A Modular GCD Algorithm

Anna Hausberger
annaahausii@gmail.com

David Auinger
post@davidauinger.eu

Bernhard Gstrein
bernhard-gstrein@mailbox.org

January 24, 2020

Abstract

We investigate the basic theory of the modular method for GCD computations. Starting with a basic algorithm for integers, we transform it into a modular one. Furthermore, we dive into a modular algorithm to compute the GCD of univariate and multivariate polynomials. We implemented all algorithms and tested them. We carried out performance measurements on the modular integer GCD algorithm and compared it with the basic one.

Contents

1 Introduction

1.1 Basic Concepts

The greatest common divisor (GCD) of non-zero elements $a_1, \ldots, a_n$ of an integral domain is defined as an element g, which and only which has the following properties:

1. g divides $a_1, \ldots, a_n$ and
2. every divisor of $a_1, \ldots, a_n$ divides g.

Euclid's algorithm can be used to compute the GCD of two positive integers a_1 and a_2. It establishes the integer remainder sequence (IRS) $a_1, a_2, \ldots, a_k$. By dividing a_{i-2} by a_{i-1}, for $i = 3, \ldots, k$, the IRS is constructed by the remainders of that division a_i:

$$a_i = a_{i-2} - q_i \cdot a_{i-1} \qquad \text{for } i = 3, \ldots, k. \tag{1}$$

The Bézout cofactors s and t can also be used to calculate the GCD of a_1 and a_2:

$$\gcd(a_1, a_2) = s \cdot a_1 + t \cdot a_2. \tag{2}$$

If the GCD of two integers a_1 and a_2 is 1, then the integers are said to be relatively prime or coprime. In this case $s \cdot a_1 + t \cdot a_2 = 1$. From this we immediately see that $s = a_1^{-1} \bmod a_2$ and $t = a_2^{-1} \bmod a_1$.

1.2 Modular algorithms

Sometimes it is easier not to solve a problem directly, but to transform it into another domain, solve the corresponding problem in that domain and transform the result back to the original domain. This is the basic idea of the modular method in computer algebra: A set of primes $p_1, \ldots, p_m$ with respective homomorphisms $\phi_{p_1}, \ldots, \phi_{p_m}$ is chosen and the problem is converted to a set of problems modulo different primes. These simpler problems are solved in $\mathbb{Z}_{p_1}, \ldots, \mathbb{Z}_{p_m}$. Finally, the partial solutions are combined by the Chinese remainder algorithm [3]:

$$\begin{array}{ccccc} & \mathbb{Z} \times \mathbb{Z} & \xrightarrow{\phi_{p_i}} & \mathbb{Z}_p \times \mathbb{Z}_p & \\ \text{GCD in } \mathbb{Z} & \downarrow & & \downarrow & \text{GCD in } \mathbb{Z}_{p_i}. \\ & \mathbb{Z} & \xrightarrow{\phi_{p_i}} & \mathbb{Z}_p & \end{array}$$

While the conversion to the modular representation and the combining of the results incurs an additional overhead, solving many problems in modular representation may be significantly easier than solving the corresponding problem in the original domain. In particular, the complexity in the modular representation is less affected by the size of the problem. Consequently, the modular method provides algorithms with extremely good complexity behavior.

It could happen that an unlucky prime was chosen, for which the GCD of $a \bmod p$ and $b \bmod p$ is different from the modular image of the integer GCD of a and b. For example, the GCD of the polynomials $a = x + 2$ and $b = x - 3$ is 1 over $\mathbb{Z}$. However, in modulo 5 representation a and b are equal, and their GCD is $x + 2 \equiv x - 3 \bmod 5$. Fortunately, there are only finitely many unlucky primes to choose from and they are rare.

1.3 Outline

The remainder of this article is organized as follows: The next section gives an overview of the modular algorithm and theory for integers. The modular algorithm and theory for univariate and multivariate polynomials is introduced in section 3. Section 4 describes our results and concludes this article.

2 Modular Algorithm for Integers

This section introduces a modular algorithm for integers. The described algorithm follows to a great extent the publication by Kenneth Weber et al.[2].

