Four Cholesky Factors of Hilbert Matrices and their Inverses

From notes for a 2nd undergraduate *Numerical Analysis* course by
Prof. W. Kahan
Math. Dept., and Elect. Eng. & Computer Sci. Dept.
University of California at Berkeley

Abstract

Numerical software for matrix inversion and factorization is often tested upon *Hilbert* matrices because, first, they are so nearly singular, the more so as dimensions increase, and secondly their determinants, inverses and *Cholesky* factors are computable accurately for comparison purposes from relatively simple integer recurrences embedded hereunder in MATLAB programs. The N-by-N Hilbert matrices $H_{N,K}$ treated here have elements $\{H_{N,K}\}_{ij} = 1/(i+j+K-1)$ for integers $K \geq 0$. Examples pit these matrices' inverses and factors against MATLAB's `inv`, `chol` and `eig` functions. Correctness tests turn out much more arduous than the programs tested.

Contents

Introduction

These are lengthy notes.

If you would rather not read them, you should demand and purchase programming languages, environments and software development systems that support extravagantly wide precision for floating-point arithmetic *as its default*. The alternative is to pay occasionally for extra time and extraordinary mathematical talent that succeeds less often than extravagantly wide precision can at delivering computed results at least about as accurate as your data deserve. IEEE Standard 754 (2008) for Floating-Point Arithmetic offers arithmetic 16 bytes wide, with at least about 32 sig. dec. It is extravagant enough to render the effects of roundoff almost surely insignificant.

We should prefer rounding errors so tiny that their effects need not be appraised. Instead they can degrade results computed nowadays by programs otherwise impeccably correct algebraically. When degradation is intolerable, what (if not whom) shall we blame? If not the program then the data. We call a blameworthy program "Numerically Unstable". We call blameworthy data "Ill-Conditioned". Such designations are oversimplifications often undeserved, as we shall see.

At least since version 6.5, MATLAB's programs for its functions `inv`, `chol` and `eig` have been *State of the Art* despite known rare failure modes that deliver misleading results for otherwise innocuous data. (Some examples of `inv`'s failures appear in lecture notes posted on my web page at <`www.eecs.berkeley.edu/~wkahan/Math128/FailMode.pdf`>.) Those failures are not the subject of the following notes, which are not intended to disparage MATLAB's functions in question. These are designed for matrices whose elements are uncorrelatedly uncertain by at least a unit in the last sig. bit retained, as if rounded off when stored in the computer's memory.

Uncertainty in data propagates, amplified by appropriate *Condition Numbers*, to uncertainty inherited in the inverses, Cholesky factors and eigenvalues that `inv`, `chol` and `eig` would produce if executed with infinitely precise arithmetic — no roundoff. Ideally, numerically stable software would let its roundoff add little more uncertainty than must be inherited anyway; but practical software falls short of this ideal. Different numerically stable programs differ in their degradation by roundoff; some programs suffer more roundoff than others do when dimensions grow; some programs tolerate bigger condition numbers before aborting. We explore programs' limitations by applying them to test data of increasing dimensions and worsening ill-condition.

Condition numbers of Hilbert matrices $H_{N,K}$ are known to grow rapidly (exponentially) with their dimension N. They serve as test data for `inv`, `chol` and `eig` because their inverses, Cholesky factors and related eigenvalues are computable accurately from exact integer input data (N, K) without first computing and rounding off the Hilbert matrices' elements. These accurate schemes are early instances of a growing body of "structure-preserving" algorithms that bypass explicit computations of a matrix whose gross ill-condition is an accident of the choice of one of many possible mathematical formulations of a problem with structured data. Preserving this structure faithfully often preserves also a benign relationship between the problem's solution and its data; then the ill-condition of that matrix is irrelevant to the benign relationship.

