Half - GCD and Fast Rational Recovery

Daniel Lichtblau

Wolfram Research, Inc.
100 Trade Centre Dr.
Champaign IL USA, 61820

danl@wolfram.com

ISSAC 2005, Beijing
July 2005

Abstract

Over the past few decades several variations on a "half GCD" algorithm for obtaining the pair of terms in the middle of a Euclidean sequence have been proposed. In the integer case algorithm design and proof of correctness are complicated by the effect of carries. This paper will demonstrate a variant with a relatively simple proof of correctness. We apply this to the task of rational recovery for a linear algebra solver. We also show how it is applicable to lat - tice reduction in the plane.

Background and brief history

The basic idea behind the "half GCD" (HGCD) algorithm is similar to that of many divide - and - conquer algorithms. We split our input integer into two parts of roughly half the size (based on the larger one). We do a recursive half GCD call It gives multipliers as well (similar to those obtained by extended gcd, and in fact HGCD is a very good way to do extended gcd computations). We use these to reduce not just the half size reduced inputs, but the full sized originals. We show that this reduces them by approximately 1/4 in size. Repeat ing this reduces to 1/2 the original size (this is the crude idea: the devil is in the details). We now have something that can indeed be used recursively.

Background and brief history

HGCD is based on a method for computing continued fractions presented by Schönhage around 1971. It is superlinear in bit complexity, as it is seen to be log(n) times worse than multiplication, which had been shown to be superlin - ear. It is related to...

Background and brief history

HGCD is based on a method for computing continued fractions presented by Schönhage around 1971. It is superlinear in bit complexity, as it is seen to be log(n) times worse than multiplication, which had been shown to be superlin - ear. It is related to...

A slower superlinear algorithm by Knuth that had just appeard. This in turn is related to...

Background and brief history

HGCD is based on a method for computing continued fractions presented by Schönhage around 1971. It is superlinear in bit complexity, as it is seen to be log(n) times worse than multiplication, which had been shown to be superlin - ear. It is related to...

A slower superlinear algorithm by Knuth that had just appeard. This in turn is related to...

An earlier algorithm, "nearly" subquadratic, by Lehmer (long before emer - gence of subquadratic multiplication). This in turn is related to...

Background and brief history

HGCD is based on a method for computing continued fractions presented by Schönhage around 1971. It is superlinear in bit complexity, as it is seen to be log(n) times worse than multiplication, which had been shown to be superlin - ear. It is related to...

A slower superlinear algorithm by Knuth that had just appeard. This in turn is related to...

An earlier algorithm, "nearly" subquadratic, by Lehmer (long before emer - gence of subquadratic multiplication). This in turn is related to...

A MUCH earlier algorithm, which had quadratic bit complexity.

Background and brief history

What Schönhage proposed was a method for efficient computation of contin - ued fractions. With small modification this becomes a gcd algorithm. It was presented as such by Moenck and subsequently Aho, Hopcroft, and Ullman. These worked primarily with the univariate polynomial case, and the latter also discussed modifications needed to handle the integer case.

The problem was that subtle details related to integer carries made that variant of the algorithm difficult to implement, and, indeed, difficult even to state clearly. Even the polynomial case is not trivial, and the version appearing in Aho, Hopcroft, and Ullman was a bit flawed.

Around 1990 Thull and Yap presented an algorithm with proof, but it is quite intricate in terms of following all possible cases. I believe corrections have been made from time to time; I do not know its present status but assume it is correct. The polynomial case was discussed in detail in the text by von zur Gathen and Gerhard, where it is mentioned that at that time (1999) similar meth ods could probably be made to work for integers, but the details made a clean proof problematic. This motivated a more recent variant by Pan and Wang that seems correct but is a bit short on detail (ruthless enforcement of ISSAC page limits?), hence difficult to implement.

Background and brief history (okay, it's not so brief anymore)

Around 2001 we at WRI (specifically David Terr, Mark Sofroniou, and myself) implemented a version that always worked but was only probabilisti - cally superlinear (though testing indicated it clearly had the expected bit com - plexity on large examples). More recently I wanted a version that was provably correct, not so much for the speed but because I wanted to experiment with rational recovery. This requires a "truly correct" HGCD, as opposed to one that might merely take us nearby to the middle in a remainder sequence, which, for asymptotic speed purposes, seems to be generally "good enough".