2.1 Introduction

Computing the GCD of two integers can be more efficient and much faster using the modular representation than in the original representation. This is because in the modular representation the given problem can be solved by calculating with lower numbers than the original input size. Effectively, this prevents the necessity for specialized big integer data types, making it feasible to solve the problem with standard integer data types supported by the hardware. In particular, input sizes up to $2^{16} = 65{,}536$ bits can be handled using a set of $2^{17} = 131{,}072$ moduli between 2^{31} and 2^{32} [2].

On the other hand, converting the problem into modular representation incurs many computationally expensive division and modulo operations. However, certain arithmetic operations can be performed in parallel, each processor performing the operation modulo a separate prime simultaneously. This makes the

Figure 1: Basic algorithm of modular GCD computation of integers [2].

Input: Positive integers U and V, with $U \geqslant V$
Output: $\gcd(U, V)$

1 $n \leftarrow \lfloor \lg U \rfloor + 1$
2 $\mathcal{Q} \leftarrow$ a set of at least $n + 2$ primes with $\pi(\min \mathcal{Q}) > \max\{n, 9\}$
3 Repeat
4 $\quad \mathcal{P} \leftarrow \{q \in \mathcal{Q} \mid V \bmod q \neq 0\}$
5 $\quad p \leftarrow$ an element of $\mathcal{P}$ for which $|U/V \bmod p|$ is minimal
6 $\quad b \leftarrow U/V \bmod p$
7 $\quad \mathcal{Q} \leftarrow \mathcal{Q} - \{p\}$
8 $\quad [U, V] \leftarrow [V, (U - bV)/p]$
9 Until $V = 0$
10 Return $|U|$

modular integer GCD algorithm particularly interesting for concurrent read, concurrent write, parallel random access machines (CRCW PRAM) [2]. On such a computational model, most of the modulo operations can effectively be performed in $O(1)$, mitigating the incurred additional cost of the modular representation.

Two variants of the algorithm are presented. The basic algorithm suppresses the modular representation for clarity and will also be used as a reference to demonstrate the benefit of the modular method. The modular algorithm extends the basic algorithm by converting the problem into the modular representation, solving the set of modular problems and converting the partial solutions back.

2.2 Basic algorithm

The basic algorithm as described by Weber et al. is depicted in Figure 1. First, we need to select a set of suitable primes $\mathcal{P}$. The primes are selected such to ensure that the reduction loop will terminate within at most $n + 2$ iterations. After the selection, the reduction loop is entered. During each iteration the input integers a and b (U and V in Weber's notation) are reduced using one of the $a/b \bmod p$ ($p \in \mathcal{P}$) values that is smallest in absolute value. This reduction step is repeated until b is zero. The final value of a is returned as the GCD of the input a and b.

2.3 Modular algorithm

The modular algorithm as described by Weber et al. is shown in Figure 2. The algorithm is split into four parts. First, a set of suitable primes $\mathcal{P}$ is selected. Second, the input integers a and b are converted to corresponding modular representations u_p and v_p ($p \in \mathcal{P}$). The third step is the reduction loop. The reduction

Figure 2: Final algorithm of modular GCD computation of integers [2].

Input: Positive integers U and V, with $U \geqslant V$
Output: $\gcd(U, V)$

MGCD1: [Find suitable moduli]
1 $n \leftarrow \lfloor \lg U \rfloor + 1$
2 $w \leftarrow$ an integer satisfying $\pi(2^w) - \pi(2^{w-1}) \geqslant \max\{\lceil 2^{w/2} + n \rceil, 9\}$
3 $\mathcal{Q} \leftarrow$ the set of $\lceil 2^{w/2} + n \rceil$ largest primes less than 2^w

MGCD2: [Convert to modular representation]
1 For all $q \in \mathcal{Q}$ do $[u_q, v_q] \leftarrow [U \bmod q, V \bmod q]$

