Mathematica Pannonica
25/2 (2014–2015), 105–114

PERFORMANCE OF A GCD ALGORITHM FOR GAUSSIAN INTEGERS

László **Gimesi**

University of Pécs, Faculty of Natural Sciences, Department of Informatics
Ifjúság útja 6. Pécs H-7624

Received: April 27, 2017

MSC 2000:

Keywords: Gaussian integer, GCD algorithm, analysis of algorithms.

Abstract: In addition to the 2300-year-old Euclidean algorithm there is a younger sibling from 1961, the so-called binary algorithm, for computing the greatest common divisor of two integers. The binary algorithm can be extended to the Gaussian integers in more than one way. The aim of this paper is to carry out large scale numerical experiments in connection with a particular version of the binary algorithm for Gaussian integers.

1. Introduction

Euclids method was described around the year 300 B.C.E. This algorithm is a simple iterative method for finding the greatest common divisor (GCD) of two integers. This algorithm has a disadvantage: the division. This is eliminated in Stein's algorithm. Stein devised a binary greatest common divisor algorithm in 1961 [6]. Let $gcd(m,n)$ denote the greatest common divisor of the integer m and n. This algorithm is based on the following three simple facts [3]:

1. if m and n are even, then $gcd(m,n) = 2gcd(m/2, n/2)$,

E-mail address: gimesi@ttk.pte.hu

2. if m is even and n is odd, then $gcd(m, n) = gcd(m/2, n)$; if m is odd and n is even, then $gcd(m, n) = gcd(m, n/2)$,
3. if m and n are both odd, then $gcd(m, n) = gcd(m - n, n)$.

This algorithm relies solely on subtraction, parity testing, and right shifting of even numbers, and requires no division. This is more suited for binary arithmetic [5].

Several papers mention new algorithms for the greatest common divisor. Weilert [9] is devoted entirely to the complexity analysis of the $(1+i)$-ary algorithm in Gaussian integers. A fast Euclidean algorithm for Gaussian integers was presented by Collins [2]. Agarwal and Frandsen [1] present extensions of Stein's algorithm to four complex quadratic rings.

Stein's algorithm was developed by Szabó [7] for complex numbers, that is, for Gaussian integers. Szabó suggests a "paper and pencil" solution. The disadvantage of this algorithm is that it contains the comparison of numbers and in case of necessity their exchange. As the computational requirements of this operation (in case of computer implementation) is major, it is an expensive operation; therefore Szabó [7] suggests an algorithm without comparison and exchange. Szabó proves in this article that the algorithm gives a result certainly. The unfolded algorithm is the forthcoming:

Elements in the form $a_1 + a_2 i$, where a_1, a_2 are integers, form a subring of the complex numbers. Let $\omega = 1 + i$ and let $\alpha = a_1 + a_2 i$, $\beta = b_1 + b_2 i$.

The algorithm for computing a greatest common divisor of two Gaussian integers α and β:

Step 1. Initial step: set $\alpha_1 = a$, $\beta_1 = b$, $\delta_1 = 1$ where a and b are the initial values of the computation, the index is the loop counter.

Step 2. Iteration: if α_k, β_k, δ_k have already been computed, then distinguish 4 cases:

(1) If $\alpha_k = \pm\beta_k$, then α_k is a greatest common divisor of α_k and β_k. Set $\delta_{k+1} = \alpha_k\delta_k$ and the algorithm terminates.

(2) If α_k or β_k is a unit, then $\delta_{k+1} = \delta_k$ and the algorithm terminates.

(3) If $\alpha_k = 0$, then $\delta_{k+1} = \beta_k\delta_k$ or if $\beta_k = 0$, then $\delta_{k+1} = \alpha_k\delta_k$ and the algorithm terminates.

