character Base (QCB). It consists of a relatively small number of primes (60 in `GGNFS`).

We accomplish this with $\mathbb{F}_2$-linear algebra. Construct a matrix over $\mathbb{F}_2$ with one column for each (a, b) pair and one row for each element of $\mathcal{R}, \mathcal{A}$, the QCB, and one additional row. Record in each entry, as appropriate, one of the following:

- The parity of the exponent of $(p, r) \in \mathcal{R}$ dividing $(a - bm)$.
- The parity of the exponent of $(p, r) \in \mathcal{A}$ dividing $\langle a - b\alpha\rangle$.
- A 1 if the (a, b) pair corresponds to a quadratic nonresidue, 0 if it corresponds to a quadratic residue.
- A 1 if $a - bm < 0$ (for moderate sized numbers, $a - bm$ will always be negative).

Then find some vectors in the kernel of the corresponding matrix. Hence the name of the step: the *matrix step* or *dependency step*. Such vectors correspond to a subset of the (a, b) pairs which satisfy the necessary conditions to produce squares. (The last row is needed to insure that the final product is positive; without it, we could wind up with something like $-(5^{12})(7^6)(11^6)(13^4)$ for example, which has all prime factors with even exponents and yet is not a perfect square).

So far, we haven't talked about particular numbers. The factor bases for moderate sized numbers may easily have 100,000 elements each. In such a case the matrix involved may be about $200,000 \times 201,000$ in which case Gaussian elimination becomes quite difficult. The situation is even worse for larger numbers where the factor bases may have over a million elements. Luckily, these matrices are sparse and Peter Montgomery gave a terrific block Lanczos algorithm which works over $\mathbb{F}_2$.

Finally there is one last problem which is not self-evident until a close inspection. Once we have such a dependency in hand we have a set $S = \{(a_1, b_1), \ldots, (a_s, b_s)\}$ so that the products $\prod_{(a,b)\in S}(a - bm) \in \mathbb{Z}$ and $\prod_{(a,b)\in S}(a - b\alpha) \in \mathbb{Z}_{\mathbb{K}}$ are both (extremely likely to be) squares. Let's say the first is $x^2 \in \mathbb{Z}$ and the latter is $\beta^2 \in \mathbb{Z}_{\mathbb{K}}$. Both x and β will be extremely large: if the factor bases have say 100,000 entries each, then we can expect that the subset S will contain roughly half of all the pairs. Since we have more than 200,000 (a, b) pairs in total, we expect that S contains about 100,000 (a, b) pairs. If each value of a and b has a size of only 5 digits each, the resulting x can be expected to have about 500,000 digits. The situation is even worse for β since, considered as an equivalence class in $\mathbb{Q}[x]/\langle f\rangle$, the degree of β remains bounded by $\deg(f)$ but some amount of coefficient explosion can be expected.

At the end of the day, it suffices to know $x \pm \psi(\beta) \mod N$ to compute the resulting divisor of N. For this, it suffices to find $x \mod N$ and $\psi(\beta)N$. We can certainly find $x \mod N$ quite easily since we know it's factorization over $\mathbb{Z}$ - just multiply out the factors modulo N. The situation for computing $\psi(\beta)$, however, is decidedly more difficult. Recall that we do *not* actually know a prime factorization of β (indeed, β does not even have

