Neural, Parallel, and Scientific Computations 22 (2014) 485-496

ONE MODIFICATION OF EUCLIDIAN ALGORITHM

P. M. BEKAKOS[1], M. P. BEKAKOS[1], I. Ž. MILOVANOVIĆ[2], E. I. MILOVANOVIĆ[2], AND M. K. STOJČEV[2]

[1]Democritus University of Thrace, Xanthi, Greece
[2]Faculty of Electronic Engineering, 18000 Niš, Serbia

ABSTRACT. Modular arithmetic lets us carry out algebraic calculations on integers with a systematic disregard for terms divisible by a certain number (called the modulus). This kind of "reduced algebra" is essential background for the mathematics of computer science, coding theory, primality testing, and much more. In this paper we propose a unique approach for calculating "mod" operation, regardless of the signs of operands by which all ambiguities present in high level languages, such as C, Java, C++, Mathematica, Matlab, are overcome. Modular arithmetic is quite a useful tool in number theory. In particular, it can be used to obtain information about the solutions (or lack thereof) of a specific equation. In order to reduce the number of iteration steps during the calculation of great common divisor (GCD) and solving linear diophantine equations in two variables, based on Euclidian and extended Euclidian algorithm, we propose the usage of mod and $\overline{mod}$ operations. Several examples which justify usage of the involved operations are given.

Key words Modular arithmetic, Euclidian algorithm, GCD, Diophantine equations, Modular equations.

1. **Introduction**

In mathematics, modular arithmetic is a system of arithmetic for integers, where numbers "wrap around" after they reach a certain value - the modulus. Modular arithmetic is referenced in number theory, abstract algebra, cryptography, computer science, etc. For example, in cryptography modular arithmetic is used in a variety of symmetric key algorithms including AES, IDEA, and RC4. In computer science, especially in digital signal processing, modular arithmetic is often applied in bitwise operations and other operations involving fixed-width, cyclic data structures and linear cyclic addressing (see for example [8], [9]). Modulo operation is implemented in many programming languages, such as C, C++, Java, Pearl, Python, Basic, SQL, etc. In essence it is the remainder that is specifically computed in different programming languages. For example, in C++ negative number will be returned if the first argument is negative, and in Python a negative number will be returned if the second argument is negative.

Received June 24, 2014

In this paper, starting from the Euclidian theorem which defines remainder of an integer division as a positive number, regardless of the sign of operands, we will define a unique approach for calculating the remainder (i.e. the "mod" operation). In this way we circumvent all ambiguities during the manipulation with "mod" operation, such as manipulation with negative addresses, negative values in residue arithmetic. Besides, we define a $\overline{\text{mod}}$ operation which gives the shortage during division of two integers, and Mod operation, which is defined as $\text{Mod} = \min\{\text{mod}, \overline{\text{mod}}\}$. In many computations based on the Euclidian algorithm, the usage of Mod can reduce the number of iteration steps substantially.

The rest of the paper is organized as follows. In Section 2 some common applications of modular arithmetic are given. In Section 3 we introduce unified definitions of mod, $\overline{\text{mod}}$ and Mod operations. Based on these definitions, in Section 4 we propose a modification of Euclidian algorithm for computing GCD aiming to reduce the number of iteration steps. In Section 5 linear diophantine equations are solved in two variables using extended Euclidian and modified extended Euclidian algorithm. We conclude in Section 6.

2. **Some Common Applications of Modular Arithmetic**

A common application of modular arithmetic includes:

1. Circular addressing - an efficient method for accessing data buffers continuously without having to reset data pointers. In DSP circular addressing is called modulo addressing in the sense that the next address larger than the length of the circular buffer is the first address in the table. Modular addressing with an arbitrary modulus (not just a power of 2) is desirable enhancement to implement the pointer wraparound a circular buffer. For example $R = B + (R - B + M) \bmod L$, where R is a content of the data address generator, B is a base address, M is the computed offset increment, and L is the length of the buffer (modulus) [8].
2. Random number generator - random numbers are used in many practical applications for simulating noises. Although we cannot produce perfect random numbers by using digital hardware, it is possible to generate a sequence of numbers that are unrelated to each other. Such numbers are called pseudo-random numbers. The linear congruential method is widely used by random number generators, and can be expressed as [8, 6].
3. Dependence testing in programs - a process of determining whether two references to the same variable in a given set of loops might access the same memory location. In the most general case, a dependence testing amounts when determining whether a system of diophantine equations has a solution within a loop boundaries [1].

