# The Quadratic Sieve - introduction to theory with regard to implementation issues

RNDr. Marian Kechlibar, Ph.D.

April 15, 2005

# Contents

# Part I

# The Quadratic Sieve

# Chapter 1

# Introduction

Question whether a given natural number has some nontrivial divisors is as old as mathematics. A slightly modified question - to determine the actual divisors of a given non-prime number - seemed for a long time to be 'almost' equivalent. Indeed, it still seems so for a man without training in number theory or cryptography.

The question of primality is the easier one. As for the time of the writing, there already is an algorithm, that solves this question in polynomial time (see [1]). Though having perfect asymptotic behavior, and an extensive theoretical background, this algorithm is far from practical. Instead, much faster probabilistic algorithms are used to tackle the primality problem in the real world.

The question of divisors seems to be significantly harder. Not only that there is no polynomial-time algorithm for divisor determination, but there is even lack of probabilistic ones. There are good algorithms to determine small divisors, but all current algorithms fail to reliably determine divisors of a number composed of two large primes in reasonable time.

This has led the famous RSA team to design of an asymmetric cipher (see [21]), which exploits precisely this quality. This cipher is worldwide used in encryption of communication, including transfers of top secrets and large amounts of money.

During three decades following publication of RSA, there have been attempts to attack the divisor problem from many sides. The most effective approach so far has been to search for pairs $X, Y \in \mathbb{Z}_N$ ($N$ is the number to be factored) such that

$$X^2 = Y^2 \mod N. \tag{1.1}$$

This leads to congruence $(X-Y)(X+Y) = 0 \mod N$, and unless $X = \pm Y \mod N$, the congruence immediately gives two proper divisors of $N$: $gcd(X-Y,N)$ and $gcd(X+Y,N)$. The former case of trivial divisors happens with probability at most 1/2. So, finding $k$ congruences $X_i^2 = Y_i^2 \mod N$ gives us a chance at least $1-(1/2)^k$ to factor $N$.

A natural question "how to generate such congruences" has been solved in several ways, the first being continued fraction algorithm, and the last being the number field sieve (NFS). We will concentrate on the predecessor of the NFS - the quadratic sieve, proposed by [20], with first major improvement described in [24], and next one proposed independently by [19] and [2].

## 1.1 The Quadratic Sieve - short description

### 1.1.1 Polynomials and relations

From this article on, $N$ will be the factored number.

Let us have polynomials $Q_j(x) = (2a_jx+b_j)^2 - N \in \mathbb{Z}[x], j \in J$, which satisfy the following equation:

$$b_j^2 = N \mod 4a_j \tag{1.2}$$

The quality 1.2 causes the following to be true:

$$(2a_jx+b_j)^2 - N = 4a_j^2x^2 + 4a_jb_jx + b_j^2 - N = 4a_j^2x^2 + 4a_jb_jx + 4a_jc_j =$$

$$= 4a_j(a_jx^2 + b_jx + c_j),$$

where $c_j$ is equal to $\frac{b_j^2-N}{4a_j}$. Specifically, we see that any value of $Q(x)$ is divisible by $4a_j$, which will be of good use.

Now, let us assume that some $Q_j(x)$ factors for some $x$ into relatively small primes (some of the exponents may be zero):

$$Q(x) = p_1^{k_1} \cdot p_2^{k_2} \cdots p_l^{k_l}$$

This equality holds in integers, so *a fortiori* modulo $N$. But modulo $N$, an even more important equality holds:

$$(2a_jx+b_j)^2 = p_1^{k_1} \cdot p_2^{k_2} \cdots p_l^{k_l} \mod N$$

**Definition 1.1.1.** Any equality of type

$$X^2 = p_1^{k_1} \cdot p_2^{k_2} \cdots p_l^{k_l} \mod N$$

will be called a *relation*.

A number $Y$ is called $B-$smooth, if its prime decomposition

$$Y = q_1^{m_1} \cdot q_2^{m_2} \cdots q_n^{m_n}$$

satisfies $q_i < B$ for all $i = 1 \ldots n$.

Let us have two relations generated by functional values of polynomials $Q_j$ and $Q_n$, say

$$(2a_j x + b_j)^2 = p_1^{k_1} \cdot p_2^{k_2} \cdots p_l^{k_l} \mod N$$

and

$$(2a_n y + b_n)^2 = p_1^{m_1} \cdot p_2^{m_2} \cdots p_l^{m_l} \mod N.$$

Then their product is another relation:

$$((2a_j x + b_j)(2a_n y + b_n))^2 = p_1^{k_1+m_1} \cdot p_2^{k_2+m_2} \cdots p_l^{k_l+m_l} \mod N$$

This principle generalizes to any finite set of relations, not only two. The way to factorization is now as follows: if we gather enough relations, we may pick up a suitable subset of them, such that exponents of all primes on the right side will be even. Such a relation

$$\left(\prod_i (2a_{r(i)} x_i + b_{r(i)})\right)^2 = p_1^{2s_1} \cdot p_2^{2s_2} \cdots p_l^{2s_l} \mod N \tag{1.3}$$

(where $r(i)$ determines number of used polynomial) gives equality of two squares modulo $N$:

$$\left(\prod_i (2a_{r(i)} x_i + b_{r(i)})\right)^2 = (p_1^{s_1} \cdot p_2^{s_2} \cdots p_l^{s_l})^2 \mod N,$$

and, setting $X = \prod_i (2a_{r(i)} x_i + b_{r(i)})$, $Y = p_1^{s_1} \cdot p_2^{s_2} \cdots p_l^{s_l}$, we have an instance of 1.1, which is what we wanted.

### 1.1.2 Smooth and partial relations

**Definition 1.1.2.** Let $B$ be an upper bound on primes $p_i$ acceptable in factorization of $Q_j(x)$. Then:

$$X^2 = p_1^{k_1} \cdot p_2^{k_2} \cdots p_l^{k_l} \mod N,$$

where all $p_i < B$, is called a *smooth relation* with respect to smoothness bound $B$, and

$$X^2 = p_1^{k_1} \cdot p_2^{k_2} \cdots p_l^{k_l} \cdot P \mod N,$$

where $B \leq P$ is a prime number, is called a *partial relation* with respect to smoothness bound $B$. Sometimes, we will refer to a partial relation as to a *1-partial relation.*

An expression

$$X^2 = p_1^{k_1} \cdot p_2^{k_2} \cdots p_l^{k_l} \cdot P \cdot Q \mod N,$$

where $P, Q > B$ are distinct primes, is called a *partial-partial relation*, or *2-partial relation.*

3- and more partial relations are defined similarly.

If, in addition to *smooth* relations, we collect also *1-partial* relations[1], they can be of good use. Any two partial relations which share the same variation yield together a suitable combination:

$$(2a_jx + b_j)^2 = p_1^{k_1} \cdot p_2^{k_2} \cdots p_l^{k_l} \cdot P \mod N$$

and

$$(2a_ny + b_n)^2 = p_1^{m_1} \cdot p_2^{m_2} \cdots p_l^{m_l} \cdot P \mod N$$

give

$$((2a_jx + b_j)(2a_ny + b_n))^2 = p_1^{k_1+m_1} \cdot p_2^{k_2+m_2} \cdots p_l^{k_l+m_l} \cdot P^2 \mod N.$$

We see that such relation is usable instead of a smooth one, because presence of a large *s*quare factor does not influence usability of a relation in 1.3.

How often does really occurence of two partial relations with the same variation occur? This is an instance of so-called *birthday paradox.* Though probability that two partial relations will have a concrete given variation

[1]This variant of the quadratic sieve is called *The Single Large Prime Variation.*

in common is small, the probability of some two relations sharing the same variation is much larger. It can be seen that the more partial relations are collected, the bigger chance for some two of them to have a common variation.

A thorough analysis of effect of collecting 1-partials on runtime can be found in [14] and [4].

A more complicated situation emerges, if we attempt to use $2-$, $3-$ and more-partial relation. We will discuss it in the next subsection.

### 1.1.3 The Double Large Prime Variation

An instance of quadratic sieve collecting smooth, 1-partial and 2-partial relations is called *The Double Large Prime Variation.* Unlike the Single LPV case, implementations of the Double LPV are faced with serious issues concerning used algorithms and techniques. These issues stem from the following properties of the Double LPV:

- It is not trivial to combine the collected partial relations to smooth ones effectively.
- The set of gathered relations, especially the 2-*partial* ones, is often too big to fit in memory.

The question of memory-savvy representation of the gathered relations is dependent on the way we choose to attack the first property; therefore, we will discuss it later.

As for the combination of the collected partial relations, the problem can be redefined in language of graph theory. Let us construct a graph having the following properties:

1. All the large primes ever met in the set of 1-*partial* and 2-*partial* relations are vertices of the graph; moreover, there is one vertex representing the integer 1.
2. All the 2-*partial* relations will be represented as edges between the two large primes.
3. All the 1-*partial* relations will be represented as edges between their large prime and 1.

Having constructed this graph, we see that finding a "good subset" of the relations set, that is, a combination having all of the large primes in even power, is equal to finding a cycle in the constructed graph. Two facts must be taken into account:

- It is necessary to include the 1-*partials* in the graph. If we try generating cycles from 2-*partials* only, we soon find out that their number is orders of magnitude lower than number of cycles generated from both 1- and 2-*partials* - sometimes as much as 1:10000. This is caused by the fact that there are many "almost perfect" subsets, which need only one prime to be raised to even power - and that is where the 1-*partials* help.
- We do not need all of the cycles. In fact, we must avoid many of them. Let us say that $C_1$ and $C_2$ are two cycles with nonempty cycle $C_3$ as an intersection. It is clear that $C_1, C_2, C_3$ are not independent; together, they give three linearly dependent rows (mod 2), which is strongly undesired. Occurrence of linearly dependent rows in the final matrix would reduce the chance to find a nontrivial solution.