a unique prime factorization in general!). Rather, we know a prime factorization of the ideal $\langle\beta\rangle \subset \mathbb{Z}_{\mathbb{K}}$. Furthermore, these ideals are not even principal, so there is no real hope of solving this problem in any straightforward way. This was one of the major stumbling blocks early on to generalizing the number field sieve to the general case. There were several early solutions to this problem which, while they did work in feasible time for modestly-sized numbers, they actually dominated the asymptotic runtime of the NFS. Luckily, Peter Montgomery solved this problem and gave a very fast way to compute the square root by successive approximations. The idea is to choose some element $\delta \in \mathbb{Z}_{\mathbb{K}}$ divisible by many of the primes in the factorization of $\langle\beta\rangle$. Then, do $\langle\beta\rangle \leftarrow \langle\beta\rangle/\langle\delta\rangle$ (working with fractional ideals, of course). Repeating this process, we shrink $N(\langle\beta\rangle)$ down to something manageable. Meanwhile, we have precomputed $\prod_{(a,b)\in S}(a - b\alpha)$ modulo some moderate sized primes, and we perform the same operations on the residue classes. If we are very careful, we wind up with a final $\langle\beta\rangle$ of very small norm and we can use CRT to lift the final result from the residue classes. It is not quite this easy - there are many subtleties along the way; for example, it would be quite possible to wind up with a final ideal of $\langle 1\rangle$, and yet the resulting element to be lifted still has extremely large coefficients. These problems are all solved very satisfactorily. The overall algorithm relies heavily on the LLL algorithm for computing reduced lattice basis vectors, and so it is quite efficient in practice.

7 Overview of `GGNFS`

To factor a number with `GGNFS`, there are five major steps:

1. Select a polynomial. This is done with either the program `polyselect` or by hand (for special form numbers, like $2^{2^9} + 1$).

2. Build the factor bases. Obviously this is done with the program `makefb`.

3. Sieve. Classical sieving (slow) is done with the program `sieve`. Lattice sieving (faster) is done with `gnfs-lasieve4I12e` or similar (these are Franke's programs, included in `GGNFS` for simplicity).

4. Process relations. This is done with the program `procrels`. It takes siever output, filters out duplicates and unusable relations. If there might be enough relations, it will attempt to combine partial relations into full relation-sets to build the final matrix. If this happens, the program will tell you that you can proceed to the matrix step. If not, go back and do more sieving.

5. Find dependencies (aka, the matrix step). This is done with the program `matsolve`. It will take the matrix spit out in the previous step and find vectors in the kernel. First, though, the program will attempt to shrink the matrix a bit by removing unnecessary columns and rows, and doing some clever column operations to further shrink the matrix. The more overdetermined the matrix is, the more luck the pruning step will have with shrinking the matrix. Thus, it is sometimes preferable to do a little more sieving than really needed to get a larger matrix than really needed, and let matsolve spend some time shrinking it.

6. Compute the final factorization. This is done by the program `sqrt`.

Each of these programs depends on the output of the previous program. The `sqrt` program can only be run when the `matsolve` program reports that it did solve the matrix and find dependencies. In turn, the `matsolve` program can only be run when the `procrels` program tells you to do so.

Each step requires separate explanation.

If you have no idea at all what these steps mean, fear not - there is a Perl script included which will take care of everything for general numbers. For special form numbers, you will have to choose a polynomial by hand. (As of version 0.70.2, the script can do polynomial selection for numbers of certain sizes - the default parameter file still needs to be filled more, though, before this will work well).

8 Polynomial selection (special numbers)

We need to construct a polynomial $f(x)$ with integer coefficients so that $f(m) = cN$ for some integer m and some small nonzero integer c. The degree of f should usually be 4 or 5, and we should try to have f monic, if possible. We will say more about this later.

We will proceed now by example, and hope the reader can figure out the general case. Suppose we wish to factor the following number
$N = 204426105102611724329344746965636820851660335185692769338127027752316645870633230282574253951$
which is a c132 divisor of $2 \cdot 10^{142} - 1$. (The typical situation is that one wants to factor a number like $2 \cdot 10^{142} - 1$, and after removing a few small divisors, there is still a relatively large divisor left over, like this c132).