4. Cryptography - modular arithmetic directly underpins public key system such as RSA and Diffie -Hellman, symmetric key algorithms such as AES, IDEA, etc [4].
5. Checksum calculations - calculation of IBANs (International Bank Accounts) which use modulo 97 arithmetic to trap user input errors in bank account numbers, ISBN calculation for published books, etc, [5].

3. **Definitions of** mod**,** $\overline{\text{mod}}$ **and** Mod **Operations**

Given any two integers a and b, there exist integers q, q_1, r and r_1, respectively called quotients, the remainder and the shortage, such that [7] :

$$a = b \times q + r, \qquad 0 \le r \le |b| - 1, \tag{1}$$

and

$$a = b \times q_1 - r_1, \qquad 0 \le r_1 \le |b| - 1 \tag{2}$$

The shortage, r_1, is the amount by which a exceeds $b \times q_1$.

In mathematics and computer science, the floor and ceiling functions (Iverson functions) map a real number to the largest previous or the smallest following integer, respectively. More precisely,

1. $floor(x) = \lfloor x \rfloor$is the largest integer not greater than x, and
2. $ceiling(x) = \lceil x \rceil$ is the smallest integer not less than x.

Many programming languages have built in an integer function $int(x) = [x]$, which is defined as

$$[x] = \begin{cases} \lceil x \rceil & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ \lfloor x \rfloor & \text{if } x 0. \end{cases}$$

Integer functions mod and $\overline{\text{mod}}$, which give the remainder, r , and shortage, r_1 (see eqns. (1) and (2)), of two integers a and b, are defined as

$$r = a \bmod b = a - b \left\lfloor \frac{a}{b} \right\rfloor, \qquad \text{and} \qquad r_1 a \overline{\text{mod}} b = b \left\lceil \frac{a}{b} \right\rceil - a \tag{3}$$

The above definitions of the remainder and the shortage suffer from inconsistency. Namely, depending on the signs of a and b these functions can return negative values. On the other hand, according to (1) and (2), values r and r_1, are always positive. As a consequence of this inconsistency, in different programming languages (such as C, C++, Mathematica, Matlab, etc.) different results will be returned when a or b, or both, are negative.

To circumvent these ambiguities, we propose the following definitions for mod and $\overline{mod}$ functions:

Definition 1. *For two arbitrary integers a and $b, b \neq 0$, functions* $\mathrm{mod}: \; Z \times Z \mapsto N_0$, *and* $\overline{\mathrm{mod}}: \; Z \times Z \mapsto N_0$, *are defined as*

$$a \bmod b = \begin{cases} a - b\left\lfloor \frac{a}{b} \right\rfloor & \textit{if } b > 0 \\ a - b\left\lceil \frac{a}{b} \right\rceil & \textit{if } b < 0 \end{cases} \quad \textit{and} \quad a \overline{\mathrm{mod}}\, b = \begin{cases} b\left\lceil \frac{a}{b} \right\rceil - a & \textit{if } b > 0 \\ b\left\lfloor \frac{a}{b} \right\rfloor - a & \textit{if } b < 0 \end{cases} \tag{4}$$

We also introduce a new function, denoted as Mod, which is defined as

$$a \,\mathrm{Mod}\, b = \min\{a \bmod b, a\, \overline{\mathrm{mod}}\, b\} \tag{5}$$

According to the involved definitions, it is not difficult to conclude that function mod, $\overline{\mathrm{mod}}$ and Mod, always return positive values, regardless to the sign of a and b. Table 1 summarizes the results of mod operation for various values of integers a and b obtained by Mathematica, Matlab, C and according to our Definition 1, i.e. equation (4).