Therefore, in order to use Double LPV, we must use an algorithm which will be memory-savvy, fast enough and which will detect only independent cycles. Such algorithm is well-described in [14]; our current implementation of MPQS/SIQS uses this algorithm with some minor variations.

By the Euler's theorem, number of independent cycles in a graph is equal to $e+c-v$, where $e$ is the number of edges, $c$ the number of components and $v$ the number of vertices in the graph. Therefore, if we keep those values in memory, and if our graph representation is accurate enough to reflect them precisely, we always know how many relations can be generated from the large set of x-*partials*, and thus know when to stop collection of the relations and start the construction of cycles.

Unlike in Single LPV, Double LPV requires also a method for distinguishing the 2-*partials* from unusable values. In Single LPV, any remainder which lasted after dividing out the factor base primes, and which was lower than $B^2$ (we usually impose a lower bound, say $128B$), was automatically a prime; in case of Double LPV, the remainder is between $B^2$ and $B^3$, and any of the three possibilities may happen:

- The remainder is a prime, which rules the value out of further consideration.

- The remainder is a product of two primes, but one of the primes is too small (around $B$) and the other too large (close to $B^2$), therefore the value is discarded again.

- The remainder is a product of two primes of acceptable size, and therefore the value is regarded a 2-*partial* relation.

In order to find the truth about a "candidate value", the remainder must be subject to primality testing (which is relatively fast, as the size of the remainder seldom exceeds 50 bits), and if it turns out to be composite, then it must be factored into its prime factors, using some factoring algorithm. This is not a trivial process; in fact, it may take as much as 50% of the total MPQS/SIQS runtime.

The situation of the programmer is even worsened by the fact that picking up a fast factoring algorithm for purpose of factoring the remainders is more of a guess than mathematics. The trial division is definitely out, but what for other algorithms? Their asymptotic behavior is well-known, but it would be a bad clue, since it only works in large numbers; in the realm of 40-50 bit numbers, the effectivity of factoring algorithms may be totally different from their asymptotic properties. The only "sensible" approach seems to be to test several algorithms on particular data, watch their performance, and then choose the fastest one. Such comparison can be programmed as well, removing the need of a human in control.

There used to be general consent that while $2-$partials are effective, collection of $3-$partials is too time–consuming to be outweighted by the increase of amount of cycles in the graph. However, it has been referred ([8]), that collection of $3-$partials *may* be in some cases advantageous because of "non-continuous" behavior of graphs. This field of research surely deserves more attention.

### 1.1.4 Problems to solve

A succesful implementation of the described quadratic sieve algorithm has to solve the following problems:

- How to pick up the set of "acceptable" primes for the right side? If they are too many, a large set of relations is needed to yield some equalities of type 1.1; on the other hand, if they are too few, it will be hard to generate any suitable relations at all.

- How to find the relations fast enough?
- How to generate suitable polynomials $Q_j(x)$? Intuitively, some choices of coefficients of $Q_j(x)$ will yield more relations than other ones.
- How, in the set of all gathered relations, to find some subset yielding 1.1? Preferrably, the method used for finding a 'good' subset should output several such subsets at a time, since each equality 1.1 gives some chance, but not 100% one, to actually factor the number.
- Are there any mathematical 'traps' which must be avoided?

These questions will be answered in the next chapter.

# Chapter 2

# Quadratic Sieve Implementation

## 2.1 The Factor Base

The first tisk is to select a suitable set $FB$ of primes, which will be allowed to divide values of $Q_j(x)$. This set of primes will be called *the factor base*. Beside prime numbers, it is convenient to insert $-1$ into the factor base - this enables us to exploit negative values of $Q_j(x)$ in search for the relations.

First of all, we must notice a mathematical limitation of the factor base. Imagine that prime $p$ belongs to factor base, and that

$$(2a_jx + b_j)^2 - N = p^k \cdot q$$

is a relation where $k > 0$ and $q$ is a product of some other primes (we intentionally take the relation in integers, not $\mod N$). From this, we see that

$$p \,|\, \left((2a_jx + b_j)^2 - N\right)$$
$$(2a_jx + b_j)^2 = N \mod p,$$

which means that equation $Z^2 = N \mod p$ has a solution (namely, $Z = 2a_jx + b_j$). But this it not the case for every prime $p$. Such situation cannot happen when $N$ is a quadratic nonresidue mod $p$, which will be for approximately half of all the primes. So our factor base will consist (at best) of $-1$ and those primes, for which $N$ is a quadratic residue, limited in size by some constant $B$.

The fact that for a given $N$, $N \mod p$ is a quadratic nonresidue mod $p$, can be changed by changing the factorized value itself. It is especially desirable to have $p = 2$ in the factor base, and this is achievable by multiplying the factored value $N$ by small multiplier $k$, so that $kN = 1 \mod 8$. A good candidate for $k$ is $N \mod 8$ itself, because all odd numbers $N$ have the quality $N^2 = 1 \mod 8$. This change enables us to insert 2 into the factor base. This is important for two reasons: first, 2 will divide many of the $Q(x)$'s to a high power; second, representation of numbers in a computer makes division by any power of 2 a trivial operation (the right shift, in C represented by operator >>), which is much faster than normal division.

In a similar way, we may multiply $kN$ by another multipliers $k_i$, in order to insert $3, 5, 7$ or other small primes into the factor base, but this effect is negated by the fact that with increasing value of $kN$, the probability that an average $Q(x)$ factors into small primes (and thus gives a relation) decreases. From this on, let us forget about the multiplier and consider $N$ as already multiplied by suitable number $k$.

Another question is to determine the optimal upper limit $B$ for the factor base. Let us say that we have collected *enough relations*, if the set of collected relations (smooth and optionally $n-$partials), processed by a given algorithm $A$ for finding combinations, is big enough to give rise to some combinations of type 1.3. We see that for small $B$, fewer relations will be needed, but they will be scarce - since very few numbers factor completely over a small set of primes; for a large $B$, there will be many relations, but more of them will be needed to generate some 1.3-like combinations. We will call the value of $B$, for which the process of gathering relations takes the least time, the *optimal value of* $B$.[1]

Optimal factor base size for a given $N$ is usually determined by trial-and-error. Exact determination in practice fails on the fact that implementations vary in efficiency of their steps. The fundamental mathematical quality that will influence runtime is the probability that a randomly chosen value of some size will produce a relation - and that is also the only part of this problem, where we can rely on mathematical theory.

**Theorem 2.1.1.** *Let $x$ be a factorized value, $B$ be the upper bound on prime divisors of $x$ and denote by $r$ the ratio $\frac{\log x}{\log B}$. Then, if $r << B$, we have probability of $x$ factorizing fully into primes smaller than $B$ approximately $r^{-r}$.*

*Proof.* Let us denote the number of primes smaller than $B$ as $\pi(B)$ and

[1]It is widely believed that the function representing the dependence of runtime of gathering relations on size of $B$ has only one minimal value, but this has not been proven.

number of numbers $\leq x$ such that they factor fully into primes smaller than $B$ as $\Psi(x, B)$.

We see that $\Psi(x, B)$ is in 1-to-1 relation with the set of all $\pi(B)$-tuples of nonnegative integers $\alpha_j$ such that

$$\prod_j p_j^{\alpha_j} \leq x.$$

This equation can be logarithmed into

$$\sum_j \alpha_j \log p_j \leq \log x.$$

Now, we may make an approximation: for most of the primes $\leq B$, we have $\log p \approx \log B$. The altered equation

$$\sum_j \alpha_j \log B \leq \log x$$

is thus turned into

$$\sum_j \alpha_j \leq r.$$

So, the question to solve is, how many such unique $\pi(B)-$tuples do really exist.

The answer is $\binom{[r]+\pi(B)}{\pi(B)}$. This can be shown by the following combinatorial argument: let us $\alpha_1, \ldots, \alpha_{\pi(B)}$ be one such $\pi(B)-$tuple. We will generate sequence $\beta_1, \ldots, \beta_{\pi(B)}$ such that $\beta_1 = \alpha_1 + 1$ and $\beta_{j+1} = \beta_j + \alpha_{j+1} + 1$. This clearly gives a 1-to-1 correspondence to the set of all $\pi(B)-$subsets of set having $[r] + \pi(B)$ elements.

Now, knowing that there are approximately $\binom{[r]+\pi(B)}{\pi(B)}$ $B-$smooth numbers smaller than $x$, let us estimate the logarithm of the probability that a randomly chosen number from this range will be $B-$smooth. With use of Stirling formula $\log(n!) \approx n \log n - n$, we get

$$\log\left(\frac{\Psi(x, B)}{x}\right) = \log(\Psi(x, B)) - \log x =$$

$$= \log\left(\frac{([r] + \pi(B))!}{[r]!\pi(B)!}\right) - r \log \pi(B)$$

$$\approx ([r] + \pi(B)) \cdot \log([r] + \pi(B)) - ([r] + \pi(B)) - ([r] \log [r] - [r]) -$$
$$-(\pi(B) \log \pi(B) - \pi(B)) - r \log(\pi(B)).$$

After another approximations, namely $[r] \approx r$ and $\log(r+\pi(B)) \approx \log(\pi(B))$ (the latter from assumption that $r << B$), most of the terms cancel out, and we get

$$\frac{\Psi(x, B)}{x} \approx r^{-r},$$

which was to be demonstrated. □

Based on this calculation, we should make a computation to determine the optimal value of $B$. However, this computation is practically useless, for two reasons:

- the process of gathering relations, as we will see, depends more on the speed of memory access than on the speed of arithmetic operations,
- the above calculation is very rough. It does not count with the fact that only half of the primes can be used, neither with the fact that generated values are not perfectly random; as a consequence, it does not work as expected.

In practice, the optimal FB size does not seem to be sharply determined; some experimental results are demonstrated in the next chapter. However, one must remember that with increasing factor base size, not only the number of relations needed to build an equality 1.1 increases, but also the time to build such equality with help of algorithm $A$ rises, and so, if two different FB sizes lead to similar results in runtime, it is always advantageous to choose the smaller one.

## 2.2 The sieving process

The process of gathering relations is called sieving.