Table 1: 16 different cases of α_k and β_k

	$(0,0)$	$(0,1)$	$(1,0)$	$(1,1)$
$(0,0)$	$\alpha_{k+1} = \frac{\alpha_k}{2}$ $\beta_{k+1} = \frac{\beta_k}{2}$ $\delta_{k+1} = 2\delta_k$	$\alpha_{k+1} = \frac{\alpha_k}{2}$ $\beta_{k+1} = \beta_k$ $\delta_{k+1} = \delta_k$	$\alpha_{k+1} = \frac{\alpha_k}{2}$ $\beta_{k+1} = \beta_k$ $\delta_{k+1} = \delta_k$	$\alpha_{k+1} = \frac{\alpha_k}{2}$ $\beta_{k+1} = \frac{\beta_k}{\omega}$ $\delta_{k+1} = \omega\delta_k$
$(0,1)$	$\alpha_{k+1} = \alpha_k$ $\beta_{k+1} = \frac{\beta_k}{2}$ $\delta_{k+1} = \delta_k$	$\alpha_{k+1} = \frac{\alpha_k+\beta_k}{2}$ $\beta_{k+1} = \frac{\alpha_k-\beta_k}{2}$ $\delta_{k+1} = \delta_k$	$\alpha_{k+1} = \alpha_k$ $\beta_{k+1} = \beta_k i$ $\delta_{k+1} = \delta_k$	$\alpha_{k+1} = \alpha_k$ $\beta_{k+1} = \frac{\beta_k}{\omega}$ $\delta_{k+1} = \delta_k$
$(1,0)$	$\alpha_{k+1} = \alpha_k$ $\beta_{k+1} = \frac{\beta_k}{2}$ $\delta_{k+1} = \delta_k$	$\alpha_{k+1} = \alpha_k$ $\beta_{k+1} = \beta_k i$ $\delta_{k+1} = \delta_k$	$\alpha_{k+1} = \frac{\alpha_k+\beta_k}{2}$ $\beta_{k+1} = \frac{\alpha_k-\beta_k}{2}$ $\delta_{k+1} = \delta_k$	$\alpha_{k+1} = \alpha_k$ $\beta_{k+1} = \frac{\beta_k}{\omega}$ $\delta_{k+1} = \delta_k$
$(1,1)$	$\alpha_{k+1} = \frac{\alpha_k}{\omega}$ $\beta_{k+1} = \frac{\beta_k}{2}$ $\delta_{k+1} = \omega\delta_k$	$\alpha_{k+1} = \frac{\alpha_k}{\omega}$ $\beta_{k+1} = \beta_k$ $\delta_{k+1} = \delta_k$	$\alpha_{k+1} = \frac{\alpha_k}{\omega}$ $\beta_{k+1} = \beta_k$ $\delta_{k+1} = \delta_k$	$\alpha_{k+1} = \frac{\alpha_k}{\omega}$ $\beta_{k+1} = \frac{\beta_k}{\omega}$ $\delta_{k+1} = \omega\delta_k$

Table 2: The types of the Gaussian integers ($\alpha = a_1 + a_2 i$)

a_1	a_2	Type of α
even	even	$(0,0)$
even	odd	$(0,1)$
odd	even	$(1,0)$
odd	odd	$(1,1)$

(4) Neither of the preceding cases holds. Then compute the type of α_k and β_k and distinguish 16 case that are summarized in Table 1. (The rows are labelled by the type of α_k and the columns by the type of β_k. The types are defined in Table 2.) Then go back to Step 2.

2. Application

The program was elaborated in C++ language. Our choice was justified by the relatively easy programming and the excellent pointer arithmetic.

Binary arithmetic was used for the implementation of the program. Accordingly binary summation and subtraction. The division and multiplication with two can be solved with the help of SHIFT. For increasing efficiency (avoiding bit movements) an index was used, that is a pointer that points to the lowest digit of the number. In case of division (SHIFT right) the value of the pointer is increased by one, while in case of multiplication (SHIFT left) the value of the pointer is decreased by one.

The algorithm contains one multiplication ($\delta_{k+1} = \beta_k \delta_k$ or $\delta_{k+1} = \alpha_k \delta_k$), which was also accomplished in binary way. The work of Korn and Korn [4] was used for the implementation of operation with complex numbers. For programming technical reasons the formula in the ((0, 1),(0, 1)) and ((1, 0),(1, 0)) cells of Table 1 were modified as follows:

$$\begin{aligned}\alpha_{k+1} &= \tfrac{\alpha_k - \beta_k}{2},\\ \beta_{k+1} &= \tfrac{\alpha_k + \beta_k}{2},\\ \delta_{k+1} &= \delta_k.\end{aligned}$$

The functionality of the algorithm is not influenced by the modification.

3. Results

The values of the following parameters were examined during the run of the program: runtime, the number of completed loops and bit operations.

It was observed during testing that the examined parameters did not only depend on the size of the number (number of bit), but also on its value. Therefore our program was run with 1000 random numbers with the same length. Figure 1 shows the frequency of count of bit operations. In this example we calculated with 2 pieces of 2x10000 bit length values.

Because of the dependence on the value, the subsequent runs were always carried out with 1000 random numbers of the same length and the averages of the results were examined. The results are summarized in Table 3. In this table the columns Re(α) and Re(β) contain the real,

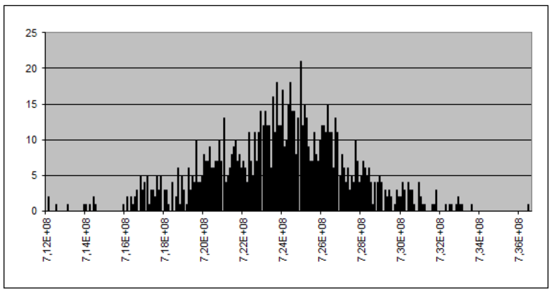

Figure 1: Distribution of frequency of operation count.
The number of bitwise operations is plotted on the horizontal axis. The values are in normal forms. For instance $7,30E+08$ means 7.3×10^8.
Y axis plots the frequency.

while columns Im(α) and Im(β) contain the imaginary bit number of the two numbers.