This necessitated that I figure out how correctly to do this, hence the main part of this work. Of course the speed is important, so Mark Sofroniou and I (more him than me) took this opportunity of overhaul to do some careful tuning and make improvements in various places.

Recent related work

In the past few years there has been a nice binary recursive method imple - mented by Stehlé and Zimmermann. It avoids carry issues by working from the least significant halves of the inputs, hence can be viewed as a superlinear cousin of the well known binary gcd method. This one has also been done by Mark Sofroniou. Quite recently there is work in progress by Niels Möller. It uses a method also due to Schönhage, developed for reducing binary quadratic forms. This, too, is closely related to work by Yap, and furthermore is a close relative to the method of this talk. Möller moreover implemented prior variants including the Stehlé and Zimmermann method, and reported on timing compari sons (all seem to be close).

One interesting point we will cover briefly is that one can in a sense reverse the quadratic form reduction process,. That is, HGCD can be utilized to reduce a binary quadratic form. We will in fact show how it may reduce a planar lattice.

HGCD: What it does

Given a pair of positive integers $\begin{pmatrix} m \\ n \end{pmatrix}$ with $m > n$. HGCD will return a multiplier matrix and a new pair $\begin{pmatrix} v_i \\ v_{i+1} \end{pmatrix}$ of consecutive values in the Euclidean remainder sequence for $\begin{pmatrix} m \\ n \end{pmatrix}$. These straddle the half size point in that $v_i \geq \sqrt{m} > v_{i+1}$. The matrix R_i is the product of elementary transformation matrices, and $R_i \begin{pmatrix} m \\ n \end{pmatrix} = \begin{pmatrix} v_i \\ v_{i+1} \end{pmatrix}$.

HGCD: basic ideas of the paper

- Build matrix R_i as product of "elementary" matrices $R_j = \begin{pmatrix} 0 & 1 \\ 1 & -q_j \end{pmatrix} R_{j-1}$ where the q_j are successive quotients in the remainder sequence of our input integers $\begin{pmatrix} m \\ n \end{pmatrix}$.
- Simple lemmas show how to recover R_{j-1} from R_j and describe growth bounds of various quantities encountered.
- One lemma from the literature gives a sufficient condition for obtaining a consecutive pair in the remainder sequence. The use is that we need not work with the original inputs but (as observed by Lehmer almost 70 years ago) can work with leading part. Later refinement: work recursively with top "half" of inputs.
- Formulate lemmas to bound sizes of values formed by multiplying the HGCD matrix for the top halves with the full inputs: due to carries this is not entirely trivial. With these we can bound number of fixup steps (forward or backwards Euclidean steps). This enforces complexity bound of $log(n)\,M(n)$, where $M(n)$ is the cost of multiplying a pair of $n-$bit integers.
- Formulate lemmas to "repair" a pair so that we can invoke the lemma regarding consecutive pairs in the sequence.

HGCD algorithm

Step 1: Take $k = \left\lfloor \frac{\log_2 m}{2} \right\rfloor$. Write $\begin{pmatrix} m \\ n \end{pmatrix} = \begin{pmatrix} 2^k f_0 + f_1 \\ 2^k g_0 + g_1 \end{pmatrix}$ with $\{f_1, g_1\} < 2^k$. We have $2^k > \sqrt{m} > 2^{k-1}$ and $g_0 \le f_0 < 2\sqrt{m}$.

Step 2: Recursively compute HGCD$\begin{pmatrix} f_0 \\ g_0 \end{pmatrix}$. Obtain matrix R_i and pair $\begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix}$ with $R_i \begin{pmatrix} f_0 \\ g_0 \end{pmatrix} = \begin{pmatrix} r_i \\ r_{i+1} \end{pmatrix}$ and $0 \le r_{i+1} < \sqrt{f_0} < r_i$.

Step 3: Compute $\begin{pmatrix} v_i \\ v_{i+1} \end{pmatrix} = R_i \begin{pmatrix} m \\ n \end{pmatrix}$. Lemmas (see paper) bound $\begin{pmatrix} v_i \\ v_{i+1} \end{pmatrix}$, so that we obtain cases of set 4. Some bounds are conditional on the size of r_i. Rather conveniently, it turns out that all conditions can be arranged to have the same "crossover" value $r_i = 2\sqrt{f_0}$. (What matters is that size conditions all be the same; the actual crossover is not important except as a technical detail in proofs). We also bound v_{i-1}.

