Contents

Abstract

The behavior of Euclid's algorithm to compute **G**reatest **C**ommon **D**ivisors and its connection with continued fractions are explained advantageously in terms of products of 2-by-2 matrices. One byproduct is a quick proof of Lamé's theorem, which bounds the number of divisions the algorithm must perform, thus explaining its speed. Among other applications of the algorithm are the solution of linear congruences $a \cdot x \equiv c$ **mod** b and the quick computation of determinants of integer matrices. A GCD is needed also to compute a **L**east **C**ommon **M**ultiple. Actual computer programs to compute GCDs and LCMs malfunction when their inputs or outputs are integers so big that the programs encounter overflow or roundoff. Because it is overlooked so often, roundoff incurred by floating-point arithmetic can have noxious consequences unless the programs incorporate complicated precautions exemplified by MATLAB programs supplied here. When allowed to do so, they take advantage of a little extra-precise arithmetic without which no practicable defense exists against plausible but utterly wrongly computed GCDs and LCMs.

MATLAB programs `gcd` and `lcm` supplied here are needed in Hilbert matrix computations in
`<www.cs.berkeley.edu/~wkahan/MathH110/HilbMats.pdf>`

Given two positive integers $a \ge b > 0$ we seek their *Greatest Common Divisor* (GCD), which is the biggest integer d that divides both a and b leaving no remainder. Ordinary long division computes a positive integer quotient $q := \ddot{e}a/b\hat{u}$ and leaves a remainder $r := a - q\cdot b$ that satisfies $0 \le r < b$. Clearly every divisor of both a and b divides r too, and conversely every divisor of both b and r divides $a = q\cdot b + r$ too; therefore $GCD(a, b) = GCD(b, r)$. But the pair (b, r) is *smaller* than the pair (a, b) in the sense that $b \le a$ and $r < b$. This leads to an algorithm …

Euclid's GCD Algorithm

Given integers $a \ge b > 0$, set $r_0 := a$ and $r_1 := b$ and perform successive long divisions getting, for $j = 1, 2, 3, \ldots, n$ in turn until $r_{n+1} = 0$, quotients q_j and remainders r_j that satisfy

$$r_{j-1} = q_j\cdot r_j + r_{j+1} \quad \text{with} \quad 0 \le r_{j+1} < r_j .$$

(Here at step j we divide r_{j-1} by r_j to get quotient q_j and remainder r_{j+1}, stopping when a remainder $r_{n+1} = 0$. At that point $q_n > 1$; can you see why?) The algorithm stops because this decreasing sequence of $n+1$ positive integers, $r_0 = a \ge r_1 = b > r_2 > \ldots > r_{n-1} > r_n > r_{n+1} = 0$, cannot have $n > b$. Then $GCD(a, b) = r_n$ because, as explained in the first paragraph,

$$GCD(a, b) =: GCD(r_0, r_1) = GCD(r_1, r_2) = \ldots = GCD(r_{n-1}, r_n) = GCD(r_n, r_{n+1}) = r_n .$$

The quotients q_j appear to play no important role in the foregoing algorithm, but appearances can mislead. By translating the algorithm's recurrence into matrix language we find uses for q_j :

Set $\begin{bmatrix} r_0 \\ r_1 \end{bmatrix} := \begin{bmatrix} a \\ b \end{bmatrix}$ first; then for $j = 1, 2, 3, \ldots, n$ in turn confirm that $\begin{bmatrix} r_j \\ r_{j+1} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -q_j \end{bmatrix} \begin{bmatrix} r_{j-1} \\ r_j \end{bmatrix}$, with $0 \le r_{j+1} < r_j$ and $r_{n+1} = 0$, so $\begin{bmatrix} r_n \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & -q_n \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -q_{n-1} \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -q_{n-2} \end{bmatrix} \cdots \begin{bmatrix} 0 & 1 \\ 1 & -q_2 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -q_1 \end{bmatrix} \begin{bmatrix} r_0 \\ r_1 \end{bmatrix}$.

Now set row $\begin{bmatrix} B & A \end{bmatrix} := \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -q_n \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -q_{n-1} \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -q_{n-2} \end{bmatrix} \cdots \begin{bmatrix} 0 & 1 \\ 1 & -q_2 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & -q_1 \end{bmatrix}$ to obtain two integers A and B (not both positive) satisfying $GCD(a, b) = r_n = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} r_n \\ 0 \end{bmatrix} = \begin{bmatrix} B & A \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = B\cdot a + A\cdot b$.

We have just found that $GCD(a, b)$ is a linear combination of a and b with integer coefficients, thus proving (regardless of whether $a \ge b > 0$ or $b \ge a > 0$) the following …

Theorem 1: As $\overline{A}$ and $\overline{B}$ run independently through all integers the expression $\overline{B}\cdot a + \overline{A}\cdot b$ runs through a set of integers among which the smallest positive integer is $GCD(a, b) = B\cdot a + A\cdot b$.

Hard Exercise: Running $\overline{A}$ and $\overline{B}$ through *all* integers is unnecessary: Theorem 1 remains true after restrictions $|\overline{A}| < a$ and $|\overline{B}| \le b \le a$ are imposed; why? Can you prove $|A| < a/GCD(a, b)$ and $|B| \le b/GCD(a, b)$? See below.

There are two ways to compute A and B. The easiest is to evaluate from-left-to-right the matrix product defining $\begin{bmatrix} B & A \end{bmatrix}$ *after* all the q_j's have been computed; this gives rise to a recurrence:

$$s_n := 1 ; \quad s_{n-1} := -q_{n-1} ; \quad \text{for } j = n-2, n-3, \ldots, 2, 1 \text{ in turn } \quad s_j := s_{j+2} - q_j\cdot s_{j+1} .$$

Finally $A := s_1$ and $B := s_2$. Another way to compute them is to evaluate from-right-to-left the matrix product defining row $\begin{bmatrix} B & A \end{bmatrix}$ *simultaneously* with the computation of the q_j's :

$\begin{bmatrix} B_0 & A_0 \end{bmatrix} := \begin{bmatrix} 0 & 1 \end{bmatrix}$; $\begin{bmatrix} B_1 & A_1 \end{bmatrix} := \begin{bmatrix} 1 & -q_1 \end{bmatrix}$; for $j = 2, 3, \ldots, n-1$ in turn $\begin{bmatrix} B_j & A_j \end{bmatrix} := \begin{bmatrix} 1 & -q_j \end{bmatrix} \begin{bmatrix} B_{j-2} & A_{j-2} \\ B_{j-1} & A_{j-1} \end{bmatrix}$.

Finally $\begin{bmatrix} B & A \end{bmatrix} := \begin{bmatrix} B_{n-1} & A_{n-1} \end{bmatrix}$. Note that q_n never figures in the computation of A and B . Whichever way be chosen to compute A, B and $GCD(a, b) = B \cdot a + A \cdot b$, the algorithm is called "the Extended Euclidean Algorithm" and has important applications. Here is one of them:

Exercise: Given integers a, c and $b > 0$, when does " $a \cdot x \equiv c$ **mod** b " have integer solutions x ? Here we pronounce " $p \equiv q$ **mod** b " as " p is *congruent* to q mod b " and mean that $p - q$ is divisible by b . Let $d := GCD(a, b)$. If $c \equiv 0$ **mod** d exhibit all d noncongruent solutions x ; otherwise prove no solution x exists.