Parameters are depicted in a 3-dimensional coordinate system, where axis x is the bit length of one of the numbers, axis y is the bit length of the other number, value z is the examined parameter. For the sake of descriptiveness the figure were made with shading for which the Kriging interpolation method of the ArcGIS program was used.

In Figure 2 the running time, in Figure 3 the cycle number, in Figure 4 the numbers of bitwise operations are showed.

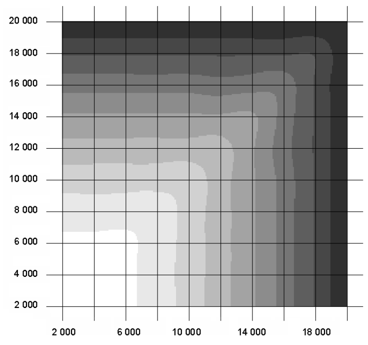

Figure 2: 3D figure of running time.
X axis plots the number of digits of α, Y axis plots the number of digits of β, Z axis (shading) plots the running time (ms).

Table 3: The parts of running parameters

Re(α)	Im(α)	Re(β)	Im(β)	Time	Cycle	Bitwise op.
1000	1000	1000	1000	48	3368	7246658
1000	1000	2000	2000	205	6737	28935905
1000	1000	3000	3000	433	10104	65108075
1000	1000	4000	4000	757	13473	115735477
1000	1000	5000	5000	1187	16842	180868699
1000	1000	6000	6000	1719	20212	260415934
1000	1000	7000	7000	2308	23574	354314039
1000	1000	8000	8000	3008	26947	462896606
1000	1000	9000	9000	3809	30327	586251065
1000	1000	10000	10000	4701	33688	723378116
⋮	⋮					
2000	2000	10000	10000	4702	33695	723590626
⋮	⋮					
3000	3000	10000	10000	4703	33688	723547718
⋮	⋮					
⋮	⋮					
8000	8000	10000	10000	4701	33684	723564106
⋮	⋮					
9000	9000	10000	10000	4701	33694	723368205
10000	10000	1000	1000	4701	33688	723378116
10000	10000	2000	2000	4702	33695	723590626
10000	10000	3000	3000	4703	33688	723547718
10000	10000	4000	4000	4701	33688	723468338
10000	10000	5000	5000	4700	33685	723439662
10000	10000	6000	6000	4701	33697	723516842
10000	10000	7000	7000	4704	33691	723460698
10000	10000	8000	8000	4701	33684	723564106
10000	10000	9000	9000	4701	33694	723368205
10000	10000	10000	10000	4703	33695	723799321

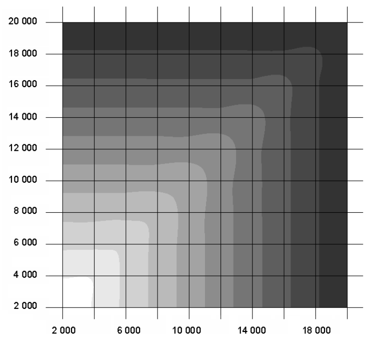

Figure 3: 3D figure of number of cycles.
X axis plots the number of digits of α, Y axis plots the number of digits of β, Z axis (shading) plots the number of cycles.

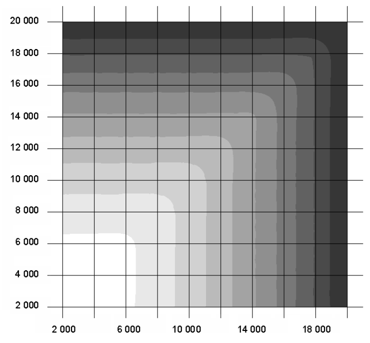

Figure 4: 3D figure of number of bitwise operations.
X axis plots the number of digits of α, Y axis plots the number of digits of β, Z axis (shading) plots the number of bitwise operations.

It can be seen in the tables and figures that the value of the parameters is defined by the number with longer bit length. During further examinations it was also concluded (see Table 3 and Table 4) that parameters are influenced by the longest member of the numbers.

Later it was examined with the method of least squares which function fits the best to the results.

Accordingly, the function that approximates the running time (see Figure 5) the best is: $y = 0.000047x^2 + 0.000734x$, the function that

Table 4: The runtime parameters, depending on the number of bits of the longest member

Bit number	Time	Cycle number	Bitwise op.
1000	48	3368	7246658
2000	202	6737	28953907
3000	435	10105	65145613
4000	769	13477	115775907
5000	1176	16846	180948712
6000	1710	20214	260489307
7000	2304	23592	354604463
8000	3008	26948	462967634
9000	3807	30330	586300778
10000	4703	33695	723799321

approximates the cycle number (see Figure 6) the best is: $y = 3.36943x$, and the function that approximates the number of bitwise operations (see Figure 7) the best is: $y = 7.23731x^2 - 1.255x$.