Step 4: Possible cases: (i) v_{i+1} may be negative. (ii) $v_i < v_{i+1}$. If neither, that is, $v_i > v_{i+1} > 0$, then we set $\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} v_i \\ v_{i+1} \end{pmatrix}$ and move to step 5. Else...

HGCD algorithm

Case i: $v_{i+1} > v_i$. Take the matrix $H = \begin{pmatrix} 1 & 0 \\ -h & 1 \end{pmatrix}$ where $h = \lfloor v_{i+1} / v_i \rfloor \geq 1$. The new pair is $\begin{pmatrix} u \\ v \end{pmatrix} = H \begin{pmatrix} v_i \\ v_{i+1} \end{pmatrix} = \begin{pmatrix} v_i \\ v_{i+1} - h\, v_i \end{pmatrix}$ which satisfies the requirement that $u > v > 0$. Move on to step 5.

Case ii: $v_{i+1} < 0$ and $v_{i+1} + v_i \geq 0$. First assume $q_i > 1$. Then take $H = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$ and proceed as in case i above to obtain a positive pair in the correct order. Note that $u - v = \mid v_{i+1} \mid < 2^{k+1} \sqrt{f_0}$. This means that a Euclidean step will bring the pair into the range claimed in step 6 below. If $q_i = 1$, again we use the matrix H as defined above, and again we obtain a positive pair in the cor - rect order. But the product $H\, R_i$ is $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ which is not an elementary matrix. To correct, multiply by $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ again, giving as product the identity matrix. This flips $\begin{pmatrix} u \\ v \end{pmatrix}$. Proceed back to step i to correct thie ordering.

Case (iii). If $v_{i+1} < 0$ and $v_{i+1} + v_i < 0$ then either $v_i \leq 2^{k-1} \sqrt{f_0}$ or $v_{i+1} \leq -2^{k-1} \sqrt{f_0}$. In either case we have $r_i \leq 2 \sqrt{f_0}$. Perform a reversal of a Euclidean step, obtaining pair $\begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} v_{i-1} \\ v_i \end{pmatrix} = R_{i-1} \begin{pmatrix} m \\ n \end{pmatrix}$. If $u < v$ then we go to case (i) above.

HGCD algorithm

Step 5: Perform a Euclidean reduction on $\begin{pmatrix} u \\ v \end{pmatrix}$, obtaining next consecutive pair $\begin{pmatrix} v \\ w \end{pmatrix}$ in the remainder sequence for $\begin{pmatrix} m \\ n \end{pmatrix}$, with elementary transformation matrix $Q = \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix}$ ($q = \lfloor u / v \rfloor$, $w = u - q\,v$). Form corresponding transformation matrix $R = Q\,R_i$.

Step 6: Examine the values of our pair $\begin{pmatrix} v \\ w \end{pmatrix}$. We know: $0 < v < 2^k\,3\,\sqrt{f_0} < 2^{k+1/2}\,3\,m^{1/4} < 2^{3/2}\,3\,m^{3/4}$ and $u > 2^{k-1} > \sqrt{m}\,/\,4$.

Case i: $w < \sqrt{m}$. If $v \geq \sqrt{m}$ we have our pair straddling $\sqrt{m}$. Return it along with the transforming matrix R. If $v < \sqrt{m}$ we do reverse Euclidean steps, updating our remainder sequence pair and transformation matrix. We have at most five such steps before an element exceeds $\sqrt{m}$; perform as many such steps as is needed to obtain the pair straddling $\sqrt{m}$ and return it with corresponding matrix.

Case (ii). $\sqrt{m} \leq w < v < 2^{3/2}\,3\,m^{3/4}$ (typical: w and v both close to $m^{3/4}$). Similarly to step 1, we take $l = \lfloor \log_2 m \rfloor - \lfloor \log_2 v \rfloor$. Note l is roughly between one fourth and one half bit length of m (Specifically: $\lfloor \log_2 m \rfloor / 4 - 3 < l < \lfloor \log_2 m \rfloor / 2 + 3$). Proceed to step 7.

HGCD algorithm

Step 7: Write $\begin{pmatrix} v \\ w \end{pmatrix} = \begin{pmatrix} 2^l f_2 + f_3 \\ 2^l g_2 + g_3 \end{pmatrix}$ with $\lfloor \log_2 f_2 \rfloor = \lfloor \log_2 v \rfloor - l$. Can show: f_2 and g_2 are no larger than $O\left(\sqrt{m}\right)$.