Continued Fractions

If $d = GCD(a, b)$ then $(a/d)/(b/d)$ exhibits a/b "in lowest terms" but it is not the only unique encoding of positive rational numbers. By substituting $r_{j-1}/r_j = q_j + 1/(r_j/r_{j+1})$ repeatedly for $j = 1, 2, \ldots, n$ in turn we obtain a *Terminating Continued Fraction*

$$\frac{a}{b} = q_1 + \cfrac{1}{q_2 + \cfrac{1}{q_3 + \cfrac{1}{\quad + \cfrac{1}{q_{n-1} + \cfrac{1}{q_n}}}}} \; .$$

This is *the* continued fraction for the rational number a/b . Here $q_1 \geq 1$ because $a \geq b > 0$; in fact every $q_j \geq 1$ and the last $q_n \geq 2$ to ensure that the encoding of each rational $a/b > 1$ by a finite sequence $(q_1, q_2, q_3, \ldots, q_{n-1}, q_n - 1)$ of positive integers be unique. Euclid's algorithm converts a rational number given as a ratio of integers into its continued fraction; how do we get back? The obvious way evaluates the continued fraction "bottom-up" : $R_{n+1} := 0$; $R_n := 1$; for $j = n, n-1, n-2, \ldots, 2, 1$ in turn $R_{j-1} := q_j \cdot R_j + R_{j+1}$; finally $a/b = R_0/R_1$ in lowest terms.

Exercise: Confirm that every integer $R_j = r_j/GCD(a, b)$.

Translating the bottom-up evaluation of the continued fraction into matrix terms yields first $\begin{bmatrix} R_{j-1} \\ R_j \end{bmatrix} = \begin{bmatrix} q_j & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} R_j \\ R_{j+1} \end{bmatrix}$, then $\begin{bmatrix} R_0 \\ R_1 \end{bmatrix} = \begin{bmatrix} q_1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} q_2 & 1 \\ 1 & 0 \end{bmatrix} \cdots \begin{bmatrix} q_{n-1} & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} q_n & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$. This last expression offers two interesting opportunities. One is a way to evaluate the continued fraction "top-down" :

$\begin{bmatrix} h_0 \\ g_0 \end{bmatrix} := \begin{bmatrix} 1 \\ 0 \end{bmatrix}$; $\begin{bmatrix} h_1 \\ g_1 \end{bmatrix} := \begin{bmatrix} q_1 \\ 1 \end{bmatrix}$; for $j = 2, 3, \ldots, n$ in turn $\begin{bmatrix} h_j \\ g_j \end{bmatrix} := \begin{bmatrix} h_{j-1} & h_{j-2} \\ g_{j-1} & g_{j-2} \end{bmatrix} \begin{bmatrix} q_j \\ 1 \end{bmatrix}$; finally $\begin{bmatrix} R_0 \\ R_1 \end{bmatrix} := \begin{bmatrix} h_n \\ g_n \end{bmatrix}$.

This top-down evaluation turns out to be a good way to evaluate endless continued fractions that encode non-rational numbers; successive ratios h_j/g_j can be shown to converge alternatingly.

Exercise: The endless continued fraction in which every $q_j = 1$ represents $\mu := (1 + \sqrt{5})/2$; can you see why?

Another opportunity offered by that long matrix product is a clear proof of ***Lamé's Theorem*** : To compute $d := GCD(a, b)$ for $a \geq b > 0$, Euclid's algorithm needs $n \leq 1 + \ln(b/d)/\ln(\mu)$ divisions.

Exercise: Prove it by showing every R_j is at least as big as if every $q_j = 1$ except $q_n = 2$, so $R_1 \geq f_{n+1}$, a Fibonacci number, and $f_{n+1} = (\mu^{n+1} - (-1/\mu)^{n+1})/(\mu + 1/\mu) \geq \mu^{n-1}$.

Seven More Exercises:

Suppose given integers $M > 1$ and $N > 1$ have $GCD(M, N) = 1 = n\cdot M - m\cdot N$ for some integers m and n whose signs are not yet determined. Whether $M \geq N$ or not won't matter anymore.

1) Show why m and n must have the same nonzero sign.

Henceforth we can assume that $n > 0$ and $m > 0$; otherwise swap M and N , *etc*.

2) What is GCD(m, n) ?

3) Show how to replace m and n respectively by $\overline{m}$ and $\overline{n}$ satisfying
$$0 < \overline{m} < M , \quad 0 < \overline{n} < N \quad \text{and} \quad 1 = n\cdot M - m\cdot N = \overline{n}\cdot M - \overline{m}\cdot N .$$

Henceforth we can assume that $0 < m < M$ and $0 < n < N$ and $n\cdot M - m\cdot N = 1$. (†)

4) Exhibit instances of pairs (M, N) and (m, n) which satisfy these assumptions (†), but for which $M > N$ in one instance, and $M < N$ in another.

5) Given that the pairs (M, N) and (m, n) satisfy (†), show how to obtain a pair $(\overline{m}, \overline{n})$ that satisfies $0 < \overline{m} < M$ and $0 < \overline{n} < N$ and $\overline{m}\cdot N - \overline{n}\cdot M = 1$, as if M and N had been swapped.

6) Show why (†) implies that $M - N$ and $m - n$ have the same nonzero signs unless $m = 1 = n$.
(Hint: $(m + n)\cdot(M - N) - 1 = (m - n)\cdot(M + N) + 1$.)

. . . . . .

7) Any two nonzero integers x and y determine a family of sets { i, j, k, m, n } of five integers satisfying $n > 0$, $\begin{bmatrix} i & j \\ k & m \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} n \\ 0 \end{bmatrix}$ and $\det(\begin{bmatrix} i & j \\ k & m \end{bmatrix}) = 1$. However { i, j, k, m, n } may be found, which of the five integers are determined uniquely by x and y ?

. . . . . .

Solution 7): Integers k, m and n are determined uniquely but i and j are not. To see why, observe first from the determinant that k and m can have no nontrivial (other than ±1) common divisor since it has to divide the determinant 1 . Next compute the inverse of the 2-by-2 matrix to obtain $\begin{bmatrix} m & -j \\ -k & i \end{bmatrix}$ and infer $\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} m \\ -k \end{bmatrix} n$. This means that $m/k = -x/y$ in lowest terms, and that n is the Greatest Common Divisor of x and y . Thus are k, m and n fixed uniquely by x and y . However, i and j can be replaced respectively by $i + L\cdot k$ and $j + L\cdot m$ respectively for any integer L with no alteration to the given constraints; this amounts merely to replacing $\begin{bmatrix} i & j \\ k & m \end{bmatrix}$ by $\begin{bmatrix} 1 & L \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} i & j \\ k & m \end{bmatrix}$.

The foregoing solution explains all that the problem requested, namely the uniqueness of the last three integers in the set { i, j, k, m, n } . To explain why integers i and j must exist is easier than explaining the Extended Euclidean Greatest-Common-Divisor algorithm that finds them.

Determinants of Integer Matrices

What use are Exercise 7's integers i, j, k, m, n ? One use is in a little-known but fairly efficient algorithm to compute the determinant or inverse of a matrix of modest dimensions whose entries are all integers. Consider the determinant of a given integer matrix E :

We can reduce E to an upper-triangular U whose determinant, the product of its diagonal elements, is the same as det(E) . To this end we premultiply E by a sequence of matrices each with determinant 1 and different from the identity matrix only in a 2-by-2 submatrix on the diagonal. First premultiply to annihilate the lower left corner element of E ; next annihilate the element above it, and so on up the first column until only its first element is nonzero. Then do the same to the second column's subdiagonal elements, and so on until all subdiagonal elements become zeros. Before each premultiplication construct a 2-by-2 matrix like the one in problem 7, using for x and y the leading (presumably nonzero) elements of adjacent rows; then the premultiplication will replace x by $n > 0$ and y by 0 . This n takes the place of the y in the next annihilation.