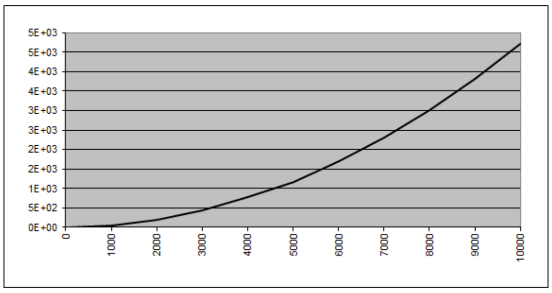

Figure 5: Graph of running time.
X axis plots the number of digits of the longest number, Y axis plots the running time (ms) in normal form.

The fit can be concluded that the running time depends on the longest member quadratic, the cycle number linearly, while the bitwise operations also quadratic.

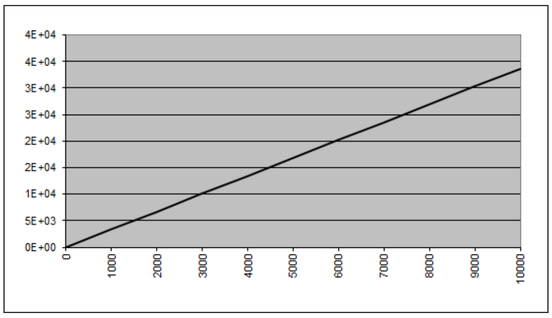

Figure 6: The number of cycles.

X axis plots the number of digits of the longest number, Y axis plots the number of cycles in normal form.

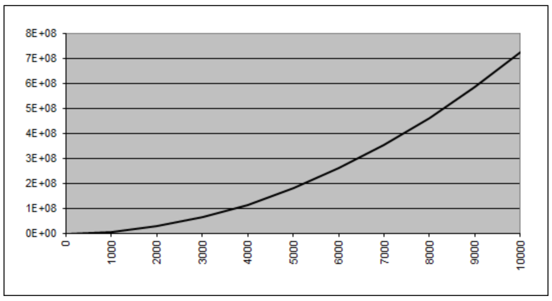

Figure 7: The number of bitwise operations.

X axis plots the number of digits of the longest number, Y axis plots the number of bitwise operations in normal form.

4. Conclusion

It was found that the parameters (running time, cycle number and bitwise operations) depend not only on that the running parameters, but also on Gaussian integers values (see Figure 1). Also the value of the parameters depends on the bit number of the longest member. In Szabó [8] the quadratic character of the dependence of the running time was established, but only in asymptotic terms.

Namely, the running time is $O(x^2)$, where x is the total number of bits of the two given numbers. Our numerical experiments suggest a more accurate form of this function. The relatively small coefficient (7.23) of x^2 clearly indicates that the binary algorithm is a practicable alternative to compute GCD in Gaussian integers.

References

[1] AGARWAL, S. and FRANDSEN, G. S., Binary GCD Like Algorithms for Some Complex Quadratic Rings, *Algorithmic Number Theory: 6th International Symposium, ANTS-VI, Burlington, VT, USA, June 13-18, 2004, Proceedings (Lecture Notes in Computer Science)*, Springer-Verlag Berlin Heidelberg. 2004, 57–71.

[2] COLLINS, G.E., A Fast Euclidean Algorithm for Gaussian Integers, *Journal of Symbolic Computation*, **33** (2002), 385–392.

[3] KNUTH, D.E., The Art of Computer Programming. Volume 2: Seminumerical Algorithms, *Addison-Wesley* 2001.

[4] KORN, G.A. and KORN, T.M., Mathematical Handbook for Scientists and Engineers, *McGraw-Hill Book Company*, 1975. 2004, 57–71.

[5] SHEUELING, C.S., From Euclid's GCD to Montgomery Multiplication to the Great Divide, *Sun Microsystems, Inc.*, 2001.

[6] STEIN, J., Computational problems associated with Racah algebra, *Journal of Computational Physics*, **1** (1967), 397–405.

[7] SZABÓ, S., A Paper-and-Pencil gcd Algorithm for Gaussian Integers, *The College Mathematics Journal*, **5** (2005), 374–380.

[8] SZABÓ, S., Variants of an algorithm of J. Stein, *Journal of International Mathematical Virtual Institute*, **1** (2011), 1–16.

[9] WEILERT, A., (1 + i)-ary GCD Computation in Z[i] as an Analogue to the Binary GCD Algorithm, *Journal of Symbolic Computation*, **30** (2011), 605–617.