Sieving means taking an interval $[-M, M]$ and determining, for which $x \in [-M, M]$ a given prime $p \in FB$ divides $Q_j(x)$. This must be done for all primes in the factor base. At the end of the sieving of an interval, we know (partial or complete) factorizations of all $Q_j(x)$ values, from which we choose smooth and $n-$partial relations.

This process is repeated for many pairwise different polynomials $Q_j(x)$, until enough relations have been found.

### 2.2.1 Interval sieving and solution of polynomials

The determining of $x$'s, for which $p \mid Q_j(x)$, is done with use of this simple principle:

$$p \mid Q_j(x) \iff p \mid Q_j(x+kp),\ k \in \mathbb{Z}.$$

Having solved the equation $Q_j(x) = 0 \mod p$ (which usually leads to two different integer roots $x_p$ and $x'_p = p - x_p$), we walk through the sieving interval, and mark numbers

$$\ldots, x_p - 2p, x_p - p, x_p, x_p + p, x_p + 2p, \ldots$$

$$\ldots, x'_p - 2p, x'_p - p, x'_p, x'_p + p, x'_p + 2p, \ldots$$

as having the quality $p|Q_j(x)$.

Solving the equation $Q_j(x) = 0 \mod p$ is easy, since this equation reduces to

$$(2ax+b)^2 = N \mod p$$

for all polynomials $Q_j(x)$.

At the start of the quadratic sieve, before any sieving is done, we compute and save values of $r_p = \sqrt{N} \mod p$ for all primes from the factor base. After that we do not need to perform any further root extractions, and computation of roots of $Q_j(x)$ reduces to the following two calculations:

$$x_p = \frac{-b_j + r_p}{2a_j},\ x'_p = \frac{-b_j - r_p}{2a_j}.$$

The most computationally intensive part of this calculation is taking the modular inverse $(2a_j)^{-1} \mod p$ for all primes $p \in FB$. Number $a_j$ is usually a long integer, with size roughly half of size of $N$ (see 2.3.2), and so the modular inversions may take a high proportion of the entire factorization process runtime. If $a_j$ is altered in every polynomial change, this proportion may easily reach 50%! So, one of the desirable properties of polynomials will be to keep the same value of $a_j$ for many of them.

### 2.2.2 Practical implementation

From practical point of view, saving of all intermediate factorizations of $Q_j(x)$'s in interval $[-M, M]$ is not a good idea. These factorizations take up memory space and their management costs time. A simpler system is usually used: instead of saving the factorizations, we only maintain an array of bytes, with one entry per one $x$ (so, having $2M + 1$ entries altogether), and if some prime divides a value of $Q_j(x)$, we add its logarithm to the corresponding entry of the array. At the end of the sieving process the $x$'s, whose entries exceed some bound, are chosen for more detailed processing. This process is remarkably (up to 10 times) faster than saving of intermediate factorizations, and its speed gain rises with size of the factored numbers, as the incidence of relations grows smaller.

Surprisingly, this process can be still fastened by a clever trick of Contini (see [7]). Say that $E$ is the bound, at which the entry is considered a good candidate for further processing. If the array of bytes is initialized with values `0x80`$-E$ (which costs the same time as initialization by zeros), the test for being a suitable candidate changes into testing, whether the entry has reached `0x80`. Now, in case of normal PCs, one can typecast the array to an array of integers, and test `entry_block & 0x80808080`. Nonzero means that at least one of the entries in a four-entry block is a candidate for processing; zero, which happens much more often, means that this entire block may be safely ignored.

## 2.3 Generation of polynomials

Another question to solve is the optimal choice of coefficients.

A quadratic polynomial is close to optimum, if, on the interval $[-M, M]$, it generates as much smooth and $n-$partial relations, as possible. With growth of the factored number, the frequency of smooth and $n-$partial relations grows naturally smaller; we shall try to maximize the probability.

In this section, we drop the notation $Q_j(x)$, which is used in other sections to underline the fact that there are many polynomials involved in sieving, and switch to notation

$$Q(x) = (2ax + b)^2 - N,$$

which will represent a polynomial satisfying 1.2.

### 2.3.1 Desirable properties of polynomials

From the implementation point of view it is easiest to factor values of $Q(x)$ in some sieving interval $[-M, M]$ around zero. We know that $Q(x)$ is always divisible by $4a$, and only the remainder $r(x)$ after division $Q(x)/4a$ needs to be factored. We may take this remainder as a random number in interval $[min(Q(x)/4a), max(Q(x)/4a]$. Assuming this, the goal is to reduce the absolute value of $r(x)$ as much as possible.

Before we go on, we should remark on the limitations of the shape of the parabola $Q(x)$. Assume that $a \geq 1$ is an integer. $Q(x)$ always has two distinct real roots

$$r_0 = \frac{-b - \sqrt{N}}{2a} < r_1 = \frac{-b + \sqrt{N}}{2a}.$$

Let $Q'(x) = 2a(2ax+b)$ be the first derivative of $Q(x)$. The value of derivative at the roots is

$$Q'(r_0) = -2a\sqrt{N} < -\sqrt{N},\ Q'(r_1) = 2a\sqrt{N} > \sqrt{N},$$

which means that the parabola cannot be 'too flat'.

It is desirable to incorporate both real roots of $Q(x)$ into the sieving interval $[-M, M]$, since around them the values of $Q(x)$ are small and have a good chance to factor over the factor base. Also, it is desirable for $Q(x)$ to be symmetric, or at least "almost symmetric" - to have the axis of symmetry close to $x = 0$. Then the 'good properties' from one side will be reflected on the other. The requirement of symmetry can be expressed as

$$|b| < |2a|,$$

which ensures position of axis of symmetry between $x = 0$ and $x = 1$. This is not achievable in all polynomial instances (as we will see later in SIQS case), but a 'moderate' shift, say between $x = 0$ and $x = 30$, is acceptable as well.

### 2.3.2 Assessment of magnitude of coefficients

Minimization of average absolute values of an 'almost symmetric' $Q(x)$ over an interval $[-M, M]$ can be roughly expressed as:

- minimization of $\sup |Q(x)|$ on $[-M, M]$,
- or minimization of integral

$$\int_0^M |Q(x)| dx,$$

both with regard to coefficients $a, b$ and condition that $b^2 - 4ac = N$.

The first minimization task is easier, as we know that $Q(x)$ reaches its extremal values at the ends and in the middle of the interval. At the middle of the interval, for $x = 0$, we have $Q(x) = b^2 - N$, which, unless $b$ is quite large (over $\sqrt{N}$), is close to $-N$. At the ends of the interval, the corresponding value is approximately $4a^2M^2 - N$. In order for these two values to be approximately identical, we must have

$$\begin{aligned} 4a^2M^2 - N &= N \\ 2a^2M^2 &= N \\ a &= \sqrt{\frac{N}{2M^2}}. \end{aligned}$$

Of course, $a$ must be an integer, so we cannot get the ideal value exactly. In practice, difference of $0,1\%$ can be considered small enough.

The second minimization task is somewhat more complicated. It is easily seen that the positive root of $Q(x)$ is equal to $r_1 = \frac{-b+\sqrt{N}}{2a}$, and so

$$\int_0^M |Q(x)|dx = \int_{r_1}^M Q(x)dx - \int_0^{r_1} Q(x)dx =$$

$$\frac{1}{6a}\left[(2ax+b)^3 - Nx\right]_{r_1}^M -$$

$$-\frac{1}{6a}\left[(2ax+b)^3 - Nx\right]_0^{r_1}.$$

This yields a real function of two variables

$$F(a, b),$$

whose minimum is of our interest; in combination with the conditions $1 \leq a \leq N$, $b < 30|a|$, the Lagrange multipliers become too complicated to be solved by hand. A result given by Maple 7 is

$$a \approx 0.7335 \cdot \sqrt{\frac{N}{M^2}}.$$

The results for both optimizations are thus quite similar ($\frac{\sqrt{2}}{2} \approx 0,7071$). Let us, for simplicity, further assume that

$$a \approx \sqrt{\frac{N}{2M^2}}. \tag{2.1}$$

The preceding paragraph raises the question of generating values of $a$ (and consequently $b$) compliant to 2.1.

### 2.3.3 MPQS - The Silverman Method

This method was described in [24]. Its principle is very straightforward - $d$ is chosen to be a prime number of magnitude

$$d \approx \sqrt{\sqrt{\frac{N}{2M^2}}}. \tag{2.2}$$

We give $a$ as $d^2$.

Generation of $b$ is limited by the requirement 1.2. This is achievable by combination of two steps:

1. taking $b_0, b_1$ as solutions of quadratic equation $x^2 = N \mod a$,
2. picking the odd one of the pair $b_0, b_1$ as desired $b$.

The first step ensures that $b^2 - N$ is divisible by $a$. Now, as $d$ is a large prime, $d^2 = a$ is of course odd, and thus at least one of $b_0, b_1 = a - b_0$ is odd, so the second step is feasible.

In the second step, assuming that $N = 1 \mod 4$ (this is achieved by use of a multiplier, see section 2.1), we get $b = \pm 1 \mod 4 \implies b^2 = 1 \mod 4$, and so $b^2 - N = 0 \mod 4$. As $gcd(4, a) = 1$, this already yields 1.2. Moreover, we see that $0 < b < a$, which means that the resulting parabola will have its axis of symmetry between 0 and $\frac{1}{2}$.

Effective performing of step 1 is very important. Solving of a general modular quadratic equation uses Shanks-Tonelli algorithm for all prime divisors of the modulus, lifting the result by Hensel lemma to higher prime powers and finally Chinese Remainder Theorem to combine the results for distinct modulus divisors. Here $a = d^2$, so we do not need the CRT. Still, if $d$ were general, there would be need to run the Shanks-Tonelli algorithm. Of course, $N$ must be a quadratic residue mod $d$.