TABLE 1. Results of $a \bmod b$ operation

a	b	Matlab R2010a	Mathematica 7	C	Eq. (4)
11	4	3	3	3	3
-11	4	1	1	-3	3
11	-4	-1	-1	3	3
-11	-4	-3	-3	-3	1

As it can be seen, the results obtained according to eq. (4) are always positive, which is in accordance with eq. (1). On the other hand, the results obtained by Mathematica, Matlab and C, are not consistent with (1). Moreover, the generation of the results depends upon the used programming tool.

Apart from giving a unified approach, Definition 1 enables us to reduce a number of iteration steps in numerous algorithms in number theory. We will illustrate this on the example of Euclidian algorithm for finding greatest common divisor (GCD), and extended Euclidian algorithm for finding particular solution of linear diophantine equations.

4. Modification of Euclidian algorithm

The greatest common divisor (GCD) of two integers is always a nonnegative integer, i.e. the following is valid $GCD(a, b) = GCD(|a|, |b|)$. Therefore, in the sequel without lost of generality, we will consider only nonnegative integers, i.e. integers from the set N_0.

Given two integers a and b, with $a \geq 0, b > 0$, for calculating $GCD(a, b)$ the well known Euclidian algorithm can be used. Standard Euclidian algorithm (Algorithm_1) and its modification (Algorithm_2) that we proposed, have the following forms:

Algorithm_1 (Euclidian)

```
$r1 := a;;$   $r2 := b;$ $r := r_1 mod r_2;$
while ($r \neq 0$) do
{ $r_1 := r_2;$   $r_2 := r;$
$r := r_1 mod r_2$ }
$GCD := r_2.$
```

Algorithm_2 (modified Euclidian)

```
$r1 := a;;$   $r2 := b;$ $r := r_1 \ Mod \ r_2;$
while ($r \neq 0$) do
{ $r_1 := r_2;$   $r_2 := r;$
$r := r_1 \ Mod \ r_2$ }
$GCD := r_2.$
```

Since $r_1 \operatorname{Mod} r_2 \leq r_1 \bmod r_2$, for each r_1 and r_2 from N_0, the number of iteration steps in Algorithm_2 is always less than or equal to the number of iteration steps in Algorithm_1. Having in mind the definition of Mod operation, it is obvious that the computational complexity of one iteration step in Algorithm_2 is greater than that of the Algorithm_1. An example of GCD computation using Algorithm_1 and _2, for the case of $a = 233$ and $b = 144$ is given in Table 2.

Table 2. Computing of GCD(233,144) with Algorithm_1 and Algorithm_2

Algorithm_1				Algorithm_2		
step	r_1	r_2	$r = r_1 \bmod r_2$	r_1	r_2	$r = r_1 \operatorname{Mod} r_2$
1	233	144	89	233	144	55
2	144	89	55	144	55	21
3	89	55	34	55	21	8
4	55	34	21	21	8	3
5	34	21	13	8	3	1
6	21	13	8	3	1	0
7	13	8	5	GCD=1		
8	8	5	3			
9	5	3	2			
10	3	2	1			
11	2	1	0			
12	GCD=1					

It is well known, see Ref. [2], that the worst possible case for Algorithm_1 is finding GCD of two consecutive Fibonacci numbers, F_{n+1} and F_n. It requires n

iteration steps. On the other hand, since for Fibonacci numbers hold

$$F_{n+1} = F_n + F_{n-1} = 2F_n - F_{n-2}, \qquad n \geq 2$$

it is easy to show that finding $GCD(F_{n+1}, F_n)$ by Algorithm_2 requires $\frac{n+1}{2}$ iteration steps, only. Let us note, that achieved benefit with respect to the number of iteration steps, is maximal. When integers a and b do not satisfy $a \geq F_{n+1}$ and $b \geq F_n$, the benefit is smaller.