As in step 2, recursively compute HGCD$\begin{pmatrix} f_2 \\ g_2 \end{pmatrix}$. As in steps 3 and 4, obtain trans formation matrix S, and consecutive pair $\begin{pmatrix} v_j \\ v_{j+1} \end{pmatrix}$ in remainder sequence for $\begin{pmatrix} m \\ n \end{pmatrix}$, with $v_j > v_{j+1} \geq 0$. If $v_j \leq 2^{l-2}\sqrt{f_2}$. If necessary, do a single reverse Euclidean step to get previous consecutive pair in the sequence. At this point we have a consecutive pair, call it $\begin{pmatrix} x \\ y \end{pmatrix}$. Theory guarantees $y < 2^{l+2}\sqrt{f_2}$ and $x > 2^{l-2}\sqrt{f_2}$.

Step 8: We know f_2 is within a factor of 2 of $2^{-l}v$, so $2^l\sqrt{f_2} \approx 2^{l/2}\sqrt{v} \approx \sqrt{\frac{m}{v}}\sqrt{v} = \sqrt{m}$ (approximation from first to last is within a factor of 2). Inequalities from step 7 imply $y < 8\sqrt{m}$ and $x > \sqrt{m}/8$. With limited number of Euclidean steps, or reversals thereof, we obtain the consecu - tive pair in the remainder sequence that straddles $\sqrt{m}$, and the transformation matrix that gives this pair.

While this is a bit intricate, it seems simpler than any other version I have seen that enforces the "straddle the half - size" condition. This in turn is important for a simple coding of rational recovery.

Example to demonstrate speed

```
fibs = {Fibonacci[10^7], Fibonacci[10^7 + 1]};
```

These have about two million digits:

```
Log[10., fibs[[1]]]
```

2.08988×10^{6}

We'll take gcd (regular and extended) to get some idea of the speed of this method. This is done on a 3 GHz machine under Linux, using a not - terribly - optimized GMP installation for the bignum arithmetic.

```
Timing[gcd = Apply[GCD, fibs];]
Timing[{gcd2, mults} = Apply[ExtendedGCD, fibs];]
mults.fibs == gcd == gcd2 == 1
```

```
{41.0518 Second, Null}
```

```
{52.0571 Second, Null}
```

```
True
```

Example to demonstrate speed

◂Previous Next▸

We contrast this to the time it takes just to multiply this pair.

```
Timing[product = Apply[Times, fibs];]
```

```
{1.36279 Second, Null}
```

Note that with 2 000 000 digit inputs, we expect the ratio between multiplica - tion and HGCD - based gcd to be, roughly, a factor of log(2 000 000) (using base of 2, since this is a divide - and - conquer algorithm).

```
41. / Log[2., 2 000 000.]
```

```
1.95876
```

This indicates that the speed is in the expected ballpark.

Rational recovery

Because HGCD brings us right to the middle pair in a remainder sequence, it is ideally suited for recovery of rationals from p - adic images. The method is explained in detail in the text by von zur Gathen and Gerhard. Given a prime power p^k and a smaller nonnegative integer x not divisible by p, we can obtain a rational a/b equivalent to x modulo p^k with both numerator and denominator smaller than the square root of the prime power. It is obtained directly from the HGCD matrix and middle pair given by HGCD$\begin{pmatrix} p^k \\ x \end{pmatrix}$.

To summarize, we have a matrix $R_j = \begin{pmatrix} s_j & t_j \\ s_{j+1} & t_{j+1} \end{pmatrix}$ with $R_j \begin{pmatrix} p^k \\ x \end{pmatrix} = \begin{pmatrix} u \\ v \end{pmatrix}$. We have $\{v, t_{j+1}\} < \sqrt{p^k}$ and $\frac{v}{t_{j+1}} \equiv_{p^k} x$ because $s_{j+1}\, p^k + t_{j+1}\, x = v$. Thus we have our desired rational.