We would like to find an $f(x)$ with degree 5. The most straightforward way to do this is as follows: notice that N is also a divisor of the number

$$10^3(2 \cdot 10^{142} - 1) = 2 \cdot 10^{145} - 1000 = 2 \cdot m^5 - 1000,$$

where $m = 10^{29}$. Thus, we *could* take $f(x) = 2 \cdot x^5 - 1000$ and we would have that N is a divisor of $f(m)$ as needed. However, this is a silly choice, and indeed `GGNFS` won't let us do it. The reason is quite simple: we have presumably removed all the small factors from N - in particular, we have already removed any factors of 2. However, $2 \cdot m^5 - 1000$ is trivially divisible by 2, so we can easily remove this factor and still have that N is a divisor of $m^5 - 500$. Thus, we should choose $f(x) = x^5 - 500$.

There are all sorts of little tricks that one can play here to construct polynomials for special form numbers. Several rules of thumb, though:

- It is preferable to have a leading coefficient of 1, if this can be done without enlarging other coefficients by too much.
- It is generally preferable to have the leading coefficient at least smaller than any others if possible.

To illustrate the first point, consider the number $N = 16 \cdot 10^{143} - 1$. If we followed the same construction as above in a literal way, the obvious choice of polynomial would be

$f(x) = 4x^5 - 25$ with $m = 10^{29}$. However, a better choice would probably be $f(x) = x^5 - 200$, with $m = 2 \cdot 10^{29}$. See what we did there? We simply multiplied through by a 200 and noticed that $200N = 32 \cdot 10^{145} - 200 = (2 \cdot 10^{29})^5 - 200$.

One caveat to watch out for: in the above example, we are factoring a c132. However, $f(m)$ is actually a number with 147 digits. Since most of the algorithm really uses just f and m, the level of difficulty is essentially the same as if we were factoring a special c147. Thus, not all special form numbers are equal: we would say that this factorization has *SNFS difficulty 147* even though it is only a 132 digit number. The rule of thumb here is that if the SNFS difficulty is less than about 135% the size of the number, it is surely worth doing it with a special polynomial. However, if we were factoring a 100 digit divisor of something like $7 \cdot 10^{149} - 1$, it would probably be preferable not to even use the special form - instead, it would be more efficient just to treat it as a general number.

Finally, we haven't yet worked out optimal degree choices for `GGNFS`, but a rough rule of thumb is probably that anything with 80-110 digits should probably have a degree 4 polynomial, and 110-160 or so should have degree 5 (perhaps even a little larger: maybe 115-170 digits).

9 Polynomial selection (general numbers)

For general numbers, use the program `polyselect` to construct a polynomial (in fact, the advanced user may want to find several candidates and do some sieving experiments to choose one).

To illustrate, consider the following file for the number RSA-100 (it is distributed with `GGNFS` as `<GGNFS>/tests/rsa100/polysel/rsa100.n`).

```
name: rsa100
n: 152260502792253336053561837813263742971806811496138068865790849458012296325895 28
deg: 5
bf: best.poly
maxs1: 58
maxskew: 1500
enum: 8
e0: 1
e1: 1000000000
cutoff: 1.5e-3
examinefrac: 0.25
j0: 200
j1: 15
```

The 'name' field is optional, but useful. Give your number a descriptive name. The 'n' field is obviously the number we want to factor and 'deg' specifies the degree of the polynomial to search for. In this range of 100 digits, we are right around the break-point between wanting degree 4 and degree 5, but degree 5 seems to be a better choice for most numbers. And degree 5 will suffice for anything upto at least 155 digits or so.

The 'maxs1' figure is used as a filter. The program will go through many choices of polynomials, performing inexpensive tests first. Any polynomials which pass through 'stage 1' will move on and be subjected to more expensive tests. Lower values mean fewer polynomials will make it through stage 1, and so the program will perform the expensive tests on fewer polynomials. Be careful not to abuse this option: while it is true that any polynomials which very high at stage 1 will not make good candidates, it is not true that a lower stage 1 score translates to a better polynomial. So this is used just to skip over polynomials which are obviously poor. The choice of 58 here seems to work well for numbers around 100 digits, but you will have to increase this if you're searching for a polynomial for a larger number. For example, for 130 digits, `maxs1: 72` seems to be a better choice. Expect to have to do some experimentation here to find what seems to be a good choice (or, ask for advice from someone other than me who's already done a factorization with GGNFS at around the same size). When more factorizations have been completed, a table of reasonable values will be made available.