Very often, in the programming we met the following loop nest

```
**DO** $i_1 = L_1, U_1$
  **DO** $i_2 = L_2, U_2$
    $\ddots$
     **DO** $i_n = L_n, U_n$
  S1: $A(f(i_1, \ldots, i_n)) = \cdots$
  S2: $\cdots = A(g(i_1, \ldots, i_n))$
     **endo**
    $\vdots$
 **endo**
```

Determining whether there is a dependence from S1 to S2 (or vice versa) is equivalent to determining whether there exists an integer solution to the equation system

$$f(x_1, x_2, \ldots, x_n) = g(y_1, y_2, \ldots, y_n) \tag{6}$$

in the space defined by

$$L_i \leq x_i, y_i \leq U_i, \qquad \forall i, \quad 1 \leq i \leq n.$$

If f and g are affine functions, that is they have a form

$$f(x_1, x_2, \ldots, x_n) = a_0 + a_1 x_1 + \cdots + a_n x_n$$

$$g(y_1, y_2, \ldots, y_n) = b_0 + b_1 y_1 + \cdots + b_n y_n,$$

the dependence problem to be solved is to find solutions in the region R to the linear diophantine equation

$$a_0 - b_0 + a_1 x_1 - b_1 y_1 + \cdots + a_n x_n - b_n y_n = 0. \tag{7}$$

Rearranging terms of Eq. (7) yields the following

$$a_1 x_1 - b_1 y_1 + \cdots + a_n x_n - b_n y_n = a_0 - b_0 \tag{8}$$

which is standard form for a linear diophantine equation.

Equation (8) has a solution if and only if $GCD(a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n)$ divides $b_0 - a_0$. Thus, if the GCD of all the coefficients of loop induction variables does not divide the difference of the constant additive terms, there can be no solution to the equation anywhere - hence, no dependence can exist.

One obvious approach to find GCD of a sequence of integers, that is $GCD(a_1, a_2, \ldots,$ $a_n)$, is to apply Euclidian algorithm $n-1$ times, i.e.

$$r = GCD(a_1, a_2); \qquad r = GCD(r, a_i), \quad i = 3, \ldots, n. \tag{9}$$

If n is large, this can be time consuming. One way to speedup the computation is to involve parallelism in computation. Here we propose the following procedure

1. If $\exists a_i = 1$, then $GCD(a_1, \ldots, a_n) = 1$, go to step 5, else go to step 2.
2. Find index ind such that $a_{ind} = \min(a_1, \ldots, a_n) \neq 0$
3. If all $a_i \neq 0$ are equal to a_{ind}, then $GCD(a_1, \ldots, a_n) = a_{ind}$ and go to step 5, else go to step 4.
4. for all $i = 1, \ldots, n$ and $i \neq ind$ compute $a_i = a_i \ Mod \ a_{ind}$ and go to step 1.
5. end.

If we have $p \leq n$ processors at a disposal, steps 1 and 4 can be parallelized. Steps 2 and 3 are executed sequentially.

Remark 1. *For arbitrary sequence , the following is valid*

$$1 \leq GCD(a_1, a_2, \ldots, a_n) \leq \min\{a_1, a_2, \ldots, a_n\}.$$

Therefore, it is useful to consider sequence $a_1, a_2, \ldots, a_n$ in an increasing order.

Example: Consider the sequence $(42, 54, 105, 126)$.

If we apply the method defined by (9), the computation will be performed in the following way

$$d = 42, \quad d = GCD(d, 54) = GCD(42, 54) = 6,$$

$$d = GCD(d, 105) = GCD(6, 105) = 3, d = GCD(d, 126) = GCD(3, 126) = 3.$$

By applying the proposed procedure, we obtain

$$GCD(42, 54, 105, 126) = GCD(42, 6, 21, 3) = GCD(3, 0, 0, 0) = 3.$$

5. **Solving linear diophantine equations**

During data dependence computation in actual programs, it is often required to find a solution of a linear diophantine equation. Euclidian algorithm which is used to find GCD of two integers, can be extended to solve linear diophantine equations in two variables [3]. This algorithm is called extended Euclidian algorithm. Here we will describe both extended Euclidian algorithm and its modification based on the usage of Mod operation defined by (4).

Linear diophantine equation in two variables is of the form

$$a \times x + b \times y = c, \qquad a \times b \neq 0. \tag{10}$$

Without loss of generality, we assume that a, b and c are positive integers. Namely if some of the integers a, b or c is negative, by involving substitutions $x := \pm x$ and $y := \pm y$, a corresponding equation will meet the requirement in (10).