The *Mathematica* code below will do this recovery given the input pair $\{x, p^k\}$.

```
rationalRecover[x_, pk_] :=
 ((#[[2, 2]] / #[[1, 2, 2]]) &)[
  Internal`HGCD[pk, x]]

rationalRecover[
 111 122 223 333 444 455 556 666 777 788 889 999,
 Prime[222]^17]
```

$$\frac{226\,563\,468\,288\,751\,478\,292\,482\,603}{350\,240\,101\,969\,175\,888\,689\,266\,729}$$

Rational recovery

It is not hard to code the pedestrian approach, using the usual Euclidean algorithm to work down to where we first get below half the size of the second input.

```
rationalRecover2[a_, b_] := Module[
  {mat, aa = a, bb = b, cc = 1, dd = 0, quo},
  mat = {{aa, cc}, {bb, dd}};
  While[Abs[aa] ≥ Sqrt[b],
   quo = Quotient[bb, aa];
   {{aa, cc}, {bb, dd}} =
    {{bb, dd} - quo * {aa, cc}, {aa, cc}};];
  aa / cc]
```

```
rationalRecover2[
 111 122 223 333 444 455 556 666 777 788 889 999,
 Prime[222]^17]
```

$$\frac{226\,563\,468\,288\,751\,478\,292\,482\,603}{350\,240\,101\,969\,175\,888\,689\,266\,729}$$

Yet another method involves lattice reduction.

```
rationalRecover3[n_, pq_] := (#[[1]] / #[[2]] &)[
  First[LatticeReduce[{{n, 1}, {pq, 0}}]]]
```

```
rationalRecover3[
 111 122 223 333 444 455 556 666 777 788 889 999,
 Prime[222]^17]
```

$$\frac{226\,563\,468\,288\,751\,478\,292\,482\,603}{350\,240\,101\,969\,175\,888\,689\,266\,729}$$

Speed of rational recovery on a larger example

We'll try a larger example to illustrate speed.

```
{v1, v2} = {Random[Integer, Prime[11 111]^2222],
   Prime[11 111]^2222};
```

```
ByteCount[v2]
```

```
4704
```

```
Timing[rat = rationalRecover[v1, v2];]
```

```
{0.023996 Second, Null}
```

```
Timing[rat2 = rationalRecover2[v1, v2];]
```

```
{17.3534 Second, Null}
```

```
Timing[rat3 = rationalRecover3[v1, v2];]
```

```
{210.435 Second, Null}
```

```
rat == rat2 == rat3
```

```
True
```

Where the fast version is particularly important is in doing linear algebra over the rationals. As dimension increases, so (typically) do sizes of resulting num - bers. Hence asymptotically fast recovery is quite useful in order to keep that step from being a bottleneck in the process. Other tactics may be found in work by Chen and Storjohann from this conference.

Planar lattice reduction

Suppose we have a 2x2 integral matrix $M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ where we regard the rows as generating a lattice in $\mathbb{Z}^2$. The goal is to find a reduced form, that is, a unimo dular multiplier matrix A such that $A\,M = L$ where L is the lattice reduced form of M.

We already saw an example where lattice reduction of a certain 2x2 matrix gave a result equivalent to HGCD (it can be proven that that will always work for rational recovery, given to certain size hypotheses). Here we want to go in the reverse direction, and apply HGCD - like methods to reduce a lattice. The literature on this topic contains needed theorems; we simply show the idea of a method for the purpose at hand.

Planar lattice reduction

What is illustrated below appears to be essentially the same as the method developed by Eisenbrand a few years ago.

Step 1. Put M into Hermite normal form. As is well known, this uses the extended gcd algorithm, hence (for large inputs) amounts to a few HGCD invo - cations. We obtain $M_1 = \begin{pmatrix} g & j \\ 0 & k \end{pmatrix}$ (where $g = \gcd(a, c)$) and a unimodular trans - formation matrix A_1 with $A_1 M = M_1$.

Step 2. See if this is lattice reduced. If so, we are finished. If not, we now have a "small" element in the upper left and a zero beneath it. Thus it makes sense to work on the second column.

Step 3. Find HGCD (j, k). Use the multiplier matrix A_2 to form $A_2 A_1 M = A_2 M_1 = \begin{pmatrix} s & m \\ t & n \end{pmatrix}$.

Step 4. We now have a short vector. If the second vector is not short we can reduce it using Euclidean steps (this method of reducing planar vectors is due to Gauss). Since we divide by an element in the short vector, the number of such steps is bounded.