The 'maxskew' field should pretty much be somewhere around 1500 or 2000. Maybe even a bit larger, but don't go too crazy with this. The program will toss out any polynomials with a skew above this figure. Highly skewed polynomials are sometimes much better than lowly skewed ones, but there may be issues if the skew is too high.

The 'enum' option works in conjuction with 'e0' and 'e1'. It specifies that `polyselect` should look at polynomials whose leading coefficient is divisible by this value. Not only that, but it will enumerate them, looking at polynomials with leading coefficients $e0 \cdot enum, (e0 + 1)enum, (e0 + 2)enum, \ldots, e1 \cdot enum$. This should generally be chosen to be a product of small primes, but choose it as small as is reasonable. However, if I were looking for a polynomial for a c130, it would take too long to enumerate all interesting polynomials whose leading coefficients are multiples of 8. So in that case, I would probably choose this to be 720 or so. The idea is that good polynomials will generally have leading coefficients divisible by small primes. But there is the possibility that a much better polynomial may exist with one or two larger prime divisors. So choose this to be as small as possible, while insuring that polyselect will get through enough iterations to look at many polynomials whose leading coefficient has reasonable size (say, at least half as many digits as m, possibly more). Often, some size ranges of the leading coefficient seem to be much more productive than other size ranges (there is a reason for this). You may then choose to hone in on that particular range, looking at more polynomials there: choose a smaller value of 'enum', but take values of 'e0' and 'e1' to start the program looking at leading coefficients in that range.

The value of 'cutoff' tells `polyselect` to store any polynomial with a score higher than this. This is for the advanced user who wishes to have several candidate polynomials to choose from at the end of the process. If you have no particular desire to do this, simply take it to be 1.0. Whatever you do, don't choose it to be too small! If you do, you may quickly start filling your harddrive with a bunch of garbage polynomials! These will be appended to the file 'all.poly', as will each new best candidate as it is found. At any given moment, though, the file 'best.poly' will contain the single best polynomial found so far during the current run.

The 'examineFrac' field is another filter. It is a filter on the size of the 3rd coefficient, and will stop polyselect from performing expensive tests on any polynomials whose 3rd coefficient

is too much larger than a certain relative level. So here, smaller means fewer polynomials will have expensive tests done and so `polyselect` will be able to examine more polynomials. But again - don't go too nuts - while it is generally true that any polynomials whose 3rd coefficient is much too large will be bad, it is not always true that smaller is better! The best choices for this seem to be between 0.2 and 0.4.

A brief word about the two filtering parameters: it is often useful to do several runs with different values. For example, I will sometimes do a quick run with a low-to-medium 'maxs1' value and a lower 'examineFrac' value to filter out many candidates. This lets me do a quick run through many polynomials and quickly get something with a moderate score. But I will do this only so I have some idea what to expect score-wise. I will then go back and tune them back a bit to allow more candidates to pass through the filters and be tested.

Finally, the 'j0' and 'j1' fields are exactly as in Murphy's thesis [8]. After some kind-of-expensive tests, a more expensive sieve-over-nearby-polynomials is done to look for any "nearby" polynomials which have similar size properties, but which may have better root properties (more small roots, meaning more algebraic primes of small norm, and so ideals are slightly more likely to be algebraically smooth). These control the sieving area, so that larger means more sieving. However, it also means slower and if these are too high, we will be looking at polynomials too far from the given candidate, which may then have radically different size properties. Some experimentation still needs to be done here and I'm not entirely sure that this part of the code is working correctly, so just keep these figures in this neighborhood for now.