The main task in solving (10), is finding its particular solution $< x_0, y_0 >$. It is well known (see [3]), that for two arbitrary positive integers a and b, linear diophantine equation of the form

$$a \times x + b \times y = GCD(a, b) \tag{11}$$

always has a solution. Necessary and sufficient condition for (10) to have a solution, is that $GCD(a, b)$ divides c. If $< X_0, Y_0 >$ is an arbitrary particular solution of (11), then a particular solution of (10) is of the form

$$x_0 = \frac{c}{GCD(a,b)} X_0 \quad \text{and} \quad y_0 = \frac{c}{GCD(a,b)} Y_0.$$

General solution has the following form

$$x = x_0 - \frac{b \cdot t}{GCD(a,b)} \quad \text{and} \quad y = y_0 + \frac{a \cdot t}{GCD(a,b)}, \quad t \in Z$$

The extended Euclidian algorithm consists of two major steps. During the first step $GCD(a, b)$ and $< X_0, Y_0 >$ are determined. In the second step it is checked whether $GCD(a, b)$ divides c. If not, the equation (10) does not have a solution. Otherwise, particular and general solutions of (10) are determined. The extended Euclidian algorithm can be described as follows.

Algorithm_3 (Extended Euclidian)

```
  $\left.\begin{array}{l} r1 := a;;\quad r2 := b; \\ x1 := 1;\quad x2; = 0; \\ y1 := 0;\quad y2 := 1; r := a \end{array}\right\}$ initialization
  while $(r \neq 0)$ do
    {  $q := \lceil \frac{r1}{r2} \rceil$;
     $r := r1 - q * r2$;   $r1 := r2$;   $r2 := r$;
     $x := x1 - q * x2$;   $x1 := x2$;   $x2 := x$;
     $y := y1 - q * y2$;   $y1 := y2$;   $y2 := y$;
    }
  $GCD := r1$;   $X0 := x1$;   $Y0 := y1$.
```

Modified extended Euclidian algorithm which is based on the usage of Mod operation, defined by (4) has the following form.

Algorithm_4 (modified extended Euclidian)

```
  $r1 := a;;$            $r2 := b;$       ⎫
  $x1 := 1;$             $x2; = 0;$       ⎬ initialization
  $y1 := 0;$             $y2 := 1;$       ⎪
  $rr := a \bmod b;$     $rs := b - rr;$  ⎭
while $(rr \wedge rs \neq 0)$ do
  if $(rr < rs)$ then
    {$q := \left\lfloor \frac{r1}{r2} \right\rfloor$;
    $rr := r1 - q * r2;$   $r1 := r2;$   $r2 := rr;$
    $x := x1 - q * x2;$   $x1 := x2;$   $x2 := x;$
    $y := y1 - q * y2;$   $y1 := y2;$   $y2 := y;$}
else
    {$q := \left\lceil \frac{r_1}{r_2} \right\rceil$;
    $rs := r1 - q * r2;$   $r1 := r2;$   $r2 := rs;$
    $x := q * x2 - x1;$   $x1 := x2;$   $x2 := x;$
    $y := q * y2 - y1;$   $y1 := y2;$   $y2 := y;$}
$GCD := r1;$   $X0 := x1;$   $Y0; = y1$
```

For the sake of illustration of running Algorithm_3 and _4, we will take the following diophantine equation in two variables

$$233x + 144y = 7 \tag{12}$$

Tables 3 and 4 outline execution steps when Algorithm_3 and Algorithm_4 are used respectively.

Both algorithms compute $GCD(233, 144)$ and particular solution $< X_0, Y_0 >$ of the equation

$$233x + 144y = GCD(233, 144).$$

Since $c = 7$, and $GCD(233, 144) = 1$, i.e. 1 divides 7, the solution of linear diophantine equation (12) is

$$x_0 = -55 * 7 = -385, \quad \text{and} \quad y_0 = 89 * 7 = 623.$$

As it can be seen from Tables 3 and 4, Algorithm_3 requires 12 computational steps, while Algorithm_4 requires 7 steps. Algorithm_4 uses one additional testing at the beginning of the loop. All other computational steps in the loop body of both algorithms are of identical complexity. This obviously justifies the usage of the involved modification.