MGCD3: [Reduction loop]
1 Repeat
2 $\mathcal{P} \leftarrow \{q \in \mathcal{Q} \mid V \bmod q \neq 0\}$
3 $p \leftarrow$ an element of $\mathcal{P}$ for which $|u_p / v_p \bmod p|$ is minimal.
4 $b \leftarrow u_p / v_p \bmod p$
5 $\mathcal{Q} \leftarrow \mathcal{Q} - \{p\}$
6 For all $q \in \mathcal{Q}$ do $[u_q, v_q] \leftarrow [v_q, (u_q - b v_q)/p \bmod q]$
7 Until $\forall q \in \mathcal{Q}$, $v_q = 0$

MGCD4: [Return standard representation]
1 $G \leftarrow 0$
2 Repeat
3 $p \leftarrow$ an element of $\mathcal{Q}$
4 $\mathcal{Q} \leftarrow \mathcal{Q} - \{p\}$
5 $G \leftarrow u_p + pG$
6 For all $q \in \mathcal{Q}$ do $u_q \leftarrow (u_q - u_p)/p \bmod q$
7 Until $\forall q \in \mathcal{Q}$, $u_q = 0$
8 Return $|G|$

step is repeated until all v_p are zero. Finally, the result in u_p is transformed back to the standard representation.

The algorithm indicates potential for parallel execution. In particular, the computations of the remainders denoted as "For all $q \in \mathcal{Q}$ do ..." in the algorithm can be executed in parallel, making the costly computation of many remainders less time consuming.

2.4 Implementation

Both algorithms have been implemented in SageMath as a Jupyter Notebook. The implementations are purely sequential and no efforts have been made in order to utilize the benefit of potential parallel execution so far.

The implementations have been tested thoroughly for various inputs. It was observed that the implementation of the presented modular algorithm did not always yield the correct results. Since we could not find support for the correctness of the backconversion to the standard representation in the proposed algorithm (MGCD4 in Figure 2), we implemented our own algorithm for the backconversion based on the Chinese remainder algorithm. With this adaption, all our tests passed.

Further, performance tests have been executed in order to demonstrate the better complexity behavior of the modular method. For this purpose, the execution times of both implementations for identical inputs have been measured using Python's timeit function.

3 Modular GCD of polynomials

In this section the theory and the algorithm for modular GCD computations for polynomials with one or more variables is described. For this purpose, we employ the algorithm from the book by Franz Winkler [3].

3.1 Introduction

Calculating the GCD of polynomials using Euclid's algorithm can lead to large growing coefficients. In order to mitigate this issue, one approach is to make the polynomial monic after each reduction. However, this requires extra integer GCD computations in order to reduce the fractions to the lowest terms. Consequently, the modular GCD algorithm is the method of choice when calculating the GCD of polynomials [1].

Any modular implementation needs a bound which defines the number of moduli that needs to be taken. Depending on the size of the coefficients of the GCD of two polynomials, we need to set a bound which should be derived from the bounds of the coefficients of the input. The coefficient can actually be larger than the coefficients of the input. The GCD of two polynomials a and b needs to be able to divide the leading coefficients a_m and b_n and the degree of the GCD of a and b can not be higher than the minimum degree of a and b.

The bound of the coefficients of the GCD of the two input polynomials $a(x) = \sum_{i=0}^{m} a_i\, x^i$ and $b(x) = \sum_{i=0}^{n} b_i\, x^i$ over $\mathbb{Z}$ ($a_m \neq 0 \neq b_n$) is described in absolute value by [3]:

$$2^{\min(m,n)} \cdot \gcd(a_m, b_n) \cdot \min\left(\frac{1}{|a_m|} \cdot \sqrt{\sum_{i=0}^{m} a_i^2}, \frac{1}{|b_n|} \cdot \sqrt{\sum_{i=0}^{n} b_i^2} \right). \tag{3}$$

The bound of the degree of the input polynomials over $F[x]$ can also be used and can be quite sufficient [1]:

$$\deg_x \gcd(a, b) \leq \min(\deg_x(a), \deg_x(b). \tag{4}$$

For multivariate polynomials, the bound depending on the degree of the input polynomials can be used [3]:

$$M = 1 + \min(\deg_x(a), \deg_x(b)). \tag{5}$$

3.2 Univariate polynomials

First, we need to select a set of primes $\mathcal{P}$ which do not divide the leading coefficients of the polynomials a and b. Then we need to compute the GCD of a and b modulo every p. The GCD of a and b modulo p will have the following properties:

1. $\deg(\gcd(a_p, b_p)) \geq \deg(\gcd(a, b))$
2. If the resultant of $a/\gcd(a,b)$ and $b/\gcd(a,b)$ modulo p is not 0, then $\gcd(a_p, b_p) \equiv \gcd(a, b) \bmod p$

To compute the GCD of two polynomials a and b, you need to first compute the Landau-Mignotte bound M, then select a prime $p \geq 2\,M$ which should not divide the leading coefficients of a and b. The next step would be to calculate the $\gcd_p$ of a and b modulo p. Then you should center the coefficients of the $\gcd_p$ around 0, which will result in values between $-p/2$ and $p/2$, then transform the GCD back to the standard representation and check if it divides a and b. If it does, then we have found the GCD. If not, we have chosen an unlucky prime and we need to select another prime and try again. Fortunately, there are only finitely many unlucky primes, so this algorithm will terminate, but it could happen, that p may be very big and the computation may be costly. The algorithm is summarized in Figure 3.

3.3 Multivariate polynomials

The polynomials will be described by the form $\mathbb{Z}[x_1, ..., x_{n-1}][x_n]$, where x_n is the main variable and $\mathbb{Z}[x_1, ..., x_{n-1}]$ are the coefficients. The GCD algorithm for univariate polynomials will be expanded, so that the GCD can be computed for multivariate polynomials. The picked prime will be an irreducible polynomials $p(x)$ in $\mathbb{Z}[x_1, ..., x_{n-1}]$. In fact, this polynomial $p(x)$ will be in the form $p(x) = x_{n-1} - r$ with $r \in \mathbb{Z}$. So this will lead to a simple evaluation at r for the reduction modulo $p(x)$. This leads us to the algorithm provided in Figure 4.

Figure 3: Modular GCD Algorithm for univariate polynomials [3].

GCD_MOD(modular gcd algorithm)
for given non-zero primitive polynomials $a, b \in \mathbb{Z}[x]^*$,
their greatest common divisor $g = \gcd(a, b)$ is computed.
Integers modulo m are represented as $\{k \mid -m/2 < k \leq m/2\}$.

(1) $d := \gcd(\mathrm{lc}(a), \mathrm{lc}(b))$;
$M := 2 \cdot d \cdot$ (Landau − Mignotte − bound for a, b);
[in fact any other bound for the size of the coefficients can be used]

(2) $p :=$ a new prime not dividing d;
$c_{(p)} := \gcd(a_{(p)}, b_{(p)})$; [with $\mathrm{lc}(c_{(p)}) = 1$]
$g_{(p)} := (d \bmod p) \cdot c_{(p)}$;

(3) **if** $\deg(g_{(p)}) = 0$ **then** $\{g := 1$; **return**$\}$;
$P := p$;
$g := g_{(p)}$;

(4) **while** $P \leq M$ **do**
$p :=$ a new prime not dividing d;
$c_{(p)} := \gcd(a_{(p)}, b_{(b)})$; [with $\mathrm{lc}(c_{(p)}) = 1$]
$g_{(p)} := (d \bmod p) \cdot c_{(p)}$;
if $\deg(g_{(p)}) < \deg(g)$ **then goto** (3);
if $\deg(g_{(p)}) = \deg(g)$
then $g := \mathrm{CRA}(g, g_{(p)}, P, p)$;
[actually CRA is applied to the coefficients of g and $g_{(p)}$]
$P := P \cdot p$

(5) $g := \mathrm{pp}(g)$;
if $g \mid a$ and $g \mid b$ **then return** g ;
goto (2) □

Figure 4: Modular GCD Algorithm for multivariate polynomials [3].

GCD_MODm (multivariate modular gcd algorithm)
for given non-zero polynomials $a, b \in \mathbb{Z}[x_1, \ldots, x_s][x_n]$ and $0 \leq s < n$
the greatest common divisor $g = \gcd(a, b)$ is computed by evaluation of x_s.