Relatively few divisions occur in this algorithm, all concerned with finding the 2-by-2 matrices, and most of these divisions have either short divisors or short quotients. Moreover annihilations starting at the bottom of each column produce a value of n , which becomes y at the next annihilation, that usually turns out to be 1 (or the column's greatest common divisor). Thus, in most of the 2-by-2 matrices three of the four elements are ± 1 or 0 ; this is why the method is efficient despite that the integers in triangular factor U can grow very big.

Hard Exercise: Compare the computational costs of the foregoing algorithm and Chio's Trick, *q.v.*, posted for this course at `<www.cs.berkleley.edu/~wkahan/MathH110/Chio.pdf>`.

The foregoing algorithm is part of the reduction of E to an upper-triangular *Hermite Normal Form*, which is on the path towards the *Smith Normal Form*, but these are stories for a different course on Abstract Algebra involving not just integers but also matrices whose entries are polynomials with rational coefficients. These normal forms can entail gargantuan integers, so their efficient computation requires algorithms much more subtle than the foregoing.

LCM, the Least Common Multiple

LCM(j, k) = LCM(|j|, |k|) is the least positive integer that both nonzero integers j and k divide leaving no remainder; consequently LCM(j, k) = |j|·|k|/GCD(j, k) .

Exercise: Confirm the last equation and then LCM(k, j) = LCM(j, k) & GCD(k, j) = GCD(j, k) .

Both functions GCD and LCM can be extended to sets $\{k_1, k_2, k_3, \ldots, k_m\}$ of $m > 2$ nonzero integers by defining first $GCD(k_1, k_2, k_3, \ldots, k_n) := GCD(GCD(k_1, k_2, \ldots, k_{n-1}), k_n)$ and then $LCM(k_1, k_2, k_3, \ldots, k_n) := LCM(LCM(k_1, k_2, \ldots, k_{n-1}), k_n)$ for $n = 3, 4, 5, \ldots, m$ in turn.

Exercise: Confirm that the order of the integers in the set $\{k_1, k_2, k_3, \ldots, k_m\}$ does not matter.

The functions GCD and LCM have interesting interlocking properties whose description will be eased first by temporary abbreviations γ(…) := GCD(…) and λ(…) := LCM(…) , and second by the temporary assumption that all three integers i, j and k are positive. For a more extensive discussion of these properties see ch. 2 of K.H. Rosen's book *Elementary Number Theory and its Applications*, 4th ed. (2000), Addison-Wesley, Mass.

Exercise: Confirm the following identities:

γ(i·j, i·k) = i·γ(j, k) and λ(i·j, i·k) = iλ(j, k) .

γ(λ(i, j), k) = λ(γ(i, j), γ(j, k)) and λ(γ(i, j), k) = γ(λ(i, j), λ(j, k)) .

γ(λ(i·j), λ(j·k), λ(k·i)) = λ(γ(i·j), γ(j·k), γ(k·i)) .

γ(i, j, k)·λ(i·j, j·k, k·i) = λ(i, j, k)·γ(i·j, j·k, k·i) = i·j·k .

i·j·kγ(i, j, k) = λ(i, j, k)·γ(i, j)·γ(j·k)·γ(k·i) .

They are not all easy to prove.

Computer Programs for GCD and LCM

Every computer has limited extent in space and time. Preoccupation with the limits distinguishes computer scientists from mathematicians. When we write a program to compute a mathematical function, we choose, perhaps unwittingly, to reconcile our expectations with those limits. A conscientious choice poses mathematical challenges some of which are sampled in what follows.

First we must decide how the programs shall handle boundary integers, namely 0 and ∞. If you prefer to exclude ∞ from the set of integers, you are in good company. Computer programs cannot be so picky; they must handle in a reasonable manner whatever input comes their way. The only alternative is to stop the computer when a program refuses to accept some input deemed invalid; but the consequences of stopping a computer are difficult to predict, always annoying, and sometimes dangerous. A program should refuse to accept data only if refusal is the least unreasonable option after all others have been considered and found to cause worse confusion.

Therefore convention assigns GCD(k, 0) := GCD(k, ∞) := k except GCD(0, ∞) := 0, and then LCM(k, 0) := 0 and LCM(k, ∞) := ∞ except LCM(0, ∞) := *NaN*, which stands for "*Not a Number*". It is an unacceptable input whose creation from non-*NaN* inputs raises an *Invalid Operation* flag that the program's user can detect subsequently at his/her/its convenience. This flag exists in hardware conforming to IEEE Standard 754 for *Floating-Point Arithmetic* but, alas, is inaccessible through most programming languages like MATLAB and *Java*.

Computers cope with at most finitely many distinct integers; the rest are too big. This limitation afflicts LCM(…) because its output is usually bigger than its inputs. If too big, the output must either *Overflow* or get *Rounded Off* to something else. Many programming languages cannot detect the altered result of *Integer Overflow*; and it stops the computer with an error message for many of the others, like MAPLE and Prof. Yuji Kida's UBASIC. Some languages, MATLAB and early versions of BASIC among them, represent all but their smallest integers as floating-point variables with some preassigned number of so-called *Significant Digits*. Any integer wider than that gets rounded off to a represenatable value or else, if extremely big, overflows to ∞. Either Procrustean action raises an *Inexact Operation* flag in standard-conforming hardware but, alas, this flag too is inaccessible from MATLAB and most other programming languages.

In short, integer inputs and outputs bigger than some obscure thresholds can invalidate computed GCDs and LCMs. In floating-point arithmetic GCDs are threatened, despite that no integers in the Euclidean algorithm need be bigger than its inputs, because some of those integers, though smaller in magnitude, may require an extra significant digit beyond the inputs'. LCMs may need more significant digits than the arithmetic affords. Extra-precise intermediate arithmetic, when available, produces correct GCDs and helps warn of LCMs too big. Even so, the following MATLAB programs are complicated almost grotesquely by their attempts to cope with all threats.

MATLAB Programs for GCD and LCM

MATLAB's own programs for these functions can be exhibited by the commands `type gcd` and `type lcm`. Partly because MATLAB's own `gcd` and `lcm` programs do not make fuller use of the floating-point hardware they run on, they malfunction for some inputs. For instance, they compute `gcd(3, 2^80)` = 3 ≠ 1 and `gcd(28059810762433, 2^53)` = 28059810762433 ≠ 1. They abort if ∞ is an input. They cannot handle more than pairs of inputs, so LCM(i, j, k) must be obtained from the expression `lcm(lcm(i, j), k)`. Its value should not depend upon the order of { i, j, k } but does for { 12647423, 712176643, 12658905 }, giving `lcm(lcm(…)…)` two values 1.14021279681837e23 and 1.98608743567039e19 of which one is the true LCM 19860874356703880745 rounded correctly to 53 sig. bits.