10 Lattice Sieving

Franke's lattice siever is included in `GGNFS` for pretty fast sieving. There are three binaries to choose from, depending on the size of the number you're factoring: `gnfs-lasieve4I12e, gnfs-lasieve4I13e, gnfs-lasieve4I14e`.

12e	GNFS numbers upto about 110 digits (SNFS 150)
13e	GNFS numbers upto around 130-140 (SNFS 180)
14e	Larger

Although the 12e one will often appear faster than the others even at the higher ranges, you should still use the recommended one. It will generally complete the factorization faster even though the siever itself appears slower. This is because, for example, the 13e siever sieves over a larger area for each special-q. The net result is that more relations are found per special-q. In the worst case scenario, if you are finding too few per special-q by using too small of a siever, you will have to use too many different q's or even run out of worthwhile q's to use. As the q's get larger and larger, the time per relation slows down. Worse still, if you are in a q-range outside the AFB, then remember that the more relations you have with the same special q's, the more full relations you will likely get after combining partials! So, you may wind up needing fewer total relations with the 13e siever than with the 12e. You may even do some experiments to see this for yourself if you don't believe me. The in-between ranges are fuzzy, though, and you should use your own judgement.

Having said that, how do you use it? The siever is actually a slightly modified version of Franke's lattice siever (for those who happen to be familiar with it). You will need a job file. Here is a sample job file:

```
n: 14025157595083242840442484597194076076753846586375228761029529117535725736124211
m: 100000000000000000000000000
c5: 79
c4: 0
c3: 0
c2: 0
c1: 0
c0: -70
skew: 1
rlim: 800000
alim: 699999
lpbr: 25
lpba: 25
mfbr: 44
mfba: 44
rlambda: 2.2
alambda: 2.2
q0: 700000
qintsize: 50000
```

The lines down to 'c0' are just the polynomial. The 'skew' figure is computed automatically for you if you use polyselect to generate a polynomial, but you need to choose it by hand for special numbers. It should be at least 1, and will be larger depending on the approximate ratio of the size of the coefficients from high-order to low order. For most SNFS numbers, it will be between 1 and 10 or so; you can take a guess, do some sieving, and try to adjust it to see if you get a speed increase. Do this several times until you've found a good choice.

Then 'rlim' is the limit for the RFB, 'alim' is the limit for the AFB. 'lpbr' is the max number of bits in a large rational prime, 'lpba' the same on the algebraic side. 'mfbr' is the max number of bits for which sieve-leftover stuff will be factored (i.e., to find leftover large rational primes). 'mfba' is the same on the algebraic side. 'rlambda' describes how far from perfect sieve values we should actually look for good relations. It is essentially a fudge factor, and higher means more sieve entries will be closely examined. If this is too low, you'll miss too many potential good relations. If it's too high, the siever will spend too much time looking at locations which do not give a good location. Generally, good values are somewhere between 1.5 and 2.6 (maybe even a little larger - I haven't done any very large factorizations yet, so I don't have enough experience to say for sure). You guessed it - 'alambda' is the same thing on the algebraic side.

Finally, 'q0' is the first special q to be examined and 'qintsize' specifies that the siever should sieve over all special-q in the range 'q0 + qintsize'. The siever insists that 'q0' should be greater or equal to 'alim' (for sieving with special-q from the algebraic side; for special-q from the rational side, it would have to be greater or equal to 'rlim').

If the job file above is called 'snfs.job', then a typical invocation of the lattice siever might be

```
gnfs-lasieve4I12e -k -o spairs.out -v -a snfs.job
```

You can look in the `src/lasieve4` directory at Franke's documentation to see what the options are. After a sieve run as above, the output will be in the file 'spairs.out'.

11 Relation processing

While the lattice siever does create a factor base, it may not necessarily be the same size as the factor base you want (in particular if you want to trick it into sieving with some special-q from the AFB which you want to use). Thus, before the first time you do relation processing, you'll actually need to run 'makefb' to create a factor base which the `GGNFS` programs can use. For this, you need to give it the polynomial and other parameters (rlim, alim, lpbr, lpba). These values can be supplied on the command-line or from with the polynomial file.