For instance, a problem in which Hilbert matrices can appear but needn't is *Least-Squares* approximation of any given function $y(\tau)$ over the interval $0 \le \tau \le 1$ by polynomials. Choosing to represent these as linear combinations $\Xi(\tau) := \sum_{1\le j\le N} \xi_j \cdot \tau^{j-1}$ of power functions τ^{j-1} is what

brings in Hilbert matrices. Their ill-condition reflects the maximal growth rate with increasing degree $N–1$ of the coefficients ξ_j of polynomials $\Xi(\tau)$ of magnitudes $|\Xi(\tau)|$ restricted by, say, $\|\Xi(\tau)\| \le 1$ on that interval. The growth is exponential for all of the usual norms $\|\ldots\|$. Choosing a better representation as a linear combination $\Xi(\tau) := \sum_{1\le j\le N} \chi_j \cdot G_{j-1}(\tau)$ of polynomials $G_{j-1}(\tau)$ orthogonal on that interval entails milder coefficients χ_j and matrices far better conditioned than Hilbert matrices when degree $N–1$ gets big. Thus does respect for the Least-Squares problem's structure pay off. But since these notes embrace rather than eschew Hilbert matrices, only a little more will be said later about those orthogonal polynomials $G_{j-1}(\tau)$.

Cleverness does not guarantee correctness. How can we know whether the MATLAB programs offered at the end of this document are accurate? We should prefer proofs. I cannot remember where I put my proofs a few decades ago; but I do remember that they were so much longer than the programs in question that the proofs' capture cross-section for error exceeded the programs' by far. That is why tests have been included, though they are lengthy too. Tests cannot prove accuracy, but they can corroborate it, or not. Examples of ineffective tests are included among the effective tests of triangular factors. Where explicit formulas for eigenvalues are unavailable, their tests here must be indirect but, alas, executable under only two old versions of MATLAB.

INTRODUCTION NOT FINISHED YET

Genesis of Hilbert Matrices

Hilbert matrices arise from (weighted) least-squares fitting of polynomials $\Xi(\tau)$ to arbitrary functions $y(\tau)$ over the interval $0 \le \tau \le 1$ as follows (using MATLAB's vector notation):

Given the function $y(\tau)$, choose integers $K \ge 0$ (for the *weight*) and $N > 0$ (for the *degree*), and then find the column-vector $\mathbf{x} := [\xi_1; \xi_2; \ldots; \xi_N]$ of coefficients ξ_j of the unique polynomial $\Xi(\tau) := \sum_{1\le j\le N} \xi_j \cdot \tau^{j-1}$ of degree less than N that minimizes

$$||\Xi(\tau) - y(\tau)||^2 := \int_0^1 \tau^K \cdot (\Xi(\tau) - y(\tau))^2 \cdot d\tau \ .$$

Most often $K = 0$ and is then omitted. The minimizing column vector $\mathbf{x}$ satisfies the

$$\textit{Normal Equations} \qquad H_{N,K} \cdot \mathbf{x} = \mathbf{b}_{N,K}$$

in which column $\mathbf{b}_{N,K} := [\beta_1; \beta_2; \ldots; \beta_N]$ has elements $\beta_i := \int_0^1 \tau^{i-1+K} \cdot y(\tau) \cdot d\tau$, and matrix $H_{N,K}$ has elements $\theta_{ij} := \int_0^1 \tau^{i+j+K-2} \cdot d\tau = 1/(i+j+K-1)$ in row #i and column #j. This $H_{N,K}$ is an N-by-N *Hilbert* matrix. MATLAB functions to generate `H = hilb(N)` in the most common case $K = 0$ have been programmed by Dr. Cleve Moler (his comes with MATLAB) and by Prof. Nicholas J. Higham, but their programs are not the best to generate test data. Here is why:

Hilbert matrix $H_{N,K}$ becomes "ill conditioned" because it approaches singular matrices rapidly as N increases. Consequently, tiny perturbations that occur when quotients $1/(i+j+K-1)$ are rounded off cause the computed inverse of a perturbed $H_{N,K}$ to change drastically, and more so as N increases. Hypersensitivity to perturbations is explored in *Relevant Course Notes Posted on my Web Pages* listed below, and in Higham's book listed below under *Further Reading*.