MATLAB's roundoff threshold for its 53-sig.-bit floating-point variables is `eps` = 2^{-52}. They can hold the *consecutive* integers 0, 1, 2, 3, …, 2^{53} = 9007199254740992. Numbers bigger than this 16-digit integer get rounded off to 53-sig.-bit integers. Because current versions of MATLAB display at most 15 sig. dec. instead of 17, distinct big integers may look the same on screen unless displayed in a 16-digit *hexadecimal* format eschewed here.

By exploiting arithmetic capabilities accessible through some versions of MATLAB on some computers, the functions `gcd` and `lcm` provided below either malfunction far less often than MATLAB's own, or offer easy ways to detect almost all practically unavoidable malfunctions.

Without those capabilities, the functions `gcd` and `lcm` provided below are slightly faster than MATLAB's own but no more reliable. Some of their malfunctions are exposed by the two test programs `gcdtest` and `lcmtest` provided below along with some of their results.

Function `precn` below is not used by `gcd` nor `lcm`, but is called by `gcdtest` and `lcmtest` only to uncover the precision of arithmetic MATLAB uses to accumulate matrix products. Early versions on PCs and old 680x0-based Macintoshes accumulated these products extra-precisely, to 64 sig. bits, before rounding them down to 53 as they were stored. MATLAB 6.5 on PCs can be commanded to do that when the factors and products fit into the computer's cache-memory. On Power-Macs and iMacs the accumulation benefits (rarely by much) from *Fused Multiply-Adds* that commit only one 53 sig. bit rounding error per expression of the form "x ± y·z". Most users of MATLAB ignore whatever `precn` exposes, though occasionally such details matter crucially; *e.g.*, see `<www.cs.berkeley.edu/~wkahan/MxMulEps.pdf>`. And the test results below show how such details determine whether `gcd` and `lcm` pass all their tests.

```
function [g,c,d] = gcd(a,b)
%GCD    Greatest Common Divisor.
%   G = gcd(A,B)  is an array of  Greatest Common Divisors  of the
%   corresponding elements of  A  and  B .  These arrays must contain
%   only integers and must have the same size unless one is a scalar.
%   By convention  gcd(x, 0) = gcd(x, Inf) = |x| ;  gcd(0, Inf) = 0 .
%   Otherwise  gcd  is a finite positive integer computed correctly,
%   despite roundoff no matter how big elements of  A and B  may be,
%   only under circumstances discussed in the fourth paragraph below.
%   Correct values of  gcd(3, 2^80) = gcd(28059810762433, 2^53) = 1 .
%
%   G = gcd(A)  is a row of which each element is the  GCD  of the
%   corresponding column of the array  A  of integers.
%
%   [G,C,D] = gcd(A,B)  also returns C and D so that  A.*C + B.*D = G
%   and  |C|.*G <= |B|  and  |D|.*G <= |A|  with equality only rarely.
%   [C, D]  is useful for solving Diophantine equations and computing
%   Hermite transformations.  Note that another possibility for pair
%   [C, D]  is  [C, D] - [S.*B./G, -S.*A./G]  where  S = sign(B.*C) ;
%   one pair  [C,D]  may suit your application better than the other.
%
%   Roundoff can spoil  A.*C + B.*D = G  unless  |A.*C| < 2/eps  and
%   |B.*D| < 2/eps .  Wherever  max(|A|,|B|) > 2/eps  there  [G,C,D]
%   MAY BE WRONG  except on  Power Macs,  whose  G  is always correct
%   even if  [C,D]  is not.  If  max(|A|,|B|) <= 2048/eps ,  or if
%   min(|A|,|B|) <= 2048 ,  then  G  (if not  [C,D])  is correct also
%   on old  680x0-based  Macs,  and also on  Intel-based  PCs  with
%   64-sig.-bit accumulation of matrix products enabled via  Matlab
%   6.x's  invocation  " system_dependent('setprecision', 64) " .
%
%   See also  LCM  and,  for  Matlab 3.5,  reshape,  isinf  and,  for
%   386-Matlab 3.5 & PC-Matlab 4.2,  r0und,  all as modified by  W.K.

%   Algorithm: See Knuth Volume 2, Section 4.5.2, Algorithm X sped up
%   Original Author: John Gilbert, Xerox PARC;  sped up by  W. Kahan
%   Original Copyright (c) 1984-98 by The MathWorks, Inc.
%   Original Revision: 5.9   Original Date: 1997/11/21 23:45:38
%   First modified by  W.K.  in  1990  to fix  gcd(3, 2^80) = 3 .
%   $Revision: 6.5.W.K. $  $Date: 2008/09/14 06:09:59 $

if (nargin == 2)  %...  Case  gcd(a,b)
% Do scalar expansion if necessary
sza = size(a) ;  szb = size(b) ;  %...  Matlab 3.5 - 6.5 compatible
if (sza == 1),   a = a*ones(szb(1),szb(2)) ;  %...  "  "  "
  elseif (szb == 1),   b = b*ones(sza(1),sza(2)) ;  end

sza = size(a) ;  if  any(sza - size(b))
    error('Arrays input to  gcd(A,B)  must have the same size.')
  else
    a = a(:) ;  b = b(:) ;
  end;

if  any(round(a) ~= a)|any(round(b) ~= b)|any(imag(a))|any(imag(b))
    error('gcd(A,B) requires all inputs to be real integers.')
  end  %...  Inf  is deemed an integer,  but  NaN  is not.
```

```
if  (nargout < 2)  % ...  save time by omitting  c  and  d
    Y = [a, b]' ;  g = b ;  L = [0, 1; 1, 0] ;
    for  k = 1:length(a)
        x = Y(:,k) ;  %... = [a(k); b(k)]
        if any(isinf(x)),   g(k) = min(abs(x(:))) ;
        else   %...  finite operands
        while  x(2)  % ... ~= 0 ;  MOD(x(1),x(2))  and  REM(...)  could
            L(2,2) = -round(x(1)/x(2)) ; %  be wrong if  x(1)  is  huge
            x = L*x ;  % ...  new |x(2)| <= old |x(2)|/2