Anyway, having made a factor base, the siever output is processed by the program 'procrels'. Such processing is needed from time to time to see how many full relations you have (a full relation is actually a relation-set, resulting from combining almost smooth relations to "cancel out" leftover large primes). The basic useage is fairly simple, and might look like:

```
procrels -fb snfs.fb -newrel spairs.out -maxrelsinff 40
```

This will take the new relations, complete their factorizations (i.e., ramified primes and such which are not actually in the AFB), filter out duplicates, and give you some idea how many more you need. If there are many relations (i.e., a few million), this may begin to take a noticeable time and it will surely take a noticeable amount of disk space to store the binary data files.

It is completely incremental - each time you run it on an spairs.out file, procrels will take the relations from that file, filter out duplicates, complete the factorizations, and add the resulting information to it's binary data files. It will not have any use for the spairs.out file again.

The final option, '-maxrelsinff 40' is telling it not to use more than 40 relations per relation-set (I think some people call this the 'merge-level', but I'm not sure whether or not it is exactly the same thing). 'procrels' will tell you if there were enough full relations, and it will build the matrix files needed by 'matsolve'. If you get a resulting matrix which is far too dense, you can lower this to 32 or even 24, but you'll have to do more sieving to make up the difference.

There is probably much more I could say about relation processing, but I'll let it go for now. However, there is one other option which is useful from time to time. If, for some reason, you believe that your binary files may have been corrupted or something very funny has happened, you can instruct the program to dump all relations to file in siever-output format. You could then delete all binary data files and other junk, and reprocess all relations

without having to repeat all the sieving. The command-line option for performing this dump is, plainly enough, '-dump'. You'll just need to also give it the name of the '.fb' file. It will dump relations to files spairs.dump.0, spairs.dump.1,... and so on, with upto 500K relations per file. Be aware that this can take quite alot of disk space and reprocessing all of those relations is going to take some time when you get to it. So try to avoid doing this if you can help it. But it beats having to start a factorization over from scratch. It also provides a means by which you can change factorization parameters mid-factorization (i.e., perhaps you decide to change factor base limits or something).

12 The matrix step

This is easy enough - if procrels tells you that it wrote such-and-such data files and "you can now run matsolve", then you are ready for this step (maybe - you might choose to do a little extra sieving to get a more overdetermined matrix which can then be pruned down to a smaller, less dense matrix which will solve quicker). Running it is quite easy: jut run it from within the directory where the data are living. It will find everything on it's own, solve the matrix, and store the result in a canonically named file.

There are some undocumented command-line options for the program which can control certain things. These will be documented in the future, but they're not necessary.

The first thing matsolve will do is take advantage of the fact that the matrix is (hopefully) very overdetermined to do some pruning. It will throw out some dense columns, throw out columns corresponding to rows with only one nonzero entry, toss out empty rows and so on. It will also do some combining of columns to reduce the matrix dimensions without increasing the matrix weight by too much. It is a very ad-hoc procedure which works acceptably. Once it gets to a point where it cannot shrink the matrix any further, it will use Montgomery's block Lanczos algorithm [7] to find 32 vectors in the kernel (in fact, it will find 64 of them, but 32 will be discarded since the remaining 32 will already be more than enough).

13 The square root step

Okay, so if the matrix step succeeded, your almost ready to receive the final factorization. Sample useage of the sqrt program is as follows

```
sqrt -fb snfs.fb -deps deps -depnum 0
```

Obviously, snfs.fb is the name of my factor base file. The 'deps' is simply the name of the file where matsolve records the dependencies by default, and '-depnum 0' indicates that we wish to run the program on dependency 0.