Such operation must be performed at each polynomial change, which happens several times per second. The Shanks-Tonelli algorithm is quite computationally heavy: its main loop contains modular exponentiation with the desired prime as the modulus, and number of iteration of the main loop is proportional to the binary length of the prime as well. In our case, the prime is quite large, with size approximately one quarter of size of $N$. So, the aim is to avoid Shanks-Tonelli entirely, if possible.

This can be achieved by further restriction on $d$ (this idea is due to Silverman, see [24]): in addition to size requirements, we furthermore assume that $d = 3 \mod 4$. In this situation, we can cut down the computational expenses for Shanks-Tonelli to just one modular exponentiation, since then

$$h_0 = N^{\frac{d+1}{4}} \mod d$$

is the required solution of $x^2 = N \mod d$. Indeed ($\mod d$),

$$h_0^2 = N^{\frac{d+1}{2}} = N \cdot N^{\frac{d-1}{2}} = N \cdot \left(\frac{N}{d}\right) = N.$$

Desired solution for $d^2 = a$ is then easily gained by use of Hensel lemma, which requires one more modular inversion modulo $d$. So we have reduced the need to perform modular inversions from several to two, at cost of shrinking the eligible set of primes $d$ to approximately a half.

When estimating the runtime, we must not forget the fact that at first we have to *generate* the candidate prime $d$ itself. This is usually done by Miller-Rabin or similar tests, which themselves perform many modular exponentiations with large exponent.

### 2.3.4 SIQS principle

From the reason that we have seen in section 2.2, changing of $a$ with every polynomial change is not optimal. If we were able to generate multiple $b$'s satisfying 1.2 for a single $a$ value, our situation would be much better - instead of calculating modular inverses in every polynomial change, we would calculate and store them just once, and for all the subsequent $b$'s the root calculation would turn into several integer operations.

This is infeasible when $a$ is a square of another prime, but it can be done if $a$ is a product

$$a = \prod_{i=1}^{s} q_i$$

of odd primes satisfying $\left(\frac{N}{q_i}\right) = 1$. Then, according to the Chinese Remainder Theorem, the equation $x^2 = N \mod a$ has $2^s$ solutions, which yields much more candidates for $b$ than the approach from subsection 2.3.3.

This approach to $a$ generation is known as the SIQS, and was proposed independently in [2] and [19]. Three major questions arise:

1. How to determine suitable $a$ divisors for optimal value of $a$?
2. How to generate the $b$'s fast enough?
3. Which of the $b$'s generated are suitable for use in a new polynomial?

The latter two questions can be solved without specific knowledge about the answer to the first one.

### 2.3.5 Desirable properties of $b$

As far as the suitability of $b$ values is concerned, we must take into consideration the following facts:

- value of $b$ must be odd, in order to satisfy 1.2,
- if $b_1, b_2$ are two values such that $b_1 = \pm b_2 \mod a$, then only one of them should be used.

The second fact is needed because $Q(x) = (2ax + b)^2 - N$ and $R(x) = (2ax - b)^2 - N$ are mutually symmetric with axis of symmetry $x = 0$. Thus they would generate the same relations twice. This is unwanted: two identical relations, when combined together, yield a trivial equality of type 1.1. If there were identical relations in our big set of relations, the number of trivial equalities of type 1.1 would thus increase and our chance to actually factor the number would get low.

From the Chinese Remainder Theorem we see that if $q_1, \dots, q_s$ are pairwise distinct divisors of $a$, and if $\pm B_1, \pm B_2, \dots, \pm B_s$ are integers such that

$$B_i^2 = \delta_{ij} N \mod q_j,$$

then any linear combination

$$b = e_1 B_1 + e_2 B_2 + \dots + e_s B_s,\ e_i \in \{-1, 1\},$$

yields an integer such that $b^2 = N \mod a$. There is $2^s$ such combinations, which are precisely all the candidates for $b$ values.

By fixing the value of $e_s$ as 1, we reduce our set to $2^{s-1}$ combinations. One easily sees that no two distinct combinations $b_i, b_j$ in this reduced set satisfy $b_i = \pm b_j \mod a$. On the other side, the combinations in this set

- do not generally satisfy $|b| < |2a|$,
- and are not all odd.

The first problem is not fatal. As we will see further, values $B_i$ themselves satisfy $|B_i| < |a|$, and so at worst $|b| < s \cdot |a|$, which is not too bad; it means shift of the parabola such that the axis of symmetry lies between 0 and $s$, which is tolerable (a typical value of $s$ is from 8 to 12, while a typical length of the sieving interval exceeds 100000).

The second problem must be solved, for otherwise the requirement 1.2 would not be satisfied. However, if $b$ is even and $a$ is a product of odd primes, then $b' = a + b$ is odd and still satisfies other requirements for $b$; this leads to another shift of axis of symmetry, at worst $|b| < (s+1) \cdot |a|$.

Taking the above calculations into account, we see that in order to get all possible $2^{s-1}$ combinations, we have at first to generate all $B_i$'s, and then walk through all $(s-1)$-tuples $(e_1, \ldots, e_{s-1}) \in \{-1, 1\}^{s-1}$. We would also like to cycle through these $(s-1)$-tuples in such way, that difference between successors would be small - preferrably, an addition or subtraction of one number only. From this, recalculation of the polynomial's roots would benefit.

### 2.3.6 SIQS - Generation of the $B_i$'s

In order to exploit the Gray code algorithm described in subsection 2.3.7, we must be able to generate the corresponding values of $B_i$, which should satisfy the following:

$$B_i^2 = N \mod q_i, \ q_i \,|\, a,$$
$$B_i^2 = 0 \mod q_j, \ q_j \,|\, a, \ j \neq i.$$

Then we have

$$(\pm B_1 \pm B_2 \cdots \pm B_s)^2 = N \mod a,$$

since all the intermediate product terms $B_k B_l$ are divisible by all primes dividing $a$, and thus by $a$ itself.

A fast method for doing this can be derived from the Chinese Remainder Theorem. Let us choose a $q_k$ dividing $a$ and solve $x^2 = N \mod q_k$ (for solving, we may use either Shanks-Tonelli algorithm, or, in case of $q_k = 3 \mod 4$, a simpler method described in 2.3.4). Let this solution be $t_k$; now, let us calculate $a_k = \frac{a}{q_k}$; this number is relatively prime to $q_k$ and thus may be inverted modularly with respect to $q_k$. Now, the product

$$B_k = t_k \cdot a_k \cdot (a_k)^{-1}$$

(the inversion with respect to $q_k$) is equal to $t_k$ mod $q_k$, and to zero mod all other divisors of $a$. Its square mod $q_i$ is, of course, either $N$ or 0, depending on whether $i = k$.

This calculation has to be performed at every change of $a$ and includes several multiple-precision operations (namely computation of $a_k$ and modular reduction of $a_k$ with respect to $q_k$, preceding the inversion) in amount proportional to number of divisors of $a$; still, this takes place at every $2^{s-1}$-th polynomial, which is, for larger values of $s$, rare enough not to slow down the entire program run.

### 2.3.7 Generation of $b$ with Gray code formulas

Having calculated all of the $B_i$'s, we generate all $b$'s with use of the following theorem, which was first stated in [2], and which is derived from the theory of Gray codes.

**Theorem 2.3.1.** *All the $b$'s are generated by the following inductive formula:*

$$b_1 = \sum_{i=1}^{s} B_i$$

$$b_{i+1} = b_i + 2(-1)^{\lceil i/2^\nu \rceil} B_\nu, \ i = 1 \ldots 2^{s-1} - 1$$

*where $2^\nu$ is the maximal power of 2 dividing $2i$.*

*Proof.* To prove correctness of the formulas, we must show that in $2^{s-1}$ iterations we really get $2^{s-1}$ distinct combinations. This is done by the following induction.

For $s = 2$, the situation is trivial: $b_1 = B_1 + B_2$ and $b_2 = -B_1 + B_2$.

For the second step, let us assume that the proposition is valid for $s$ and we want to extend it to $s + 1$.

At first, realize that in the $s$-case, we got combinations $\pm B_1 + \pm B_2 + \cdots + \pm B_{s-1} + B_s$. But that is very similar to the $s+1$ case - in the $s+1$ case, in the first $2^{s-1}+1$ iterations we get combinations $\pm B_1 + \pm B_2 + \cdots + \pm B_{s-1} +$

$B_s + B_{s+1}$. That is because sign of $B_s$ changes if and only if $2^s$ divides $2i$, and this does not happen before $i = 2^{s-1}$. So we may say that induction assumption gives us 'good behaviour' of the first $2^{s-1} - 1$ iterations.

It is easy to see that on the $2^{s-1}-$th iteration, the sign of $B_s$ changes, and it stays for the rest of the iterations. Now, we proclaim that the remaining $2^{s-1} - 1$ iterations cycle through the same combination set as the first $2^{s-1} - 1$ iterations, only in reverse order, and of course with $B_s$ sign negative. So, our aim is to show, that signs of $B_j$ in $b_{2^s-i}$ are the same as signs of $B_j$ in $b_i$ for $i = 1, \dots, 2^{s-1} - 1$, $j = 1, \dots, s-2$.

For this, let us use induction again. We know that signs of $B_j$, $j = 1, \dots, s-2$, are the same in $b_{2^{s-1}-1}$ and $b_{2^{s-1}}$. Now, it suffices to show that the change of sign in calculation of $b_{2^s-i+1}$ takes place at the same $B_\nu$ that change of sign in calculation of $b_{i+1}$, and that these changes are opposite.

To show this, we must at first realize that these two changes really do have common value of $\nu$: but this is easy. $2^\nu$ is the maximal power of 2 dividing $2i$ if and only if it is the maximal power of 2 dividing $2(2^s - i)$.