Planar lattice reduction example

We start with a matrix for which we know a small row exists. We will explic - itly form the reduced lattice and observe sizes of its elements.

```
SeedRandom[1111];
row =
  Table[Random[Integer, {-10^100, 10^100}], {2}];
lat = {row, row + {10^10, 10^20}};
redlat = LatticeReduce[lat];
Log[10., Abs[lat]]
Log[10., Abs[redlat]]
```

```
{{99.9937, 99.8255}, {99.9937, 99.8255}}
```

```
{{10., 20.}, {99.9937, 89.9937}}
```

```
{a0, hnf} = Developer`HermiteNormalForm[lat];
Log[10., Abs[{a0, hnf}]]
```

```
{{{98.8304, 98.8304}, {99.9937, 99.9937}},
 {{0., 118.83}, {-∞, 119.994}}}
```

```
{a1, col2} =
  Internal`HGCD[Apply[Sequence, hnf[[All, 2]]]];
```

We check that this is correct.

```
a1.hnf[[All, 2]] == col2
```

```
True
```

Planar lattice reduction example

```
lattoo = a1.a0.lat
```

```
{{-1 690 297 741,
   98 562 207 300 476 944 016 323 888 983 594 108 894 500 ⋱
    069 318 932 811 218 331 933 156 802 505 639 031 591 ⋱
    298 243 692 275 868 793 220 380 069 225},
  {10 000 000 000, 100 000 000 000 000 000 000}}
```

We see that we have recovered the short vector from the known reduced form.

But now we can take a quotient, form a multiplier matrix similar to that used in HGCD, and obtain a reduction of the "large" vector".

```
q = Quotient[lattoo[[1, 2]], lattoo[[2, 2]]]
```

```
985 622 073 004 769 440 163 238 889 835 941 088 945 000 ⋱
 693 189 328 112 183 319 331 568 025 056 390 315 912 982 ⋱
 436 922
```

Planar lattice reduction example

```
a2 = {{1, -q}, {0, 1}};
latthree = a2.lattoo
```

```
{{-9 856 220 730 047 694 401 632 388 898 359 410 889 450 ↘
    006 931 893 281 121 833 193 315 680 250 563 903 159 ↘
    129 824 369 221 690 297 741,
  75 868 793 220 380 069 225},
 {10 000 000 000, 100 000 000 000 000 000 000 000}}
```

Now check that the transformations are unimodular, and get the sizes of the elements in the reduced lattice.

```
mult = a2.a1.a2;
Det[mult]
Log[10., Abs[latthree]]
```

```
-1
```

```
{{99.9937, 19.8801}, {10., 20.}}
```

Summary

We have seen a rich history of the HGCD algorithm. The major actors in this play are largely of illustrious status in computational mathematics (I'm trying not to drastically damage this chain). Various applications make this all the more of interest to computational math.

The method we presented has several virtues:

- Provably fast.
- Provably correct.
- Reasonably straightforward in regards to fixup steps.
- Can be applied immediately to rational recovery.
- Can be applied to reduction of planar lattices such as are given by binary quadratic forms.
- Faster extended gcd$\Longrightarrow$faster Hermite form$\Longrightarrow$faster linear diophantine solving. This carries over to nonlinear algebra (over rationals and integers), because it can enhance speed of Gröbner basis computations over both domains.

Selected references

A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison - Wesley Publishing Company, Reading Massachussetts, 1974.

Z. Chen and A. Storjohann. A BLAS based C library for exact linear algebra on integer matrices. These proceedings, 2005.

F. Eisenbrand. Short vectors of planar lattices via continued fractions. Information Processing Letters **79** 121 - 126, 2001

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

R. T. Moenck. Fast computation of GCDs. Proceedings of the 5th ACM Annual Symposium on Theory of Computing. 142 - 151. ACM Press, New York City, 1973.

N. Möller. On Schönhage's algorithm and subquadratic integer gcd computation.Submitted, 2005.

V. Y. Pan and X. Wang. Acceleration of Euclidean algorithm and extensions. Proceedings of the 2002 International Symposium on Symbolic and Algebraic Computation (ISSAC 2002). 207 - 213. ACM Press, New York City, 2002.

A. Schönhage. Schnelle Berechnung von Kettenbruchentwicklungen. Acta Informatica **1** 139 - 144, 1971.

D. Stehlé and P. Zimmermann. A binary recursive GCD algorithm. Rapport de recherche INRIA 5050, 2003. Published in: Proceedings of the 6th Algorithmic Number Theory Symposium (ANTS VI), 2004, D. Buell ed. 411 - 425. LCNS 3076. 2004.

K. Thull and C. K. Yap. A unified approach to HGCD algorithms for polynomials and integers. Manuscript, 1990. Available at: http://cs.nyu.edu/cs/faculty/yap/allpapers.html