Fraction-Integer Method (FIM) for Calculating Multiplicative Inverse

Sattar J Aboud
Department o f Computers Science, Philadelphia University
Jordan – Amman
E-mail: sattar_aboud@yahoo.com

ABSTRACT

Multiplicative inverse is a crucial operation in public key cryptography. Public key cryptography has given rise to such a need, in which we need to generate a related public/private pair of numbers, each of which is the inverse of the other. One of the best methods for calculating the multiplicative inverse is Extended-Euclidean method. In this paper we will propose a new algorithm for calculating the inverse, based on continuous adding of two fraction numbers until an integer is obtained.

Key words: Cryptography, Fraction-Integer Method, Multiplicative Inverse, Greater Common Divisor.

1. INTRODUCTION

The multiplicative inverse of (e) modulus (n) is an integer (d) $\in Z_n$ such that e * d ≡ 1 mod n, d is called the inverse of e and denoted e^{-1} [5]. The study of inverse calculation was a stubborn science due to lack of real improvement, due to [1] the modulus inverse problem is a lot more difficult to solve. However, there were only a couple of methods. One is trivial and lengthy in calculating the inverse, because it is a sequential search. (i.e. start by d = 1, keep on adding 1 to d until e * d ≡ 1 mod n). Euclidean (the oldest, yet) the most powerful one, which is based on finding the greater common divisor between e and n, such that gcd (e, n) = gcd (e, n mod e). The algorithm solves x * y such that e * x + n * y = 1. Stein method [7] [3] which improve Euclidean method by testing for odd and even numbers of e, n, and divide e and/or n by 2 if needed before calculating the inverse. Gordon method [2] is based on using shifts to avoid lengthy multiplication and division. Baghdad method [6] is based on continuous adding of two integer numbers until an integer is obtained. The description, the storage requirements and the complexity of each method is discussed in the following sections.

2. EUCLIDEAN METHOD

This method is based on the idea that if n > e then gcd (e, n) = 1, also on finding e * x + y * n = 1 in which x is the multiplicative inverse of e [3, 4].

Algorithm
Input: a $\in Z_n$ such that gcd (a, n) = 1
Output: e^{-1} mod n, where e^{-1} = i provided that it exists

1. Set g ← n, u ← e, i ← 0, v ← 1.
2. while u > 0 do the following:
   2.1 q ← ⌊g/u⌋, t ←g - q * u,
   2.2 g ← u, u ← t, t ← i – q * v
   2.3 I ← v.
   2.4 v ← t.
3. if i < 0 then i ←n + i.
4. e^{-1} ← i

Example
Let a←7, n←60

g	u	i	v	q	t
60	7	0	1	0	0
7	4	1	-8	8	-8
4	3	-8	9	1	9
3	1	9	-17	1	-17
1	0	-17	-52	3	-52

e^{-1} ← n + i = 60 + (-17) = 43

The method needs around 6 variables, and used subtraction, multiplication division, and comparison as operations with complexity of O(log2 n).

3. STEIN METHOD

This algorithm was described by [7] and improved by penk [3] which avoids multiplications. It is based on the observation that gcd (x, y) = gcd (x/2, y) if x is even, also gcd (x, y) = 2, gcd (x/2, y/2) if both x, y are even, and gcd (x, y) = gcd ((x - y) / 2, y) if x, y are both odd.

Algorithm
Input: e $\in Z_n$ such that gcd (e, n) =1
Output: e^{-1} mod n, provided that it exists

1. while e and n is even do
   1.1 e ← ⌊e/2⌋, n ← ⌊n/2⌋.
2. u_1←1, u_2 ← 0, u_3← e, v_1← n, v_2 ←1-e, v_3 ← n.
3. if e is odd then t_1← 0, t_2 ← -1, t_3 ← -n
4. else t_1←1, t_2←0, t_3← e
5. repeat
   5.1 while t_3 is even do
      5.1.1 t_3 ← ⌊t_3/2⌋.
      5.1.2 if t1 and t_2 is even then
      5.1.3 t_1← ⌊t_1/2⌋, t_2 ← ⌊t_2/2⌋
      5.1.4 else t_1 ← ⌊(t_1 + n)/2⌋,
      5.1.5 t_2 ← ⌊(t_2 - e)/2⌋.
   5.2 if (t_3 > 0) then u_1←t_1, u_2← t_2, u_3 ← t_3
   5.3 else v_1 ← n - t_1, v_2 ← - (e + t_2), v_3 ← -t_3
   5.4 t_1 ← u_1 - v_1, t_2←u_2 - v_2, t_3← u_3 - v_3.