As for the opposition of the changes: sign of the change is determined by oddity or evenness $\lceil i/2^\nu \rceil$. But $i/2^\nu$ is nothing than a right shift of $i$ by $\nu$ bits: and so the sign of the change depends on the value of $\nu + 1$-th bit of $i$ from the right. So, our task is to show that $2^s - i$ and $i$ have different bits at the $\nu + 1$-th position. To show this, realize at first that $2^\nu$ divides maximally both $2i$ and $2(2^s - i)$, which means that $\nu-$th bits of both $i$ and $2^s - i$ are equal to 1. The sum $i + (2^s - i) = 2^s$ has all bits zero except the leading one; but we have just shown that in summation of these two numbers, a carry occurs on the $\nu$-th bit. This means that sum of the $\nu + 1$-th bits must be 1 to yield 0 with the carry; and this precisely means that the $\nu + 1$-th bits in $i$ and $2^s - i$ are different. So the changes of sign are opposite and the proof is finished. $\square$

It is easy to see that this method of $b$'s generation is very advantageous from the solution point of view. Say that we have solutions of a preceding polynomial

$$x_{p_{b_i}} = \frac{-b_i + r_p}{2a} \mod p, \; x'_{p_{b_i}} = \frac{-b_i - r_p}{2a} \mod p.$$

Now, let $b_{i+1} = b_i + 2(-1)^k B_\nu$. Then recalculation of the solutions is as easy as multiplying

$$s_\nu = 2B_\nu \times (2a)^{-1} \mod p$$

and summing

$$x_{p_{b_{i+1}}} = x_{p_{b_i}} - (-1)^k s_\nu,\ x'_{p_{b_{i+1}}} = x'_{p_{b_i}} - (-1)^k s_\nu,$$

again modulo $p$. These operations do not include multiple precision computation, because, though $2B_\nu$ are rather large (of the same order of magnitude as $a$), we can precompute and store their values mod all primes from the factor base, and these are ordinary integers. The same applies to modular inverses $(2a)^{-1}$.

However, in practical PC programming, there is a small problem with some OSes: if two 32-bit numbers like $2B_\nu \mod p$ and $(2a)^{-1} \mod p$ are multiplied together mod $p$, the intermediate result may overflow 32 bits (unless we limit ourselves to $p < 2^{16}$, which is too restricting for most 'real' factorizations). So, an intermediate ordinal data type with 64 bits of size is needed; such data type is provided in both Linux and Win32 platforms for $C$ language, but under different names, and such code raises portability issues.

### 2.3.8 SIQS - General remarks on $a$ determination

The sieving phase of MPQS/SIQS is ideal for massive paralellization; each machine may sieve on some given interval with dedicated polynomials, and then send the gathered relations to the central node for processing. In this, it is strongly desirable that no two distinct machines sieve with the same polynomial; otherwise the generated relations would be duplicate, and one machine's work thus useless.

As the polynomials change many times a second, it is better to generate suitable polynomials on each machine itself, instead of having them precalculated by some dedicated node and sent them to the appropriate computer altogether with other synchronization data.

In standard SIQS, this problem is soluble by leaving a special interval in the factor base (say from $M_1 = 35$-th to $M_2 = 100$th prime) for so-called "machine-specific divisors". Each of the nodes in the sieving network obtains a unique number and from this number deduces a unique subset of indices between $M_1$ and $M_2$. Primes on these indices will be chosen among divisors of $a$ for all polynomials generated by this machine. Uniqueness of the triple guarantees that no two machines will sieve on the same polynomial. The values of $M_2$ and $M_1$ can be adjusted with regard to the expected number of sieving computers. It is reasonable to have $2 - 4$ machine specific divisors; we will denote their number as $m_d$.

Having determined the machine-specific divisors, the task is to determine a good value of $s$.

The limitation from the downside is given by the fact that having too small numbers dividing $a$ leads to decrease of number of relations: one must realize that by putting $p$ into the set of divisors of $a$, the original quadratic equation $Q(x)$ reduces to a linear equation. So, instead of two series of numbers divisible by $p$:

$$\dots, x_p - 2p, x_p - p, x_p, x_p + p, x_p + 2p, \dots$$

$$\dots, x'_p - 2p, x'_p - p, x'_p, x'_p + p, x'_p + 2p, \dots$$

we have just one. This, of course, means smaller logarithm sums gathered during sieving.

The limitation on the upper side is given by two facts. First of all, it is desirable to have bigger $s$, because this reduces the frequency of "heavy" polynomial changes (those with recomputation of $a$). Second, it is desirable to approach the ideal value of $a$ as close as possible, and this is hard to do with too big divisors.

Having determined the $m_d$ machine-specific divisors and a suitable value of $s$, we must determine the range, from which to choose the remaining $a$ divisors. This is usually done by the following method: at first, divide the ideal value of $a$ by all the machine-specific divisors, and then take $s - m_d$-th root of the quotient. Its value $p_{id}$ will be close to an "ideal" divisor of $a$; by choosing an appropriately long interval within the factor base, with centre in $\lfloor p_{id} \rfloor$, we have a set of primes suitable for being divisors of $a$. From this set, actual $a$ divisors are chosen.

Currently, $s$ is usually chosen between 8 and 12 for factorization of 70− to 90−dec numbers, with satisfying results. Question of optimal $s$ value does not seem to be crucial in the whole sieving process.

### 2.3.9 SIQS - The bit method for $a$ coefficient

This is a relatively easy method for choosing divisors of $a$, suitable for factorization of smaller numbers (say, 60 decimal digits) on a PC. At the beginning, 32 primes from the determined "divisor interval" are chosen and they are given indices from 0 to 31. Then, binary numbers in range $0 \dots 2^{32} - 1$ (this is the range of an `unsigned int` type in most current PCs) are generated, and if a number of binary weight equal to $s - m_d$ is found, the corresponding $s - m_d$-tuple of the primes is chosen.

This approach has very modest system requirements - basically, only RAM for 32 primes is needed, and the computations of binary weights are very cheap. On the other hand, the generated products of primes are biased - at the beginning towards zero and at the end towards infinity. This can be remedied a little by starting the iterations not from zero, but from $2^{31}$. Then, every generated $s - m_d$-tuple is guaranteed to have at least one bigger divisor, which reduces the bias towards zero at the beginning.

A good use of this method is in distributed sieving, if current instance of sieve runs on a machine with limited hardware capabilities. Low memory and processor usage are a great benefit in such situation.

This method, with several minor changes, is used in computer algebra package LiDIA (see [15]).

### 2.3.10 SIQS - The Carrier-Wagstaff method for $a$ coefficient

Method developed by Carrier and Wagstaff is much more sophisticated than previous one, at a cost of greater hardware requirements.

A determined set of divisor candidates is split into two disjoint subsets - one consists of primes with odd indices in factor base, the other of primes with even indices.

Of all the primes with even indices, all possible triples are formed, and a table is constructed. This table contains two entries per triple: the first entry is list of primes which build this triple, and the second is logarithm of the product of these primes. It is advantageous to sort this table with respect to the value of the logarithm, because this reduces the search time greatly, and the table will be searched through at every change of $a$. The generation process takes both time and memory. If there are 100 primes with even indices, all triples are $\binom{100}{3}$ - altogether 161700, which means (with 16 bytes of memory per table row) more than 2 megabytes of RAM - an amount easily tolerable by PCs, but usually unavailable in smaller (handheld etc.) devices.

Of all the primes with odd indices, all possible $s - m_d - 3$-tuples are constructed. Authors of the method propose using `NEXKSB` algorithm for this purpose (for description of the `NEXKSB` algorithm, see [18]); it is fast and does not require much memory (surely not over 100 bytes). After each $s - m_d - 3$-tuple is generated, its product and the corresponding logarithm are calculated. Then, both the logarithms of the machine-specific divisor product and of the current $s - m_d - 3$-tuple product are subtracted from the logarithm of ideal value of $a$, and the table of "even triples" is searched for the closest possible value. This triple is then chosen to complete the set of divisors of $a$, yielding a value very close to the optimum.

As the `NEXKSB` algorithm generates unique subset at each iteration, unicity of $a$ is guaranteed. Subsets in `NEXKSB` are generated in lexicographical order, and so there is an inevitable bias: we see that first subsets will have smaller corresponding product then subsets generated later. It is possible that several millions of the first combinations will not have a matching triple in the set of even-indexed triples, because products of those first combinations are too small to be outweighted by any triple from the given interval. Of course, iterating through these unusable combinations is a loss of time.

This problem can be removed. There is no need to pick the odd-indexed and even-indexed primes from the same interval; the only thing we have to satisfy is the oddity and evenness of indices, which ensures disjointness of the two sets. So, we may at first generate the set of all even-indexed triples in some interval, and then shift the interval of odd-indexed primes upwards, until the first generated $s - m_d - 3$-tuple has a product large enough; there, we stop the shift, because we know that if already the first $s - k - 3$-tuple has a complement in set of even-indexed triples, the same must be true for the following $s - m_d - 3$-tuples, whose products are greater.

There is a dual problem, where even the last $s - k - 3$-tuple is too small to have a complement in set of even-indexed primes, and it is analogously soluble by shifting the interval of odd-indexed primes downwards.

With reasonable choice of $s$, these two problems should never occur at the same time.

Carrier-Wagstaff method is well usable if $s - m_d - 3 \geq 3$, but for lower values it cannot be used effectively. Having 3 divisors from even-indexed triples does not leave much space for the odd-indexed combinations. As values of $s \leq 8$ are usually used in factorization of smaller numbers, a viable idea is to get rid of "machine-specific divisors" entirely - for factorization of 60-70 dec numbers, one contemporary PC more than suffices, and so there is no need for parallelization. Another idea is to reduce triples to couples.

## 2.4 Combination of the relations, partial relations and linear algebra

The problem of finding combinations of relations yielding 1.3-like combinations can be transformed into a linear algebraic problem in the following way.

Let us form a set of vectors, each of the vectors having one coordinate for each member of the factor base. For each relation, we will have one vector. This vector will consist of zeros, except for the locations, where a prime number is, that divides the relation with odd exponent. In these coordinates we put a 1.