To avoid perturbing the data, a program `hilbl(N, K)` supplied below computes $Y := L \cdot H_{N,K}$ for the least integer $L := \mathrm{LCM}([K+1, K+2, \ldots, 2N+K-1]) > 0$ whose quotients $L/(i+j+K-1)$ are all integers computed exactly, except when this L gets so big that it has to be rounded off.

Ideally, if N or K is too big for `hilbl(N,K)` to be computed accurately, the program should stop at an error message. It will after the command `system_dependent('setprecision', 64)` has been executed by PC MATLAB 6.5 to enable extra-precise accumulation of sufficiently small matrix products. Otherwise recent versions of MATLAB on some hardware may fail to discover when `hilbl(N,K)` is wrong. This is due to an unavoidable failure of `lcm` explained in my web page's course notes `<…/Math128/GCD5.pdf>` from which `gcd` and `lcm` were obtained. Such failures would not occur if MATLAB supported the *Inexact Exception Flag* mandated by IEEE Standard 754 for Floating-Point arithmetic. To preclude such failures, invoke `hilbl(N,K)` only with integers $N > 0$ and $K \ge 0$ that are not too big; these are tabulated on the next page.

Let $Y_{N,K}$ and $L_{N,K}$ denote the results produced exactly by `[Y, L] = hilbl(N,K)`, whence $H_{N,K} := Y_{N,K}/L_{N,K}$; this N-by-N matrix, with elements $1/(i+j+K-1)$ in its row #i and column #j, is a block out of a bigger Hilbert matrix H_{N+K}, and turns out to have only integer elements in its inverse. Program `invhilbl` below computes this inverse as accurately and as quickly as MATLAB can.

Tabulated under each listed K < 100 is the biggest N found to be not too big for `hilbl(N,K)` :

K	0	1	2	3	4	5	6	7	8	9
max N	21	21	21	20	20	19	19	18	18	17
K	10	11	12	13	14	15	16	17	18	19
max N	17	16	16	15	15	14	14	13	13	12
K	20	21	22	23	24	25	26	27	28	29
max N	12	11	11	10	10	11	10	10	9	9
K	30	31	32	33	34	35	36	37	38	39
max N	9	10	9	9	9	8	8	8	10	9
K	40	41	42	43	44	45	46	47	48	49
max N	9	9	8	8	8	7	8	8	7	7
K	50	51	52	53	54	55	56	57	58	59
max N	9	8	8	8	8	7	7	7	7	7
K	60	61	62	63	64	65	66	67	68	69
max N	7	7	7	7	6	7	6	6	7	7
K	70	71	72	73	74	75	76	77	78	79
max N	6	6	6	7	7	6	7	6	6	7
K	80	81	82	83	84	85	86	87	88	89
max N	6	6	6	6	6	6	6	7	7	6
K	90	91	92	93	94	95	96	97	98	99
max N	6	6	6	6	6	6	6	6	6	6

Computing Cholesky Factors and Determinants of $H_{N,K}$ and its Inverse :

In principle, $H_N^{-1} = L\cdot(L\cdot H_N)^{-1}$ could be computed using Matlab's `inv(…)` function, but this incurs rounding errors that cause at least about as much damage as would rounding off H_N's elements. To avoid that damage, MATLAB provides a special function `invhilb(N)` for use instead of `inv(hilb(N))` to get an accurate inverse by computing a diagonal matrix D of integers for which $H_N^{-1} = D\cdot H_N\cdot D$. The program `invhilbl(N,K)` below takes K into account too; it uses a recurrence derived from one first published by Dr. Sam Schechter in 1959.

Roundoff poses the same threat to `det(H)` as to `inv(H)` , and the threat is avoided the same way, namely by computing the diagonal matrix D of integers that figures in the formula $H^{-1} = D\cdot H\cdot D$ whence we get integers $\det(H^{-1}) = |\det(D)|$ and $\det(Y) = \det(L\cdot H) = |\det(L\cdot D^{-1})|$ from the MATLAB program `[dy, L, dhi] = dethilbl(N, K)` supplied below.