5.5 If ($t_1 < 0$) then $t_1 \leftarrow t_1 + n$, $t_2 \leftarrow t_2 - e$.

6. until $t_3 = 0$.
7. $e^{-1} \leftarrow u_1$.

Example

Let $e \leftarrow 7$, $n \leftarrow 60$.

e	n	u1	u2	u3	v1	v2	v3	t1	t2	t3
7	60	1	0	7	60	-6	60	0	-1	-60
								30	-4	-30
								15	-2	-15
					45	-5	15			
								-44	5	-8
								16	-2	
								8	-1	-4
								34	-4	-2
								17	-2	-1
					43		1			
								-42	5	6
								18	-2	
								9	-1	3
		9	-1	3						
								-43	4	2
								26	-3	
								43	-5	1
		43	-5	1						
								0	0	0

$e^{-1} \leftarrow u_1 = 43$

The algorithm needs around 11 variables, and uses addition, subtraction, multiplication, division by 2, and comparison with complexity of O(log2 n).

4. GORDON METHOD

This algorithm is based on the observation that (q) at Euclidian algorithm does not need to be the remainder of n / e but it can be any power of 2 up to that limit [2].

Algorithm

Input: $e \in Z_n$ such that gcd (e, n) = 1
Output: e^{-1} mod n, provided that it exists

1. $g \leftarrow n$, $i \leftarrow 0$, $v \leftarrow 1$, $u \leftarrow e$.
2. repeat
   - 2.1 $s \leftarrow -1$, $p \leftarrow 0$.
   - 2.2 If u > g then
     - 2.2.1 $t \leftarrow 0$
   - 2.3 else
     - 2.3.1 $p \leftarrow 1$, $t \leftarrow u$.
     - 2.3.2 while ($t \leq g$) do
       - 2.3.2.1 $s \leftarrow s + 1$.
       - 2.3.2.2 $t \leftarrow$ left shift t by 1.
     - 2.3.3 $t \leftarrow$ right shift t by 1.
   - 2.4 $t \leftarrow g - t$, $g \leftarrow u$, $u \leftarrow t$, $t \leftarrow i$, $i \leftarrow v$.
   - 2.5 if p =1 then
     - 2.5.1 $v \leftarrow$ left shift v by s.
     - 2.5.2 $t \leftarrow t - v$.
   - 2.6 $v \leftarrow t$.
3. until u = 0 or u = g.
4. if $i < 0$ then $i \leftarrow n + i$.
5. $e^{-1} \leftarrow i$.

Example

Let $e \leftarrow 7$, $n \leftarrow 60$.

g	u	i	v	s	p	t
60	7	0	1	0	1	14
				1		28
				2		58
				3		112
						56
7	4					4
		1				0
			8			-8
			-8			
				-1	0	
					1	4
				0		8
						4
4	3					3
		-8				1
						9
			9			
				-1	0	
					1	3
				0		6
						3
1	1					1
		9				-8
						-17
			-17			
				-1	0	
					1	1
				0		2
				1		4
						2
1						1
		-17				9
			-3			43
			43			

$e^{-1} \leftarrow 60 - 17 = 43$

The algorithm needs around 7 variables, and uses addition, subtraction, comparison, and shifts with complexity of O(log n)

5. FRACTION-INTEGER METHOD (FIM)

The idea behind the proposed method is very simple. Start with divide 1 by e, and divide n by e, then keep on adding the two results in any variable until an integer obtain.

Algorithm

Input: $e \in Z_n$ such that gcd (e, n) =1
Output: e^{-1} mod n, provided that it exists

1. Let $d \leftarrow 1.0 / e$
2. $err \leftarrow 1.0 / 2.0 * n$)
3. Let $def \leftarrow$ (double) n / e
4. cout << showpoint << fixed << setprecision(15)
5. do{
   - 5.1 $d \leftarrow (d + def)$.
   - 5.2 cout << d << endl
6. while (d – (int) (d + err) > err)
7. cout << "The multiplicative inverse of ("<<e<<") % (" << (int) (d + err) << endl
8. $e^{-1} \leftarrow d$.