The aim now is to find a combination (or, preferrably, more combinations at a time) of the vectors that yields zero mod 2. This precisely means that the combination has all divisors with even exponent. Finding of the combination is now equivalent to a search for a solution of a system of linear equations, with operations performed modulo 2; this is a linear algebraic problem. Solution exists whenever number of vectors exceeds number of their coordinates; to be sure, we rather generate a little more (say, 1%) relations than this limit.

The rows in matrix representing the linear system mod 2 are very sparse, having only several 1's in thousands of 0's. Moreover, this matrix is not "balanced" - at the edge corresponding to smaller primes, nonzero elements arise quite often, while at the edge corresponding to large primes, nonzero elements are extremely rare. So the matrix consists of heavy columns on the left, gradually getting more sparse, and very sparse columns on the right.

From the runtime point of view, there is a great disproportion between gathering only smooth relations and gathering also partials. If only smooth values are gathered, the probability of their occurence remains roughly constant. The generated polynomials are (or at least should be) of roughly the same mathematical characteristics, which, most of all, influence the incidence of relations. On the other hand, though the partial relations also emerge with constant frequency, their contribution to the total relation pool is nonconstant. In the first moments of sieving, when only a few partial relations have been collected, the percentage of them having a common variation is negligible - typically under 1% of the total number of partials collected so far. In the last moment, the set of gathered partial relations is large, with much higher probability for birthday paradox phenomenon, which is reflected in the fact that from the collected partial relations, as much as 30%-40% actually combine into usable relations.

## 2.5 Linear algebra step

Bit matrix resulting from the sieving process is usually large, having more than 5000 rows and columns. In case of large numbers (exceeding 100 dec), it may have well over 100 000 rows and columns; this is too big to be solved by ordinary Gaussian elimination. Moreover, Gaussian elimination does not take any advantage of sparsity of the matrix.

The answer is the block Lanczos method, an algorithm designed with a single purpose - to solve sparse binary linear systems efficiently. Block Lanczos method is an iterative algorithm, whose output is a block of linear dependencies. Block nature of the algorithm (large matrices are multiplied in blocks) enables parallelization of the process, though not as easily as the sieving process itself.

A detailed description of the algorithm can be found in [16]. From our point of view, it has two important features:

1. The runtime is indirectly proportional to density of the matrix. That is, a very sparse matrix will be solved in shorter time than a matrix not-so-sparse.

2. Block Lanczos is only able to solve symmetric linear systems. Matrix $B$ resulting from the sieving step is far from symmetric.

The first feature should manifest itself in factorizations using, vs. not using, partial relations. The difference cannot however be measured directly, because removal of singletons (see later) behaves very differently in those two situations. Especially, number of singletons and zero-occurrence primes is much lower when using partial relations, which means that the resulting matrix is not only denser, but also larger.

The second feature is very limiting and must be circumvent. Approach proposed by Montgomery (see [16]) is following: take $B$ as the bit matrix generated by sieving process, transpose it and form a symmetric matrix $A = B^T B$. Now, any solution of $Bx = 0$ automatically satisfies $Ax = 0$; this is not true in the opposite direction. For more problems in this, see section 2.6.

A practical implementation issue is how to represent matrix $A$. In normal conditions, product of two sparse matrices should be relatively sparse as well, because average incidence of nonzeros is small and chance, that in two randomly chosen vectors of low weight, at least one position of nonzero will be common, is small. So, theoretically, for a product of two sparse matrices, averagely no more space than for the factors itself should be needed. Unluckily, this is not the case of the sieving matrix and its transpose. Distribution of nonzeros in the sieving matrix is very uneven, with heavy columns on the left, and so the probability that within two randomly chosen vectors at least one of their nonzeros will be common, is much higher than in average case. This was shown in practice, where, though the initial sieving matrix $B$ had density between $0,01 - 0,0001\%$, the density of the symmetric matrix $A$ usually exceeded 40%, with a tendency to settle around 45%. This, at least, was the case for relatively small sieving matrices (less than 10000x10000 in size). For larger matrices, the measuring was complicated by the fact that big nonsparse matrices take up too much memory.

## 2.6 The Singleton Gap

One of the strangest properties of the sieving matrix was discovered by the author in practical testing. This property hindered further development of the implementation for several weeks, until it was avoided by improving the post-sieving and pre-algebra step, which sorts and preprocesses the gathered relations.

As it was already mentioned in the previous paragraph, the block Lanczos method accepts only symmetric matrix as an input. Montgomery's method of generating $A = B^T B$ had success in worldwide implementations. Obviously, no one of the developers had a serious problem with the fact that $rank(A) \leq rank(B)$; mostly because the ranks usually turned out the same.

This was not the case of the author's implementation. In this implementation, the gap between $rank(A)$ and $rank(B)$ started as relatively narrow, and slowly widened (with respect to total size of $B$), reaching over 32, when $B$ had around 5000 rows. Seriousness of the problem was imminent; block Lanczos generates in normal PC 32 random dependences from the zero space of matrix $A$; now, chance that a randomly chosen dependence of matrix $A$ also belongs to the zero space of $B$, is in case of matrices over $\mathbb{Z}_2$ proportional to $1/2^{rank(A)-rank(B)}$. Thus, with the gap exceeding 32, almost none or none of the generated dependences were usable for actual factorization.

Peter Montgomery observed a similar problem, though not to such extent.

In his implementation, a smooth relation, or a combination of partial relations with all exponents even sometimes occurred. This, of course, yielded a zero row in the sieving matrix. On the first look, one should be happy about occurrence of such row, because it immediately gives us an instance of 1.1. However, these instances almost never give anything more than trivial divisors. This is definitely true in case, when such zero row originates from a suitable smooth relation, because a single smooth relation cannot give us any further information on $N$. Surprisingly, this is usually also case of such a row originating from two partial relations, where this is not an automatic law.

Situation as the one described above may also occur, when two rows of the matrix are equal. This does not necessarily mean their total equality, because all the exponents are taken modulo 2; however, this means that by summing the rows together, we get an instance of 1.1. This is essentially (at least from the rank-related point of view) equivalent to a situation with one row being the same as before and the other being zero.

Rows identical mod 2 occur more often than entirely zero rows, and must be removed before start of linear algebra processing. However, detection of a duplicite pair is not so trivial. A viable method is the following: while reading the relations (smooth, partial or combined), a hashcode of the binary vector is calculated and inserted into a binary tree of hashcodes together with number of the relation. If some hashcode is detected again, the two rows are identical mod 2; their product is calculated, which gives an instance of 1.1, and this instance is tested for triviality. So far, none of such combinations in any of the test runs has been nontrivial (which would mean an instant factorization of $N$). In the trivial case, one of the relations is accepted and the other discarded.

Frequence of occurrence of duplicite rows is very low, getting lower with expansion of the factor base (which is natural, as longer vectors have lower probability of being equal). The frequency should be greater if no *-partials are used; then, an overall average binary weight of relations is lower. If reality, 0 or 1 duplicities occur in the whole sieving; the only situation, when number of duplicities is significant, is in SIQS, when two polynomials with almost the same (varying only imn 1 prime divisor) coefficient $a$ are used. This can be avoided easily.

Nevertheless, neither zero vectors nor identical vectors were at root of the rank gap problem. When revealed, the reason of the rank gap problem turned out to be in singletons.

By a *singleton* we mean such a prime from the factor base, which occurs to an odd exponent in only one of the collected relations. This prime manifests itself in the sieving matrix as a column with precisely one nonzero bit. There are two obvious effects of singletons on the sieving matrix:

- Occurrence of a singleton raises the rank of the sieving matrix by 1, because the corresponding line is linearly independent of the others mod 2.
- A relation with a singleton cannot be used to build any dependencies, because any nontrivial combination of this relation has a 1 on the singleton's position.

Removal of singletons looks useful, because it reduces dimensions of the sieving matrix $B$. With every removed relation containing a singleton, we may at the same time expunge the corresponding prime from the factor base. This leads to shrinking the sieving matrix in both dimensions, and such action is desirable - like any other linear algebra method, block Lanczos also runs more swiftly on smaller matrices.

Process of removal of singletons must be iterative. If we remove a singleton-bearing relation from set of the collected relations, we might have created another singleton - from a prime, who had only two occurences of odd exponent, and one of them was in the removed relation. So another iteration of singleton removal is needed, and thus further, until no other singletons are found. The numbers of singletons removed in each iteration form an interesting sequence, with roughly exponential decline (see chapter 3)

Oddly enough, the lack of singleton removal was at the root of the rank gap problem. Singleton removal was originally omitted from the relation processing stage, with intention to rise the overall rank of $B$ as much as possible; that was, because with higher rank of $B$, the rank of $B^T B$ might be closer to the rank of $B$ itself. After inclusion of the singleton removal process, the Lanczos method started working, and since this time, the gap of ranks has never exhibited itself again.

The reasons for this behaviour do not lie in the singletons themselves, but rather in removal of the unusable primes from the factor base. Remember that not only singleton primes, but also primes with no occurrence at all may be safely removed from the FB; and these form a substantial number of the primes removed at the first iteration of the singleton removal process. See the results in chapter 3.

Their removal means omission of many zero columns from the matrix $B$, which leads to a dramatic increase of the overall rank. If, for instance, a factor base has $m$ elements, and we have generated 5% extra relations, we have a matrix of dimensions $m \times 1,05m$. If 10% of the primes of the factor base never occur with an odd exponent, the matrix is instantly reduced to $0,9m \times 1,05m$, with the rank being the same; so, the new matrix is significantly closer to a full-rank matrix than the old one at no real cost.

The effect of singleton removal is clearly illustrated by the measurings in subsection 3.4.

# Chapter 3

# Experimental Results

## 3.1 Sieving speed - dependence on FB size

In the next two tables, $B$ means the upper limit of the factor base, $|FB|$ is the size of factor base itself, and columns SIQS and MPQS contain times of collecting the relations, according to the sieving method. All times are given in $mm : ss$ notation. The sieving has been done with use of 1-partials.

Measuring was done with author's implementation of MPQS and SIQS, Mobile Athlon XP 2000+, Mandrake Linux 10.0. The process consumed 98-99% of the processor time.