Recall that a dependency may produce the trivial factorization with about a 50% chance. If this happens, no problem: re-run it with '-depnum 1' to try again with dependency 1, and so on, until you get it. On success, the divisors will be output on screen as 'r1=...' and 'r2=...' and stored in the logfile, 'ggnfs.log'. These will be followed by `(pp ¡digits¿)` or `(c ¡digits¿)` depending on whether the given divisor is probable prime or composite.

14 Running the script

Luckily for all of us, there is a Perl script which can perform nearly all of the above steps automagically. As of 0.70.2, the only thing it cannot do is to construct a polynomial for SNFS numbers.

It can choose parameters and run all the programs in order automatically. The parameter choices it makes may not be optimal, so you may still want to learn a good deal about the process so you can tweak it as you go.

I'll say more later about customizing the script and its capabilities. For now, let's see how to run it:

Case I: I have a polynomial file for the number I want to factor. It's called 'snfs-c112.poly'. This is easy:

```
factLat.pl snfs-c112
```

or

```
factLat.pl snfs-c112.poly
```

Case II: I have a general number I want to factor. It is moderate sized (say, less than 110 digits), and I'm willing to accept the fact that the script will make sub-optimal choices and the factorization might take longer than it should. In this case, make a new text file with the single line:

```
n: my-number-here
```

call it myc102.n. Then just do

```
factLat.pl myc102
```

The script will look first for a file called 'myc102.poly'. When it cannot find it, it will assume you want the script to generate it for you, so it will lookup some parameter choices and launch polyselect. There is also a built in cap for how much time it will spend looking for a good poly, so just let it go. It will do everything on it's own, and eventually (hopefully) return to you with a factorization in a day or so (for a c100 - expect several days for a c110 and maybe a week or a little over for a c120). Of course, the timings depend on the machine - these are roughly for my Athlon 2800+ laptop.

References

[1] E. Bach and J. Shallit. *Algorithmic Number Theory, Vol. 1.* The MIT Press, Cambridge, 1996.

[2] D. Bernstein and A. Lenstra. A general number field sieve implementation. In A. Lenstra and H. Lenstra, editors, *The development of the number field sieve*, Lecture Notes in Mathematics 1554. Springer-Verlag, 1993.

[3] H. Cohen. *A Course in Computational Algebraic Number Theory*. Springer-Verlag, New York, third edition, 1993. Corrected Printing, 1996.

[4] S. Lang. *Algebraic Number Theory*. Springer-Verlag, 2 edition, 1994.

[5] A. Lenstra and H. Lenstra, editors. *The development of the number field sieve*. Lecture Notes in Mathematics 1554. Springer-Verlag, 1993.

[6] Peter L. Montgomery. Square roots of products of algebraic numbers. In *Mathematics of Computation 1943–1993: a half-century of computational mathematics (Vancouver, BC, 1993)*, volume 48 of *Proc. Sympos. Appl. Math.*, pages 567–571. Amer. Math. Soc., Providence, RI, 1994.

[7] Peter L. Montgomery. A block Lanczos algorithm for finding dependencies over GF(2). In *Advances in cryptology—EUROCRYPT '95 (Saint-Malo, 1995)*, volume 921 of *Lecture Notes in Comput. Sci.*, pages 106–120. Springer, Berlin, 1995.

[8] B. Murphy. *Polynomial selection for the number field sieve factorization algorithm*. PhD thesis, The Australian National University, 1999.

[9] Ken Nakamula. A survey on the number field sieve. In *Number theory and its applications (Kyoto, 1997)*, volume 2 of *Dev. Math.*, pages 263–272. Kluwer Acad. Publ., Dordrecht, 1999.

[10] J. Pollard. Factoring with cubic integers. In A. Lenstra and H. Lenstra, editors, *The development of the number field sieve*, Lecture Notes in Mathematics 1554. Springer-Verlag, 1993.

[11] J. Pollard. The lattice sieve. In A. Lenstra and H. Lenstra, editors, *The development of the number field sieve*, Lecture Notes in Mathematics 1554. Springer-Verlag, 1993.