          end % ... of inner loop traversed fewer than  40  times
        g(k) = abs(x(1)) ;  end %...  of usual finite case
      end  % ... of  k
    g = reshape(g, sza) ;
    return
  end  % ...  of  Case  gcd(A,B)  with  nargout < 2

%  Case  [G,C,D] = gcd(A,B)  with  nargout == 3 ,  presumably.
Y = [a, b, b] ;  % ...  initialized to the right size
I = eye(2) ;  L = flipud(I) ;

for  k = 1:length(a)
    X = [I, Y(k,1:2)'] ; % ... = [1, 0, a(k);  0, 1, b(k)].
    if  isinf(X(1,3)),  X = flipud(X) ;  elseif ~isinf(X(2,3))
        while X(2,3)  % ... ~= 0  and everything is finite ...
            L(2,2) = -round(X(1,3)/X(2,3)) ;
            X = L*X ; % ... new |X(2,3)| <= old |X(2,3)|/2
          end  % ... of inner loop traversed fewer than  40  times.
      end  % ...  of finite  a(k)  and  b(k)
    if  (X(1,3) < 0),  X = -X ;  end  % ...  invert  g(k) < 0 .
    Y(k,:) = X(1,:) ;
  end  % ...  of  k

g = reshape(Y(:,3), sza) ;
c = reshape(Y(:,1), sza) ;
d = reshape(Y(:,2), sza) ;
return
%  end of Case  [G,C,D] = gcd(A,B)

elseif  (nargin == 1)  %...  Case  gcd(A)  treated recursively
if (nargout > 1)
    error('G = gcd(A)  has just one output.'),  end
g = a(1,:) ;  [isr, isc] = size(a) ;
for  k = 2:isr,  g = gcd(g, a(k,:)) ;  end
return
%  end of  Case gcd(A)

else  error('gcd(A,B)  accepts just one or two arguments.')

%  For  Matlab 3.5,  isinf(x) = ~( finite(x)|isnan(x) ) ,  and
%  retrofitted  reshape(X, size(...))  works.  And for
%  386-Matlab 3.5 & PC-Matlab 4.2,  use  r0und  instead of buggy  round .
end  %...  of  gcd

%  ========================================================================
```

```
function L = lcm(a,b,x)
%LCM    Least Common Multiple,  with optional correctness test.
%   L = lcm(A,B) = lcm(abs(A), abs(B)) >= 0  is an array of  Least
%   Common Multiples  of corresponding elements of integer arrays
%   A and B.  They must have the same size unless one is a scalar.
%   WARNING:  Roundoff may have spoiled  L  wherever  L >= 2/eps .
%
%   L = lcm(A,B,x)  substitutes the scalar  x  for any element of
%   L >= 2/eps  that fails an optional appended correctness test.
%   Among plausible choices  x  are  0, Inf  and  NaN,  depending
%   upon how  lcm's  user will respond to these error-indicators.
%
%   Alas,  some errors can evade detection by the test.  It works
%   best when  Matlab  accumulates matrix products either with
%   Fused Multiply-Adds,  as it does on  Power Macs,  or else
%   extra-precisely as do versions 3.5-5.2 on  680x0-based Macs,
%   and versions 3.5-4.2 on a PC,  and version 6.5 on a PC after
%   it executes the command  system_dependent('setprecision',64).
%   Then  lcm(A,B,x)  should detect any erroneous  L < 2048/eps .
%
%   L = lcm(A)  is a row whose every element is the  LCM  of the
%   corresponding column of the array  A  of integers.   WARNING:
%   Wherever  L >= 2/eps  roundoff may make  L  utterly erroneous
%   though  lcm(A)  tries to substitute  Inf  for each such error
%   unless aborted by a  NaN  produced by  lcm(0,Inf) .  Wherever
%   lcm(flipud(A))  differs from  lcm(A) ,  both may be wrong.
%
%   Requires  gcd(...)  as modified by  W.K.  after  1990.
%                                  W. Kahan,  1990 - 14 Sept. 2008

if any(imag(a(:)))
    error('lcm(A,...)  accepts no complex argument.'),  end
a = abs(a) ;
if (nargin > 1)  %...  Cases  lcm(a,b)  and  lcm(a,b,x)
if any(imag(b(:)))
    error('lcm(A,B,...)  accepts no complex argument.'),  end
b = abs(b) ;
% Do scalar expansion if necessary
sza = size(a) ;  szb = size(b) ;  %...  Matlab 3.5 - 6.5 compatible
if (sza == 1),   a = a(ones(szb(1),szb(2))) ;
  elseif (szb == 1),   b = b(ones(sza(1),sza(2))) ;  end

%  Gcd(A,B)  will expose other erroneous inputs,  namely ...
%     input arrays  A  and  B  of different sizes,  or
%     any element in  |A|  or  |B|  not an integer.
%     Gcd  deems  Inf  an integer,  but not  NaN .

g = gcd(a,b) ;  g = g+(g==0) ;  Lg = isinf(g) ;
if any(Lg(:)),  g(Lg) = Lg(Lg) ;  end  %...  lcm(inf, inf) = inf .
a = a./g ;  L = a.*b ;
if (nargin == 2),  return,  end  %...  of  Case  lcm(a,b)

%  Case  lcm(a,b,x)'s test:
if (L(~Lg) < 2/eps),  return,  end  %...  no further test needed
if (length(x(:)) ~= 1),  x = x
  error('x  in  lcm(A,B,x)  must be a scalar,  not array.'),  end
```

```
%   What follows substitutes poorly for  IEEE 754's INEXACT flag:
g = g(:) ;  b = b(:)./g ;  a = a(:) ;  Lg = isinf(a)|isinf(b) ;
[m,n] = size(L) ;  mn = m*n ;
L = L(:) ;  q = round(L./g) ;
for  j = 1:mn  %...  seek erroneous finite  L(j)  only where ...
  if ~Lg(j)    %...  both  a(j)  and  b(j)  are finite:
    if ( [L(j), q(j)]*[-1; g(j)]~=0 ),  L(j) = x ;  %...  L  is wrong
    elseif ( [q(j), a(j)]*[-1; b(j)]~=0 ),  L(j) = x ;  end %... " "
   end, end  %...  of finite  a(j)  and  b(j) ,  and of  j
L = reshape(L, m,n) ;  return
end  %...  of  Case  lcm(a,b,x)

%  Case  lcm(A)  treated tail-recursively:
L = a(1,:) ;  [isr, isc] = size(a) ;
for  k = 2:isr ,  L = lcm(L, a(k,:), Inf) ;  end
%  end of  Case lcm(A)

%  For  Matlab 3.5,  isinf(x) = ~( finite(x)|isnan(x) ) .  For
%  386-Matlab 3.5,  use  W.K's  r0und  instead of buggy  round .

%  ========================================================================
```