It is clear that the optimal size of the factor base is not very sharply determined.

It can be seen that the difference between MPQS ans SIQS is rather low in case of 50-digit number and higher in case of 62-digit number. This can be explained easily.

In case of a 50-digit number, candidates for smooth and 1-partial values occur with high frequency, and for each polynomial $Q_j(x)$, 20-50 values must be trial-factored (of which, at least half is actually suitable). This factoring involves many multiple-precision operations and, in terms of runtime, dominates the whole process. With such frequency of relation occurence, the sieving ends after several hundred polynomial switches. The cost of polynomial switching represents only a moderate per cent of the total sieving runtime.

In case of a 62-digit number, candidates are much more sparser and the process of factoring shrinks down to 10-12% of the overall runtime. This lets the difference in costs of polynomial switching become more visible. The difference observed is of factor 3, which exceeds the difference described by Contini ([7]), whose value was 2. The probable reason is in Carrier-Wagstaff method of generating the $a$ coefficient - it is very effective in generating $Q_j(x)'$s close to the theoretic optimum. A significant speed gain due to this method has been already observed by the authors of this method in their paper [6].

| $B$ | $\|FB\|$ | SIQS | MPQS |
|---|---|---|---|
| 10000 | 628 | 0:12 | 0:20 |
| 15000 | 898 | 0:08 | 0:14 |
| 20000 | 1171 | 0:07 | 0:12 |
| 25000 | 1422 | 0:07 | 0:11 |
| 30000 | 1650 | 0:07 | 0:11 |
| 35000 | 1893 | 0:07 | 0:11 |
| 40000 | 2127 | 0:08 | 0:11 |
| 45000 | 2362 | 0:08 | 0:12 |
| 50000 | 2596 | 0:09 | 0:11 |
| 55000 | 2818 | 0:09 | 0:13 |
| 60000 | 3040 | 0:10 | 0:14 |
| 65000 | 3262 | 0:11 | 0:14 |
| 70000 | 3472 | 0:13 | 0:16 |
| 75000 | 3717 | 0:13 | 0:17 |
| 80000 | 3942 | 0:14 | 0:18 |
| 85000 | 4153 | 0:16 | 0:19 |
| 90000 | 4372 | 0:17 | 0:20 |
| 95000 | 4588 | 0:17 | 0:21 |
| 100000 | 4809 | 0:19 | 0:22 |

50dec digit - table of dependence
of sieving time upon factor base size.
Sieving interval: $[-80000, 80000]$.

| $B$ | \|FB\| | SIQS | MPQS |
|---|---|---|---|
| 40000 | 2115 | 1:51 | 5:22 |
| 50000 | 2577 | 1:34 | 4:43 |
| 60000 | 3038 | 1:29 | 4:22 |
| 70000 | 3486 | 1:19 | 3:57 |
| 80000 | 3945 | 1:16 | 3:53 |
| 90000 | 4414 | 1:13 | 3:50 |
| 100000 | 4870 | 1:14 | 3:49 |
| 110000 | 5297 | 1:15 | 3:52 |
| 120000 | 5728 | 1:15 | 3:55 |
| 130000 | 6172 | 1:17 | 3:57 |
| 140000 | 6603 | 1:16 | 3:51 |
| 150000 | 7014 | 1:19 | 3:55 |
| 160000 | 7434 | 1:23 | 4:00 |
| 170000 | 7823 | 1:27 | 4:05 |
| 180000 | 8269 | 1:28 | 4:10 |
| 190000 | 8691 | 1:30 | 4:13 |
| 200000 | 9093 | 1:34 | 4:20 |
| 210000 | 9482 | 1:37 | 4:24 |
| 220000 | 9872 | 1:40 | 4:28 |
| 230000 | 10292 | 1:45 | 4:37 |
| 240000 | 10679 | 1:48 | 4:41 |
| 250000 | 11082 | 1:52 | 4:49 |

62dec digit - table of dependence
of sieving time upon factor base size.
Sieving interval: $[-80000, 80000]$.

## 3.2 Sieving speed - dependence on usage of 1-partials

This table shows the effect of using vs. not using 1-partials on the runtime of the sieving process.

The sieving has been done in the same conditions as in previous section. There was difference in the "sieving threshold" - the value, at which we consider a value in interval $[-M, M]$ promising enough to be trial-factorized. If 1-partials are to be collected, we must reduce the threshold by logarithm of the maximal acceptable large prime divisor.

Testing has been done on 6 numbers having 50 to 65 decimal digits, with two factor bases - one of them limited by $B = 50000$, the other by $B = 120000$. Abbreviations lpv. and sm. mean "large prime variation" and "smooth only", respectively. Time is given in $mm : ss$ format. There was no testing of the 62 and 65 dec number with the smaller factor base, because the sieving time would be probably too long.

The advantage of "smooth only" method over "large prime variation" in 50 and 53 decimal numbers and $B = 120000$ is caused by the fact that in case of such small numbers, occurence of 1-partials is enormous (several thousand per each polynomial), and the process of their trial-factoring takes majority of the sieving time. This is not reflected in case of $B = 50000$, because the limit for maximal large prime divisor is much smaller and 1-partials do not occur so often.

| Dec size | 50K - lpv. | 50K - sm. | 120K - lpv. | 120K - sm. |
|---|---|---|---|---|
| 50 | 0:09 | 0:17 | 0:24 | 0:13 |
| 53 | 0:19 | 0:49 | 0:33 | 0:30 |
| 56 | 0:41 | 3:16 | 0:44 | 1:40 |
| 59 | 2:32 | 12:49 | 1:48 | 5:16 |
| 62 | – | – | 1:27 | 7:08 |
| 65 | – | – | 6:12 | 23:49 |

The effect of large prime variation on the sieving time. Sieving interval: $[-80000, 80000]$.

## 3.3 Singletons - dependence on $\log(N)$ and FB size

This table of measurings shows the dependence of number of removed singletons upon three parameters. These are:

1. *Size of $N$ in digits*, where growth of $N$ should result in decrease of number of singletons. That is because for a longer $N$, the factorized values $Q_j(x)$ are longer, include more divisors - and more divisors mean decrease of chance of a prime to become a singleton.

2. *Usage of 1-partials*, where usage should result in slight decrease of number of singletons. That is because combination of two 1-partials has more divisors than an ordinary smooth relation, and more divisors mean decrease of chance of a prime to become a singleton.

3. *Size of the FB*, where growth of $B$ should result in increase of number of singletons. That is because the large primes around $2 \cdot 10^5$ seldom fit into the sieving interval, and are found very infrequently in relations.

The fourth obvious parameter, length of the sieving interval, was constant (80000 for all numbers, except for 71dec, where it is 120000). Prolonging of the interval means that more large primes will have chance to get into factorization of some $Q_j(x)$ value, and thus reduce the number of singletons removed.

Measuring was done with author's implementation of SIQS, Desktop Athlon XP 1600+, Mandrake Linux 10.0.

| | |
|---|---|
| $B$ | Upper bound of the factor base ($K$ =thousand). |
| $\|FB\|$ | Number of elements of the factor base. |
| Rel. | Total number of relations used in Lanczos method. |
| x+y | (In relations) smooth rels vs. rels gained from 1-partials. |
| Zero | Number of primes with zero occur. |
| Sing. | Number of true singletons removed. |
| a+b+c+ | Number of singletons removed in each iteration. |
| Rest | Size of the FB after removal of singletons and 0-occ. primes. |
| Iter. | Number of block Lanczos iterations. |

The larger two numbers were factorized with use of large prime variation only, since run on smooth numbers would take too much time.

| $B$ | $\|FB\|$ | Rel. | Zero | Sing. | Rest | Iter. |
|---|---|---|---|---|---|---|
| $60K$ | 3004 | $3322 = 1924 + 1398$ | 55 | $103 + 25 + 3$ | 2759 | 92 |
| $75K$ | 3654 | $3971 = 2322 + 1649$ | 68 | $142 + 43 + 8 + 4 + 1 + 1 + 1 + 1$ | 3312 | 110 |
| $90K$ | 4334 | $4733 = 2822 + 1911$ | 90 | $195 + 75 + 11 + 4 + 2$ | 3857 | 128 |
| $105K$ | 4982 | $5379 = 3285 + 2094$ | 121 | $244 + 78 + 18 + 6$ | 4385 | 145 |

53 dec number with large prime variation

| $B$ | $\|FB\|$ | Rel. | Zero | Sing. | Rest | Iter. |
|---|---|---|---|---|---|---|
| $60K$ | 3004 | 3164 | 70 | $162 + 44 + 12 + 30$ | 2638 | 88 |
| $75K$ | 3654 | 3813 | 123 | $207 + 70 + 19 + 3 + 1$ | 3095 | 103 |
| $90K$ | 4334 | 4533 | 171 | $214 + 86 + 20 + 4$ | 3646 | 121 |
| $105K$ | 4982 | 5182 | 189 | $309 + 136 + 33 + 10 + 3 + 1$ | 4087 | 135 |

53 dec number with smooth relations only

| $B$ | $\|FB\|$ | Rel. | Zero | Sing. | Rest | Iter. |
|---|---|---|---|---|---|---|
| $60K$ | 3005 | $3322 = 1740 + 1582$ | 23 | $34 + 4$ | 2921 | 97 |
| $75K$ | 3665 | $3982 = 2088 + 1894$ | 28 | $79 + 7 + 2$ | 3518 | 117 |
| $90K$ | 4302 | $4699 = 2555 + 2144$ | 41 | $124 + 24 + 2 + 1 + 1$ | 4064 | 135 |
| $105K$ | 4950 | $5348 = 2959 + 2389$ | 55 | $174 + 32 + 9 + 1 + 1 + 1$ | 4627 | 153 |