Example:
Let e ← 7, n ← 60.

d	def
0.1429	8.5714
8.7143	
17.2857	
25.8571	
34.4286	
43.0000	

$e^{-1} \leftarrow d = 43$

The algorithm needs only 2 variables, and uses addition and division only, and comparison with complexity of O(log2 n))

Proof of FIM
In order to prove the algorithm, we need to prove that the algorithm will give integer number only when d is the inverse of e.

As we know that if d is the inverse of e then

1. Both e, d are positive integer numbers between [1,n] ……. ……(1)
2. gcd(e,n)=1. ……..……..……...(2)
3. $e * d \equiv 1 \bmod n$, i.e. $e * d = 1 + k * n$ (for $k \in Z$), ……...(3)

so
$d = (1+ k * n) / e = 1/ e + k * n / e$……..…... (4)

From the algorithm we see that

d =1/ e + (def + def +…+ def) i times until d is integer.

$d = 1/ e + i * def = 1/ e + i * n / e$……..... (5)

From that we know that the algorithm above is correct for i = k, but if this is the case we need to prove that (5) will give a none integer for all values of i < k, and the only integer value is when i = k, so we know d is an integer so (1+ k *n) / e is also an integer for an integer value of k. Assume that this is true for some value k (by equations (3, 4)

Then we need to proof that (1+ i * n) / e is never an integer for all values of i between [1, k - 1]. Assume that there is another value of i, 1 < i < k such that d = (1+ i * n) / e is also an integer, i.e.
i = k - 1 ---------- (6)
Then d = (1+ (k.-.1) * n) / e will be integer. So
$d = (1+ k * n - n) / e$
$= (1+ k * n) / e – n / e$
$= 1/ e + k *n / e – n / e$

But by definition (1, 3, 4) we know that $1/ e + k * n / e$ is integer, also that gcd(e, n) should be 1 (2) so if there is no greater common divisor between e and n except 1, that mean n / e is a non integer value. So subtracting a non integer value form an integer value will yield d is not an integer. Which contradicts our assumption (that d is an integer)…… (6).

Now assume that there exist an i = k - q such that d is an integer for q between [1, k - 1]. Then $d = (1+ (k - q) * n) / e = 1 / e + k * n / e – q * n / e$, and if this to be integer then q * n / e must be integer, but since gcd (e, n) =1 then q must be a multiple of e so
$d = 1/ e + k *n / e – x * n$ …… (5)
This will lead to d being a negative number d < 0 but from definition we know that both e, d must be positive (1) so there is no values for x that satisfy the definition. So the only value for q that satisfy the conditions is when q=0 and that i = k (done).

Problem of FIM method*:*
We have proved that FIM algorithm is correct, but the question is that is it implemental? Yes i.e. the algorithm will terminate giving the correct answer when implemented using the computer programming languages?

Let d_m be the mathematical value of d where $d = d_m$.
Let d_c be the calculated value of d in the computer memory and registers.
Let ζ be the error in calculating, between the mathematical value and the computer value (round off error). So

$d_m = (1_m + k_m * n_m) / e_m$ so
$= 1_m / e_m + k_m * n_m / e_m$
$= (1 / e)_m + (k * n / e)_m$

But we know that the calculated value of fractions is never exactly as the mathematical value for big values of e that when used to divide 1 and n will give a cyclic fraction number, so $(1 / e)_m = (1 / e)_c + \zeta_1$ and $(n / e)_m = (n / e)_c + \zeta_2$ where $\zeta_1 << (1 / e)_c$ and $\zeta_2 << (n / e)_c$, and $d_c = (1 / e)_c + (k * n / e)_c + \zeta_1 + k*\zeta_2$ such errors will yield that either $d_m \leq d_c$ or $d_m \geq d_c$, $d_m = d_c$ if and only if $\zeta_1 + k*\zeta_2 = 0$ i.e. $(1 / e)_m = (1 / e)_c$ and $(n / e)_m = (n / e)_c$. We know that the error ζ_1, ζ_2 is small, but multiplying ζ_2 with k will give big value to the error and the error will multiply by k, so as k is increasing the error also will increase so the best approach is to use small values for e.