The built-in function `round(x)` in 386-Matlab 3.5k and in PC-Matlab 4.2 has a bug:

If odd integer $|x| > 1/\text{eps} = 2^{52}$ (in which case $|x| < 2^{53}$ too)
then `round(x) - x == sign(x)` instead of 0 .

This bug is fixed, albeit slowly, by `r0und.m` :

```
function  y = r0und(x)
%R0UND  Round to a nearest integer;  ONLY  for PCs Matlab 3.5 & 4.2
%  r0und(x) = integer "nearest"  x ,  fixing a bug in  round.m:
%  386-Matlab 3.5's  and  PC-Matlab 4.2's  buggy  round(x)  yields
%  x + sign(x)  whenever  odd  |x| > 2^52  (and therefore
%  |x| < 2^53  too).  This fixes  gcd.m,  lcm.m,  etc.
%                                    W. Kahan  22 Sept. 1997
y = round(x) ;
J = (abs(x) > 1/eps) ;
if any(J(:)),  y(J) = x(J) ;  end


%  ========================================================================
```

```
function  m = precn(n)
%PRECN senses precision carried while accumulating matrix products
%  m = precn(n)  uses only  add,  subtract and multiply operations
%  (no divisions)  upon a few artfully chosen integers  (no loops)
%  to sense the current precision  |m| sig. bits  in which  MATLAB
%  accumulates matrix multiplications of not too big dimensions
%  before storing the product in  53 sig. bits.  This  m  depends
%  upon the version  ( < 7)  of  MATLAB,  the computer's hardware,
%  and possibly some control bits in the processor,  as follows:
%  v. 3.5  on  PCs & 680x0-based Macs:  m = 24, 53 or 64 (default)
%  v. 4.2 - 5.2  on  680x0-based Macs:  m = 64
%  v. 5.2  on  Power-Macs & iMacs (Fused Multiply-Adds):  m = -53
%  v. 4.2  on  PCs:  m = 64
%  v. 5.3  on  PCs & every v. on SUN SPARCS:  m = 53
%  v. 6.5  on  PCs:  m = 24, 53 (default) or 64
%
%  Argument  n  of  precn(n)  can be omitted and is ignored except
%  by versions of  precn  running on  PCs  under  MATLAB 3.5 & 6.5
%  with appropriate leading  "%"  characters deleted from the file.
%  Then  m = precn(n)  is determined after the precision is reset
%  to  n  by an invocation appropriate to the version.
%                            Revised 21 Sept. 2008.     W. Kahan

if (nargin == 0),  n = 0 ;  else

%  For  386-Matlab 3.5  on  PCs  delete leading  "%"  in this block:
%      if (n == 64)                                                %:
%          ! CTRL87 33D
%      elseif (n == 53)                                            %:
%          ! CTRL87 23D
%      elseif (n == 24)                                            %:
%          ! CTRL87 03D
%      else  N = n                                                 %:
%          error(' precn(N)  requires  N = 24, 53 or 64 .')        %:
%        end                                                       %:

%  For  Matlab 6.5  on  PCs  delete leading  "%"  in this block:
%      if ~( (n==24)|(n==53)|(n==64) ),  N = n                 %:
%          error(' precn(N)  requires  N = 24, 53 or 64 .')    %:
%        end                                                   %:
%      system_dependent('setprecision', n)                     %:

  n = 1 ;  end  % ...  to impede compiler over-optimization

a = 409891 ;  ab = 2731 + n ;  a = [ab, -a]*[a; n] ;
b = 7623851 ;  ab = 1441 + n ;  b = [ab, -b]*[b; n] ;
ab = a*b ;  % ... = (2^65 + 1)/3  rounded to  min(m, 53) sig. bits

c = 2761 ;  d = 4051+n ;  cd = [c, -n]*[d; c] ;  %... = (2^25 + 1)/3

t11 = 2048 ;  t15 = 16*t11 ;  t40 = t15*t15*1024 ;  %... tk = 2^k

e = ([c, -c, a]*[(d-n)*t15; (d-n)*t40; b])*3 - t15 ;  %... = 1 ?
f = ([a, -c, c]*[b; (d-n)*t40; (d-n)*t15])*3 - t15 ;  %... = 1 ?
ef = [e, f] ;
```

```
if      (ef == [1, 1]),        m = 64  ;
elseif  (ef == [1, -t11]),     m = -53 ;
elseif  (ef == [-t11, -t11]),  m = 53  ;
elseif  (ef == [-t15, t40]),   m = 24  ;
else  Version = ver,  Machine = computer,  EF = ef
    error(' Why does  precn  malfunction on this machine?')
  end

%  ======================================================================
```

CTRL87.EXE was compiled from the following *Borland Turbo-Pascal* program:

```
{$R-,S-,I-,D-,T-,F-,V-,B-,N-,L+ }
{$M 1024,0,1024 }

      program ctrl87;
   {  CTRL87 <ctl<, msk>>  uses two 3-hex-digit parameters  ctl and msk
        to set as many as 9 bits in the  ix87  Control-Word  as follows:
               New CW  :=  (msk AND ctl) OR (NOT(msk) AND Old CW) .
        If  msk  is omitted,  0F00  is used in its place.  If both  msk
        and  ctl  are omitted,  or if either is  " ? "  or not hexadecimal
        they will be prompted from the keyboard after the display of
        DOC  below,  which explains how they affect subsequent floating-
        point arithmetic operations.  To do nothing,  [Enter]  nothing.

        To prevent mishaps,  msk  is filtered thus:  msk := msk AND 0F3D
      ====================================================================
      }
      const
         n = 19 ;   {  n = current number of lines in  DOC  }
         DOC: array[1..n] of string[55] = (
            '  CTRL87 <ctl<, msk>>  sets the  ix87  Control-Word ',
            '  C-W := (msk AND ctl) OR (NOT(msk) AND C-W)  from 2',
            '  3-hex-digit parameters  ctl and msk .  C-W''s bits ',
            '  are  OR''d  to affect floating-point thus:     C-W ',
            '  TRAPS:  (default)   Disable All traps    ...  _3D ',
            '   or     Disable  trap for  INVALID OP    ...  _01 ',
            '     and  Disable  trap for  DIV by ZERO   ...  _04 ',
            '     and  Disable  trap for  OVERFLOW      ...  _08 ',
            '     and  Disable  trap for  UNDERFLOW     ...  _10 ',
            '     and  Disable  trap for  INEXACT       ...  _20 ',