TABLE 3. The results of execution steps for Algorithm_3

Algorithm_3										
step	q	r_1	r_2	r	x_1	x_2	x	y_1	y_2	y
1	1	233	144	89	1	0	1	0	1	-1
2	1	144	89	55	0	1	-1	1	-1	2
3	1	89	55	34	1	-1	2	-1	2	-3
4	1	55	34	21	-1	2	-3	2	-3	5
5	1	34	21	13	2	-3	5	-3	5	-8
6	1	21	13	8	-3	5	-8	5	-8	13
7	1	13	8	5	5	-8	13	-8	13	-21
8	1	8	5	3	-8	13	-21	13	-21	34
9	1	5	3	2	13	-21	34	-21	34	-55
10	1	3	2	1	-21	34	-55	34	-55	89
11	2	2	1	0	34	-55	149	-55	89	-333
12		1	0		-55		89			
$GDC(233, 144) = 1 \quad X_0 = -55 \quad Y_0 = 89$										

TABLE 4. The results of execution steps for Algorithm_4

Algorithm_4										
step	q	r_1	r_2	r	x_1	x_2	x	y_1	y_2	y
1	2	233	144	55	1	0	-1	0	1	2
2	3	144	55	21	0	-1	-3	1	2	5
3	3	55	21	8	-1	-3	-8	2	5	13
4	3	21	8	3	-3	-8	-21	5	13	34
5	3	8	3	1	-8	-21	-55	13	34	89
6	3	3	1	0	-21	-55	-309	34	89	233
7		1		-55		89				
$GDC(233, 144) = 1 \quad X_0 = -55 \quad Y_0 = 89$										

In cryptography, it is often necessary to find out the so called multiplicative inverse of a given number a, i.e.

$$z = \left(\frac{1}{a}\right) modb.$$

This problem is equivalent to finding out the solution of the modular equation

$$a \times z \equiv 1 (mod b), \quad GCD(a, b) = 1,$$

for which extended Euclidian algorithm can be used.

6. Conclusion

Modular arithmetic is frequently used in number theory, group theory, ring theory, abstract algebra, cryptography, computer science, chemistry, e-banking, etc. Modulo operation is implemented in many programming languages, such as C, C++, Java, Pearl, Python, Basic, SQL, etc. In essence it is the remainder that is specifically computed in different programming languages. For example, in C++ a negative number will be returned if the first argument is negative, and in Python a negative number will be returned if the second argument is negative. In order to bypass all ambiguities during manipulation with mod operation, we have introduced a definition of mod, $\overline{\text{mod}}$ and Mod operations that always return positive values regardless to the sign of operands. This is in accordance with the Euclid's definition of the remainder. According to the involved operations, we have proposed a modification of Euclid's and extended Euclid' s algorithms aiming to reduce the number of iteration steps. We have illustrated the benefits of the introduced modifications through several examples.

REFERENCES

[1] R. Allen, K. Kennedy, *Optimizing compilers for modern architectures: A dependence based approach*, Morgan Kaufmann , 2001.

[2] G. Akritas, *Elements of computer algebra with applications*, John Wiley and Sons, Inc. placeStateNew York, 1989.

[3] J. M. Anderson, *Discrete mathematics with combinatorics*, Prentice Hall, StateNew Jersey, 2004.

[4] B. Forouzan, *Cryptography & Network Security*, McGraw Hill, 2007.

[5] B. Forouzan, F. Mosharraf, *Computer Networks: A top down approach*, Mc Graw Hill, 2011.

[6] J. Gentle, *Random number generation and placeMonte Carlo methods*, Springer, 2003.

[7] D. Knuth, *The art of computer programming*, Vol 2: Seminumerical Algorithms, Reading State MA: Addison-Wesley, 1981.

[8] Sen M. Kuo, Bob H. Lee, WeNshun Tian, *Real-Time digital signal processing: Implementations and applications*, Second Edition, John Wiley & Sons, Inc. 2006.