Timing*:*
Figures (1 and 2) show the comparison between the proposed. The FIM algorithm with some of known algorithms, Extended Euclid, Stein, and Gordon methods, and are shown in Appendix A. We have implemented the algorithm for different numbers from one digit to 6 digits for e numbers and the result are shown in the figures below. We noticed that the time for Extended Euclid algorithm is approximately irrelevant to e or n, but other algorithms are affected by e and n. The proposed FIM algorithm outperform the other methods for small number of e and irrelevant to n. As we can see that FIM algorithm is based only on addition which is the fastest operation, and that is why it outperform the other methods except Euclid for big numbers of e.

7. CONCLUSION

For security reasons, cryptography recommends smaller values for public keys and bigger values for private keys [4]. The suggested algorithm needs lower values for public keys (lower value of e) and higher values for private key, which is fully compatible with the preferred cryptographic algorithm. The method is simple, fast and needs less storage, and its complexity is also less.

8. REFRENCES

1. B. Schneier, **applied Cryptography**, Second Edition, John Wiley and sons, 1996, p 246.

2. J. Gordon, **Fast Multiplicative inverse in modular arithmetic**, Cryptography and Coding, Clarendon Press Oxford, 1989, .pp 269 - 279.

3. D. E. Knuth, **The art of computer programming**, 2nd Ed., Addison - Wesley, Vol. 2, 1981, pp 319, 321, 339, 599.

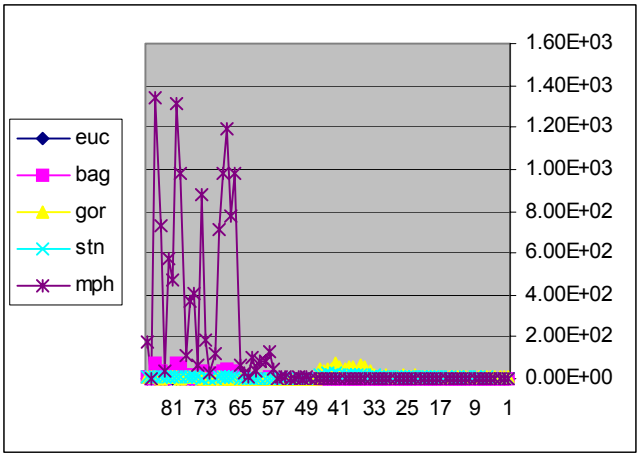
4. A. Menezes. et al, **Handbook of applied cryptography**, CRT Press, 1996, p 67, p 71.

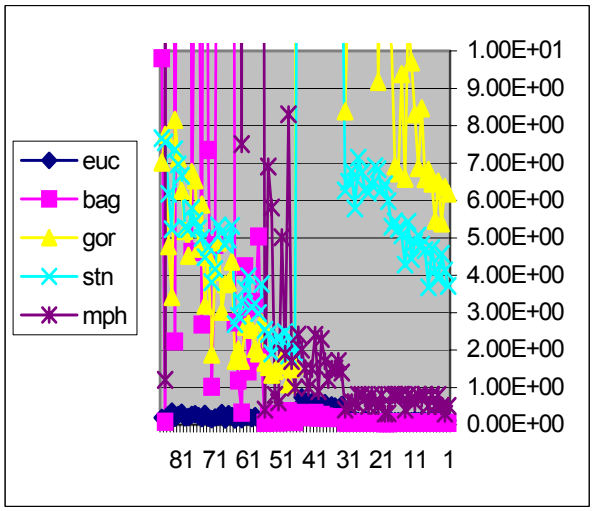
5. R. Rivest, A. Shamir., and L. Adlemen, **A method for obtaining digital signatures and public key cryptosystems**, ACM, 1978, pp 120-126.