            '  PRECISION:  (default) Round to  REAL*10  ...  3__ ',
            '               or else  Round to  REAL*8   ...  2__ ',
            '               or else  Round to  REAL*4   ...  0__ ',
            '  DIRECTION:  (default) Round to  Nearest  ...  0__ ',
            '               or else  Round     Down     ...  4__ ',
            '               or else  Round     Up       ...  8__ ',
            '               or else  Round to  Zero     ...  C__ ',
            '  Initial Control-Word  ctl  set by  FINIT ...  33D ',
            '  Default msk = 0F00 .  Maximal effective msk = F3D '  );
      Sctl = '  Current setting of  Control-Word  ctl    ... ' ;
      S3H =  '  Enter  3  hex digits for ' ;
      msx = $0F3D ;  { ... maximal  msk }

      var
         ctl, i, j, k, L, msk : word ;   s : string ;

       function  Wrd2Str( i : word ) :  string ;
```

```
    { ...  converts word  i  to its string of  4  hex digits.}
    var  j, k : word ;  s : string[4] ;
    begin
      s := '' ;
      for  k := 0 to 3 do begin
        j := i AND $F ;
        i := i shr 4 ;
        if j > 9 then  j := j + $37
                 else  j := j + $30 ;
        s := Concat( Chr(j), s ) ;
        end ; {  k  }
      Wrd2Str := s
    end; { Wrd2Str }

  procedure  GetHex( var j, k : word;  s : string );
    begin   { converts string  s  to  4-hex-digit word  j }
       Val( ConCat('$',s), j, k ) ; { j = value of $s if  k = 0 }
       if  k > 0  then  Writeln(s, '  is not hexadecimal.') ;
       end ; { GetHex }

  begin
    inline( $9B/$D9/$3E/i/$9B ) ;  {  fstcw  i ;  old Control-Word }
    L := ParamCount ;
    if  L = 0  then  k := 1  else begin
      s := ParamStr(1) ;  { = first parameter on DOS command line }
      if  Copy(s,1,1) = '?'  then  k := 1  else  GetHex(ctl, k, s) ;
      if  k = 0  then
        if  L < 2  then  msk := $0F00
                   else  GetHex(msk, k, ParamStr(2)) ;
      end ;  { L > 0 }

    while  k > 0  do begin { Prompt for  ctl  and  msk .}
       for  j := 1 to n  do  Writeln( DOC[j] ) ;
       Writeln( Sctl, Wrd2Str( i AND msx ) ) ;
       Writeln( S3H, 'new  ctl :' ) ;
       Readln(s) ;
       if  (s = '') or (s = ' ') or (s = '  ')
              then  Exit ;  { Do nothing.}
       if Copy(s,1,1) = '?' then  k := 1  else  GetHex(ctl, k, s) ;
       if  k = 0  then  { Prompt for  msk .}
         repeat
           Writeln( S3H, ' msk  or accept  0F00 :' ) ;
           Readln(s) ;
           if  (s = '') or (s = ' ') or (s = '  ')
                                       then  msk := $0F00
                                       else  GetHex(msk, k, s) ;
          until  k = 0 ;  { Prompted for  msk .}
        L := 0 ;
       end ;  { Prompted values for  ctl  and  msk .}

    msk := msk and msx ;  { Don't change  8087 vs. 387  C-W .}
    ctl := (msk and ctl) or ((not msk) and i) ;
    inline( $9B/$D9/$2E/ctl/$9B ) ;  {  fldcw  ctl  }
    if  L = 0  then  Writeln( Sctl, Wrd2Str( ctl AND msx ) ) ;
  end.
{ ================================================================== }
```

```
% gcdtest.m is a Matlab script to test versions of gcd.m
format compact, format long g
diary gcdtest.txt
disp ('  GCDtest puts its results into GcdTest.txt')
DateTime = round(clock)
Machine = computer, ver
MatMultPrecn = precn
disp( '   ')
disp ('  1st test:  G = gcd(A, B)  and  gcd(B, A) :')

p = 28059810762433 ;   %...  a prime
t53 = 2^53 ;
A = [ 77,  77,  77,   0;    3,     15, t53,  t53-1 ] ;
B = [132,   0, Inf, Inf; 2^80, 2+2^52,   p,     p  ] ;
G = [ 11,  77,  77,   0;    1,      3,   1,     1  ] ;
G0 = gcd(A, B) ;
G1 = gcd(B, A) ;
K = (G ~= G0)|(G ~= G1) ;
if ~any(K(:))
    disp('  1st test passed.')
  else  %...  when matrix mult'n is not accumulated extra-precisely ...
    disp('  1st test failed in these cases:')
    F = [A(K)'; B(K)'; G0(K)'; G1(K)'; G(K)'] ;
    disp('  A;B;gcd(A,B);gcd(B,A);trueGCD = '), F
  end  %... of 1st test

disp('   ')
disp('  2nd test:  [G, C, D] = gcd(A,B)  and  gcd(B, A) :')
f80 = 2^40 ;
f80 = ((f80 - 1)/3)*(f80 + 1) ;  %... = (2^80 - 1)/3  rounded
A = [ 77, 77,  77,   0;    3,            -15, t53,        t53-1     ] ;
B = [132,  0, Inf, Inf; 2^80,          2+2^52,   p,           p      ] ;
G = [ 11, 77,  77,   0;    1,               3,   1,           1      ] ;
C = [  7,  1,   1,   1; -f80, 300239975158033,  -1,  -14029905381217 ] ;
D = [ -4,  0,   0,   0;    1,               1, 321, 4503599627370656 ] ;
[G0, C0, D0] = gcd(A, B) ;
[G1, C1, D1] = gcd(B, A) ;
K = (G0~=G)|(G1~=G)|(C0~=C)|(C1~=D)|(D0~=D)|(D1~=C) ;
if ~any(K(:))
    disp('  2nd test passed.')
  else  %...  when matrix mult'n is not accumulated extra-precisely ...
    disp('  2nd test failed in these cases:')
    F = [A(K)'; B(K)'; G0(K)'; C0(K)'; D0(K)'; G1(K)'; C1(K)'; D1(K)'] ;
    disp(' A;B;[gcd(A,B);C0;D0];[gcd(B,A);C1;D1];trueGCD;C;D = ')
    F = [F; G(K)'; C(K)'; D(K)']
  end  %... of 2nd test

disp('   ')
disp('  3rd test:  G = gcd(E)  and  gcd(flipud(E)) :')
g = [1, 3, 11, 17] ;  e = [77; 132; 144] ;  E = e*g ;
E = [E, [124+t53; t53-19; t53+254]] ;  g = [g, 13] ;
g0 = gcd(E) ;  g1 = gcd(flipud(E)) ;
k = (g0~=g)|(g1~=g) ;
if ~any(k(:))
```

```
    disp('  3rd test passed.')
  else  %...  when matrix mult'n is not accumulated extra-precisely ...
    disp('  3rd test failed in these cases:')
    disp('  E;gcd(E):gcd(flipud(E));trueGCD = ')
    F = [E(:,k); g0(k); g1(k); g(k)]
  end  %... of 3rd test
disp('  '),  disp('  ')

%  ====================================================================
```