(0) **if** $s = 0$ **then** $g := \gcd(\mathrm{cont}(a), \mathrm{cont}(b))GCD_MOD(\mathrm{pp}(a), \mathrm{pp}(b))$; **return** g ;

(1) $M := 1 + \min(\deg_{x_s}(a), \deg_{x_s}(b))$;
$a' := \mathrm{pp}_{x_n}(a)$; $b' := \mathrm{pp}_{x_n}(b)$;
$f :=$ GCD_MODm$(\mathrm{cont}_{x_n}(a), \mathrm{cont}_{x_n}(b), s, s-1)$;
$d :=$ GCD_MODm$(\mathrm{lc}_{x_n}(a'), \mathrm{lc}_{x_n}(b'), s, s-1)$;

(2) $r :=$ an integer s.t. $\deg_{x_n}(a'_{x_s - r}) = \deg_{x_n}(a')$ or $\deg_{x_n}(b'_{x_s - r}) = \deg_{x_n}(b')$;
$g'_{(r)} :=$ GCD_MODm$(a'_{x_s - r}, b'_{x_s - r}, n, s-1)$;
$c := \mathrm{lc}_{x_n}(g'_{(r)})$;
$g_{(r)} := (d_{x_s - r} \cdot g'_{(r)})/c$ (but if the division fails goto (2)) ;

(3) $m := 1$;
$g := g_{(r)}$;

(4) **while** $m \leq M$ **do**
$r :=$ a new integer s.t. $\deg_{x_n}(a'_{x_s - r}) = \deg_{x_n}(a')$ or $\deg_{x_n}(b'_{x_s - r}) = \deg_{x_n}(b')$;
$g'_{(r)} :=$ GCD_MODm$(a'_{x_s - r}, b'_{x_s - r}, n, s-1)$;
$c := \mathrm{lc}_{x_n}(g'_{(r)})$;
$g_{(r)} := (d_{x_s - r} \cdot g'_{(r)})/c$ (but if the division fails continue) ;
if $\deg_{x_n}(g_{(r)}) < \deg_{x_n}(g)$ **then goto** (3);
if $\deg_{x_n}(g_{(r)}) = \deg(g)$
then incorporate $g_{(r)}$ into g by Newton interpolation ; $m := m + 1$;

(5) $g := f \cdot \mathrm{pp}_{x_n}(g)$;
if $g \in \mathbb{Z}[x_1, \ldots, x_s][x_n]$ and $g \mid a$ and $g \mid b$ **then return** g ;
goto (2) □

problem size	basic algorithm [s]	modular algorithm [s]
10	0.009 24	0.0244
20	0.0512	0.0997
30	0.119	0.168
40	0.223	0.338
50	0.392	0.545
60	0.630	0.808
70	0.970	1.14
80	1.37	1.53
90	1.89	1.98
100	2.46	2.57
110	3.19	3.13
120	3.99	3.84

Table 1: Execution times of the basic and the modular integer GCD algorithms for different problem sizes

4 Results and discussion

In this article, we have demonstrated how the integer GCD can be computed using the modular method. Further, we have discussed the specific benefits and challenges for the modular algorithm.

In particular, the modular algorithm comes with additional computational overhead, but has a better complexity behavior. This is also confirmed by our performance tests. Table 1 lists the results for the execution times of the basic and the modular integer GCD algorithms for different problem sizes. Apparently, the basic algorithm is significantly faster for small problems. However, going to larger problems, the execution times of the basic algorithm increases dramatically, but not as much for the modular algorithm. From a certain problem size, the modular algorithm outperforms the basic algorithm.

Importantly, these measurements were done without taking advantage of potential parallelization. By utilizing parallel execution, the advantage of the modular algorithm can become even higher.

Furthermore, we have tested the modular approach on univariate and multivariate polynomials. We have demonstrated that the modular method is very useful for implementing a GCD algorithm for polynomials.

References

[1] J. von zur Gathen. *Modern Computer Algebra - Third Edition*. Cambridge University Press, 2013.

[2] K. Weber, V. Trevisan, and L. F. Martins. *A modular integer GCD algorithm*. Elsevier Inc., 2004.

[3] F. Winkler. *Polynomial algorithms in computer algebra*. Texts and monographs in symbolic computation. Springer, 1996.