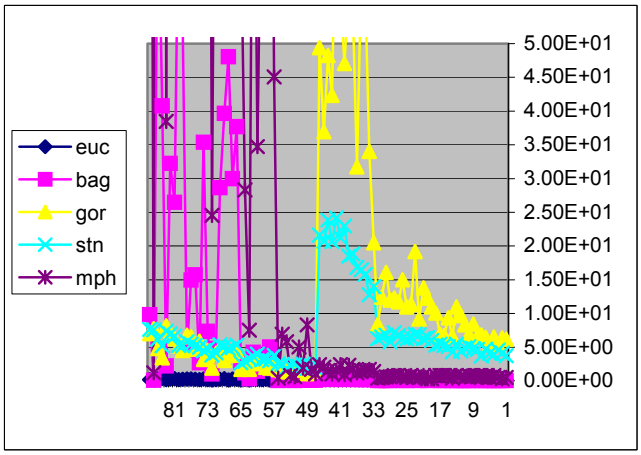
6. S. Aboud, **Baghdad Method for calculating Multiplicative Inverse**, The International Conference on Information Technology (ITCC 2004), IEEE, 5 -7 April 2004, Las Vegas, U .S. A..

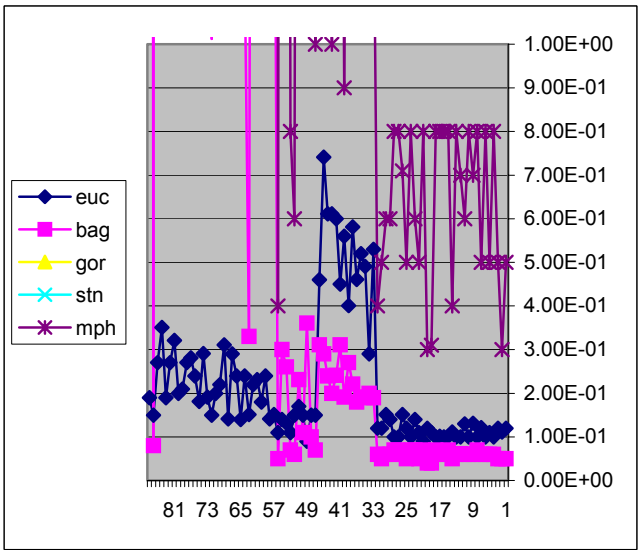
7. J. Stein, **Comp. Phys**, 1, (1967), p 397-405.

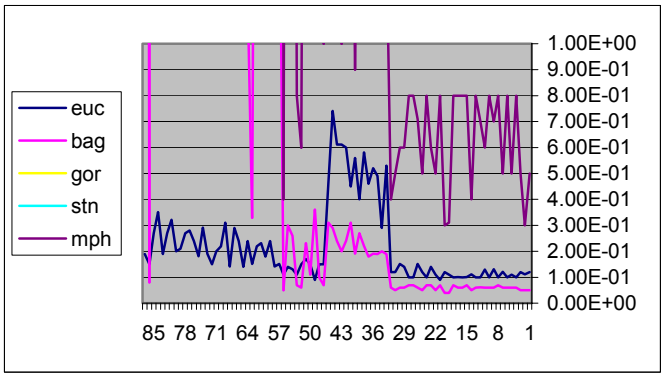
Appendix A.


euc
bag
gor
stn
mph
1.60E+03
1.40E+03
1.20E+03
1.00E+03
8.00E+02
6.00E+02
4.00E+02
2.00E+02
0.00E+00
81 73 65 57 49 41 33 25 17 9 1


euc
bag
gor
stn
mph
1.00E+01
9.00E+00
8.00E+00
7.00E+00
6.00E+00
5.00E+00
4.00E+00
3.00E+00
2.00E+00
1.00E+00
0.00E+00
81 71 61 51 41 31 21 11 1


euc
bag
gor
stn
mph
5.00E+01
4.50E+01
4.00E+01
3.50E+01
3.00E+01
2.50E+01
2.00E+01
1.50E+01
1.00E+01
5.00E+00
0.00E+00
81 73 65 57 49 41 33 25 17 9 1


euc
bag
gor
stn
mph
1.00E+00
9.00E-01
8.00E-01
7.00E-01
6.00E-01
5.00E-01
4.00E-01
3.00E-01
2.00E-01
1.00E-01
0.00E+00
81 73 65 57 49 41 33 25 17 9 1


euc
bag
gor
stn
mph
1.00E+00
9.00E-01
8.00E-01
7.00E-01
6.00E-01
5.00E-01
4.00E-01
3.00E-01
2.00E-01
1.00E-01
0.00E+00
85 78 71 64 57 50 43 36 29 22 15 8 1