The following **GCDtest results** were obtained from an IBM T21 laptop running MS Windows 2000, and were replicated on a Dell Optiplex running MS Windows XP. As expected, the tests failed when run with the default `MatMultPrecn = 53` and passed with `MatMultPrecn = 64`.

As expected, the failures for `MatMultPrecn = 53` were replicated by PC MATLAB 5.3, and the passed tests for `MatMultPrecn = 64` were replicated by PC MATLABs 3.5 and 4.2, and by Mac MATLABs 3.5, 4.2 and 5.2 on a 68040-based Mac Quadra 950. Passed tests were replicated also by Mac MATLAB 5.2 with `MatMultPrecn = -53` revealing *Fused Multiply-Adds* on a Power Mac 8600 and on an iMac.

```
  GCDtest  puts its results into  GcdTest.txt
DateTime =  2008  9  14          14:16:15
Machine = PCWIN
--------------------------------------------------------------------------
MATLAB                                  Version 6.5        (R13)
MatMultPrecn =    53

  1st test:  G = gcd(A, B)  and  gcd(B, A) :
  1st test failed in these cases:
  A;B;gcd(A,B);gcd(B,A);trueGCD =
F =
                         3     9.00719925474099e+015
     1.20892581961463e+024            28059810762433
                         3            28059810762433
                         3            28059810762433
                         1                         1

  2nd test:  [G, C, D] = gcd(A,B)  and  gcd(B, A) :
  2nd test failed in these cases:
 A;B;[gcd(A,B);C0;D0];[gcd(B,A);C1;D1];trueGCD;C;D =
F =
                         3     9.00719925474099e+015     9.00719925474099e+015
     1.20892581961463e+024            28059810762433            28059810762433
                         3            28059810762433                         1
                         1                         0                        -1
                         0                         1                       321
                         3            28059810762433                         1
                         0                         1                       321
                         1                         0                        -1
                         1                         1                         1
    -4.02975273204876e+023                        -1          -14029905381217
                         1                       321     4.50359962737066e+015
```

```
  3rd test:  G = gcd(E)  and  gcd(flipud(E)) :
  3rd test failed in these cases:
  E;gcd(E):gcd(flipud(E));trueGCD =
F =
     9.00719925474112e+015
     9.00719925474097e+015
     9.00719925474125e+015
                        1
                       13
                       13
```

```
system_dependent('setprecision', 64)
gcdtest
  GCDtest  puts its results into  GcdTest.txt
DateTime =  2008  9  14          14:16:52
Machine = PCWIN
--------------------------------------------------------------------------------
MATLAB                                      Version 6.5       (R13)
MatMultPrecn =    64

  1st test:  G = gcd(A, B)  and  gcd(B, A) :
  1st test passed.

  2nd test:  [G, C, D] = gcd(A,B)  and  gcd(B, A) :
  2nd test passed.

  3rd test:  G = gcd(E)  and  gcd(flipud(E)) :
  3rd test passed.


quit


%  ================================================================================
```

```
% lcmtest.m is a Matlab script to test W.K.'s version of lcm.m
format compact,  format long g
diary  lcmtest.txt
disp ('  lcmtest  puts its results into diary  lcmtest.txt')
DateTime = round(clock)
Machine = computer,  ver
MatMultPrecn = precn
disp( '   ')

disp ('  1st test:  L = lcm(A, B)  and  lcm(B, A) :')
p = 28059810762433 ;  i = 712176643 ;  %... primes
j1 = 2203 ;  j2 = 5741 ;  %... primes
k1 = 2205 ;  k = k1*j2 ;  j = j1*j2 ;
t52 = 2^52 ;  t53 = 2*t52 ;  t80 = 2^80 ;
A = [ 77,    j,   i,   i;     3,       15,   t53,   t53-1 ] ;
B = [132,    k,   j,   k;   t80,    2+t52,     p,       p ] ;
L = [924, j*k1, i*j, i*k; 3*t80, 10+5*t52, p*t53, t53*p-p ] ;
L0 = lcm(A, B) ;
L1 = lcm(B, A) ;
K = (L ~= L0)|(L ~= L1) ;
if ~any(K(:))
    disp('  1st test passed.')
  else  %...  when matrix mult'n is not accumulated extra-precisely ...
    disp('  1st test failed in these cases:')
    F = [A(K)'; B(K)'; L0(K)'; L1(K)'; L(K)'] ;
    disp('  A;B;lcm(A,B);lcm(B,A);trueLCMrounded  = '), F
  end  %... of 1st test
disp('   ')

disp('  2nd test:  L = lcm(A, B, Inf)  and  lcm(B, A, Inf) :')
A = [ 77,    j,   i,   i;     3,       15,   t53,   t53-1 ] ;
B = [132,    k,   j,   k;   t80,    2+t52,     p,       p ] ;
L = [924, j*k1, i*j, Inf; 3*t80,      Inf, p*t53,     Inf ] ;
L0 = lcm(A, B, Inf) ;
L1 = lcm(B, A, Inf) ;
K = (L ~= L0)|(L ~= L1) ;
if ~any(K(:))
    disp('  2nd test passed.')
  else  %...  when matrix mult'n is not accumulated extra-precisely ...
    disp('  2nd test failed in these cases:')
    F = [A(K)'; B(K)'; L0(K)'; L1(K)'; L(K)'] ;
    disp('  A;B;lcm(A,B,Inf);lcm(B,A,Inf);expectedLCMchecked  = '), F
  end  %... of 2nd test
disp('   ')

disp('  3rd test:  L = lcm(E)  and  lcm(flipud(E)) :')
g = [1, 3, 11, 17] ;  e = [77; 132; 144; 11088] ;  E = e*g ;
l = E(4,:) ;  E = E(1:3,:) ;
E = [E, [i, j; j, i; k, k]] ;  l = [l, Inf, Inf] ;
l0 = lcm(E) ;  l1 = lcm(flipud(E)) ;
k = (l0~=l)|(l1~=l) ;
if ~any(k(:))
    disp('  3rd test passed.')
  else  %...  when matrix mult'n is not accumulated extra-precisely ...
```

```
    disp('  3rd test failed in these cases:')
    disp('  E;lcm(E):lcm(flipud(E));expectedLCM = ')
    F = [E(:,k); l0(k); l1(k); l(k)]
  end  %... of 3rd test
disp('  ')

disp('  4th test:  lcm(0, x),  lcm(Inf, x)  and  lcm(Inf, 0) :')
a = [  77,  0, Inf, 0, Inf, Inf ] ;
b = [ 132, 77,  77, 0, Inf,   0 ] ;
L = [ 924,  0, Inf, 0, Inf, NaN ] ;
L0 = lcm(a,b) ;  L1 = lcm(b,a) ;  L2 = lcm(a,b,99) ;
K = [0,0,0,0,0, isnan(L0(6))&isnan(L1(6))&isnan(L2(6))] ;
K = ((L0==L)&(L1==L)&(L2==L))|K ;
if all(K(:))
    disp('  4th test passed.')
  else
    disp('  4th test failed in these cases:')
    disp('  a;b;lcm(a,b);lcm(b,a);lcm(a,b,99);expectedLCM = ')
    F = [a(~K); b(~K); L0(~K); L1(~K); L2(~K); L(~K)]
  end  %... of 4th test
disp('  '),  disp('  ')

%  ========================================================================
```

The following **LCMtest results** were obtained from an IBM T21 laptop running MS Windows 2000, and were replicated on a Dell Optiplex running MS Windows XP. As expected, the tests failed when run with the default `MatMultPrecn = 53` and passed with `MatMultPrecn = 64`.

As expected, the failures for `MatMultPrecn = 53` were replicated by PC MATLAB 5.3, and the passed tests for `MatMultPrecn = 64` were replicated by PC MATLABs 3.5 and 4.2, and by Mac MATLABs 3.5, 4.2 and 5.2 on a 68040-based Mac Quadra 950. Passed tests were replicated also by Mac MATLAB 5.2 with `MatMultPrecn = -53` revealing *Fused Multiply-Adds* on a Power Mac 8600 and on an iMac.

```
  lcmtest  puts its results into diary  lcmtest.txt
DateTime =  2008  9  14          14:18:04
Machine = PCWIN
------------------------------------------------------------------------------
MATLAB                                    Version 6.5       (R13)
MatMultPrecn =    53

  1st test:  L = lcm(A, B)  and  lcm(B, A) :
  1st test failed in these cases:
  A;B;lcm(A,B);lcm(B,A);trueLCMrounded  =
F =
                         3     9.00719925474099e+015
     1.20892581961463e+024            28059810762433
     1.20892581961463e+024     9.00719925474099e+015
     1.20892581961463e+024     9.00719925474099e+015
     3.62677745884389e+024      2.5274030658756e+029

  2nd test:  L = lcm(A, B, Inf)  and  lcm(B, A, Inf) :
  2nd test failed in these cases:
```

```
  A;B;lcm(A,B,Inf);lcm(B,A,Inf);expectedLCMchecked  =
F =
  Columns 1 through 3
                         3     9.00719925474099e+015                 712176643
     1.20892581961463e+024            28059810762433                  12658905
     1.20892581961463e+024                       Inf     9.01537646695592e+015
     1.20892581961463e+024                       Inf     9.01537646695592e+015
     3.62677745884389e+024      2.5274030658756e+029                       Inf
  Column 4
     9.00719925474099e+015
            28059810762433
      2.5274030658756e+029
      2.5274030658756e+029
                       Inf

  3rd test:  L = lcm(E)  and  lcm(flipud(E)) :
  3rd test failed in these cases:
  E;lcm(E):lcm(flipud(E));expectedLCM =
F =
                 712176643                  12647423
                  12647423                 712176643
                  12658905                  12658905
     1.98608743567039e+019     1.98608743567039e+019
     1.98608743567039e+019     1.14021279681837e+023
                       Inf                       Inf

  4th test:  lcm(0, x),  lcm(Inf, x)  and  lcm(Inf, 0) :
  4th test passed.


system_dependent('setprecision', 64)
lcmtest
  lcmtest  puts its results into diary  lcmtest.txt
DateTime =  2008  9  14          14:18:54
Machine = PCWIN
------------------------------------------------------------------------------
MATLAB                                      Version 6.5        (R13)
MatMultPrecn =    64

  1st test:  L = lcm(A, B)  and  lcm(B, A) :
  1st test passed.

  2nd test:  L = lcm(A, B, Inf)  and  lcm(B, A, Inf) :
  2nd test passed.

  3rd test:  L = lcm(E)  and  lcm(flipud(E)) :
  3rd test passed.

  4th test:  lcm(0, x),  lcm(Inf, x)  and  lcm(Inf, 0) :
  4th test passed.

quit
```