59 dec number with large prime variation

| $B$ | $\|FB\|$ | Rel. | Zero | Sing. | Rest | Iter. |
|---|---|---|---|---|---|---|
| $60K$ | 3005 | 3164 | 31 | $115 + 13 + 6 + 1$ | 2804 | 93 |
| $75K$ | 3665 | 3824 | 64 | $124 + 35 + 4 + 1 + 1 + 1 + 1$ | 3365 | 112 |
| $90K$ | 4302 | 4501 | 92 | $157 + 33 + 9 + 1$ | 3910 | 130 |
| $105K$ | 4950 | 5149 | 129 | $213 + 80 + 19 + 6 + 1$ | 4362 | 145 |

59 dec number with smooth relations only

| $B$ | $\|FB\|$ | Rel. | Zero | Sing. | Rest | Iter. |
|---|---|---|---|---|---|---|
| $60K$ | 3027 | $3345 = 1787 + 1558$ | 9 | $49 + 3$ | 2955 | 98 |
| $75K$ | 3713 | $4031 = 2105 + 1926$ | 25 | $75 + 11 + 1$ | 3574 | 119 |
| $90K$ | 4379 | $4777 = 2551 + 2226$ | 37 | $78 + 11 + 1 + 1$ | 4212 | 140 |
| $105K$ | 5052 | $5530 = 2998 + 2532$ | 47 | $119 + 21 + 4 + 1$ | 4811 | 160 |

65 dec number with large prime variation

| $B$ | $\|FB\|$ | Rel. | Zero | Sing. | Rest | Iter. |
|---|---|---|---|---|---|---|
| $60K$ | 3049 | $3367 = 1975 + 1392$ | 15 | $32 + 1$ | 2985 | 99 |
| $75K$ | 3724 | $4042 = 2308 + 1734$ | 9 | $66 + 3$ | 3635 | 120 |
| $90K$ | 4390 | $4788 = 2759 + 2029$ | 14 | $87 + 23 + 1$ | 4260 | 142 |
| $105K$ | 5055 | $5533 = 3258 + 2275$ | 40 | $88 + 11 + 2$ | 4874 | 162 |

71 dec number with large prime variation

## 3.4 Properties of the sieving matrices

In this section we demonstrate the behavior of the sieving matrix $B$ (mod 2) and of the symmetric matrix $A = B^T B$ used as input for the block Lanczos algorithm, with regard to two properties.

The first table concerns measurings of the total density of matrices $B$ and $A$, and demonstrates the fact that product of two very sparse matrices need not be sparse at all. The measuring has been done for matrices up to $10^5 \times 10^5$, greater sizes being infeasible because of memory size limitations. It is clear that while the sieving matrix is very sparse, having less than 1% nonzero entries, the symmetric matrix is close to 50% of nonzero entries.

The second and third table demonstrate the singleton gap. Instances of the same factorization task are being run, once with and once without removal of the singletons and zero–occurence primes. We see that if singletons are removed, ranks of $A$ and $B$ stay similar, while without the removal they tend to diverge.

Legend:

| | |
|---|---|
| $B$ | Upper bound of the factor base ($K$ =thousand). |
| $\|FB\|$ | Number of elements of the factor base. |
| Size | Size of the sieving matrix |
| Zero | Number of primes with zero occur. removed |
| Sing. | Number of true singletons removed. |
| r(A) | Rank of matrix $B^T B$. |
| r(B) | Rank of matrix $B$. |

Measuring was done with author's implementation of SIQS, Desktop Athlon XP 1600+, Mandrake Linux 10.0. The common settings were: factorization of a 60-dec (198-bit) number, $M = 80000$, using $1-$partials.

| $\|FB\|$ | Density(A) | Density(B) |
|---|---|---|
| 4950 | 49.94% | 0.43% |
| 6695 | 49.93% | 0.38% |
| 8420 | 49.95% | 0.26% |
| 10522 | 49.99% | 0.20% |

Density of sieving matrix and its symmetrization

| $B$ | $\|FB\|$ | Size(A) | Size(B) | Zero | Sing. | r(A) | r(B) | r(B)-r(A) |
|---|---|---|---|---|---|---|---|---|
| $105K$ | 4950 | $5064 \times 5064$ | $5064 \times 5064$ | 90 | 194 | 4562 | 4562 | 0 |
| $135K$ | 6695 | $6712 \times 6712$ | $6712 \times 5995$ | 136 | 405 | 5991 | 5993 | 2 |
| $185K$ | 8420 | $8333 \times 8333$ | $8333 \times 7358$ | 230 | 575 | 7355 | 7356 | 1 |
| $245K$ | 10883 | $10522 \times 10522$ | $10522 \times 9249$ | 345 | 894 | 9247 | 9247 | 0 |

Removal of singletons - normal situation

| $B$ | $\|FB\|$ | Size(A) | Size(B) | r(A) | r(B) | r(B)-r(A) |
|---|---|---|---|---|---|---|
| $105K$ | 4950 | $5348 \times 5348$ | $5348 \times 4950$ | 4842 | 4846 | 4 |
| $135K$ | 6695 | $7253 \times 7253$ | $7253 \times 6695$ | 6519 | 6534 | 15 |
| $185K$ | 8420 | $8985 \times 8985$ | $8985 \times 8420$ | 8143 | 8161 | 18 |
| $245K$ | 10883 | $11761 \times 11761$ | $11761 \times 10883$ | 10450 | 10486 | 36 |

No singleton removal - manifestation of the singleton gap

# Bibliography

[1] M. Agrawal, N. Kayal and N. Saxena, 'PRIMES is in P', *preprint*

`http://www.cse.iitk.ac.in/news/primality_v3.pdf`

[2] W. Alford, C. Pomerance, 'Implementing the self initializing quadratic sieve on a distributed network', *Number Theoretic and Algebraic Methods in Computer Science, Proc. of Int'l Moscow Conference, June-July 1993* (1995), A. J. van der Poorten, I. Shparlinski, H. G. Zimmer, eds., World Scientific, 163–174.

Unavailable for public download.

[3] R. Baer: 'Polyminimaxgruppen', *Math. Annal.* 175 (1968), 1–43.

Unavailable for public download.

[4] H. Boender, H. J. J. te Riele, 'Factoring Integers with Large–Prime Variation of the Quadratic Sieve', *Department of Numerical Mathematics, Centrum voor Wiskunde en Informatica, Amsterdam* (1995)

Available by search at `http://citeseer.ist.psu.edu/`.

[5] N. Bourbaki, 'Algèbre commutative', Hermann, 1965

Unavailable for public download.

[6] B. Carrier, S. Wagstaff: 'Implementing the Hypercube Quadratic Sieve with Two Large Primes', *Proceedings of the International Conference on Number Theory for Secure Communications*, 2003.

`https://www.cerias.purdue.edu/tools_and_resources/bibtex_archive/archive/2001-45.pdf`

[7] S. Contini, ‘Factoring Integers with the Self-Initializing Quadratic Sieve’, *Diploma Thesis* (1996)

`http://www.crypto-world.com/documents/siqs.ps.gz`

[8] B. Dodson, 'Filtering NFS relations: weeding, cycles and clique-deletes' (lecture), *CWI Workshop on Factoring Large Numbers* (2003)

Unavailable for public download.

[9] D. Eisenbud, 'Commutative Algebra with a View Toward Algebraic Geometry', *Springer-Verlag*, New York, 1995

Unavailable for public download.

[10] P. Hall, 'On the finiteness of certain soluble groups', *Proc. London Math. Soc.* 9, (1959), 595–622

Unavailable for public download.

[11] I. Kaplansky, 'Commutative Rings', *Allyn and Bacon*, Boston, 1970

Unavailable for public download.

[12] M. Kechlibar, T. Kepka, J. Kortelainen, 'Notes on commutative semilocal rings', *(preprint)*

Unavailable for public download.

[13] T. Kepka, P. Nìmec, 'Basic properties of radical rings', *(preprint)*

Unavailable for public download.

[14] A. K. Lenstra, M. S. Manasser, 'Factoring with two large primes', *Math. Comp.*93 (1194), 785–798

Available for subscribers of `http://www.jstor.org`

[15] LiDIA - the computer algebra package

`http://www.informatik.tu-darmstadt.de/TI/LiDIA/`

[16] P. Montgomery, 'A Block Lanczos Algorithm for Finding Dependencies Over GF(2)'. *EUROCRYPT 1995*, (1994) 106–120.

`http://202.115.65.116/Cipher/HTML/PDF/E95/106.PDF`

[17] M. Nagata, 'Local Rings'. *John Wiley & Sons, Inc* (1962)

Unavailable for public download.

[18] A. Nijenhuis, H. Wilf, 'Combinatorial Algorithms for Computers and Calculators'. *Academic Press, Orlando FL, second edition* (1978)

`http://www.cs.sunysb.edu/~algorith/implement/wilf/implement.shtml`

[19] R. Peralta, 'Implementation of the Hypercube Multiple Polynomial Quadratic Sieve', *preprint.*

`http://cs-www.cs.yale.edu/homes/peralta/papers/HMPQS.ps`

[20] C. Pomerance, 'The Quadratic Sieve Factoring Algorithm', *Advances in Cryptology: Proceedings of EUROCRYPT 84* Springer–Verlag (1985), 169–182.

Unavailable for public download.

[21] R. L. Rivest, A. Shamir and L. M. Adleman, 'A Method for Obtaining Digital Signatures and Public-Key Cryptosystems', *Comm. ACM* 21 (1978), 120–126.

`http://theory.lcs.mit.edu/~cis/pubs/rivest/rsapaper.ps`

[22] P. Samuel, 'Méthodes d'algèbre abstraite en géométrie algébrique', *Springer-Verlag*, Berlin, 194 Unavailable for public download.

[23] I. Shafarevich, 'Basic Algebraic Geometry', *Springer-Verlag*, New York, 1994

Unavailable for public download.

[24] R.D. Silverman, 'The Multiple Polynomial Quadratic Sieve', *Math. Comput.* 48, (1987), 329–339.

Available for subscribers of `http://www.jstor.org`

[25] O. Zariski, P. Samuel, 'Commutative Algebra', *D. van Nostrand*, Princeton, 1958