Upper-triangular Cholesky factors of $H_{N,K}$ and its inverse are computed with nearly minimal rounding errors from mostly integer formulas below more complicated to derive. These produce results far more accurate (unless N+K is small) than can be obtained from MATLAB's built-in `chol(hilbl(N,K))` and `chol(invhilbl(N,K))` .

Closed-form Formulas for Inverses and Triangular Factors of Hilbert Matrices

The element of N-by-N matrix $H_{N,K}$ in its row #i and column #j is $\{H_{N,K}\}_{i,j} := 1/(i+j+K-1)$. In most of our other other matrices the *Combinatorial Coefficients* ${}^{N}\text{Ç}_{K} := N!/((N-K)!\cdot K!)$ will be needed. In the formulas that follow, the subscripts N,K will be taken for granted so that the abbreviation H can be used for $H_{N,K}$ and similarly for all other matrices except $ and **u'** whose only subscript would be their dimension N . Here $:= Diag([1, –1, 1, –1, …, $(-1)^{N-1}$]) and row **u'** := [1, 1, 1, …, 1] . The N-by-N diagonal matrix D has integer elements

$$\{D\}_{j,j} := d_{N,K,j} := (-1)^j\cdot j\cdot {}^{N}\text{Ç}_{j}\cdot {}^{N+K+j-1}\text{Ç}_{N} .$$

$\{H^{-1}\}_{i,j} = \{D\cdot H\cdot D\}_{i,j} = d_{N,K,i}\cdot d_{N,K,j}/(i+j+K-1)$ also has elements all integers;

$\det(H^{-1}) = |\det(D)| = |\prod_j d_{N,K,j}|$ is an integer that grows huge very fast with N+K .

$\mathbf{u}'\cdot H^{-1}\cdot\mathbf{u} := \sum_i\cdot\sum_j \{H^{-1}\}_{i,j} = N\cdot(N+K)$ is useful to test the accuracy of H^{-1} .

$$\mathbf{u}'\cdot\$\cdot H^{-1}\cdot\$\mathbf{u} := \sum_i\cdot\sum_j |\{H^{-1}\}_{i,j}| = \left(4^{N-1}\cdot(N+K)^{2N-1}/((N-1)!)^2\right)\cdot\left(1 + O((N+K)^{-2})\right) \quad \text{as } K \to +\infty .$$

$$\|H^{-1}\|_F^2 := \sum_i\cdot\sum_j (\{H^{-1}\}_{i,j})^2 = \left((2N-2)!\cdot(N+K)^{2N-1}/((N-1)!)^4\right)^2\cdot\left(1 + O((N+K)^{-2})\right) \quad \text{as } K \to +\infty .$$

The four N-by-N upper-triangles U and R and their inverses are Cholesky factors of $H = U'\cdot U = R^{-1}\cdot R'^{-1}$ and of $H^{-1} = R'\cdot R = U^{-1}\cdot U'^{-1}$ and, though not generally integer matrices, can be assembled out of integer matrices starting with these four N-by-N diagonals:

$\{\$\}_{j,j} := (-1)^{j-1}$; $\{¥\}_{j,j} := K + 2j - 1$; $\{£\}_{j,j} := {}^{K+2j-2}\text{Ç}_{j-1}$; and $\{\Omega\}_{j,j} := {}^{K+N-1+j}\text{Ç}_{N-j}$.

These combine with two N-by-N integer-element upper triangles C and G defined by

$$\{C\}_{i,j} := {}^{2j-1+K}\text{Ç}_{j-i} \quad \text{and} \quad \{G\}_{i,j} := {}^{i+j-2+K}\text{Ç}_{j-i} = \{\$\cdot¥\cdot C^{-1}\cdot¥^{-1}\cdot\$\}_{i,j}$$

to produce the four Cholesky factors and inverses:

$U = \sqrt{¥}\cdot C\cdot(¥\cdot£)^{-1}$, $U^{-1} = £\cdot\$\cdot G\cdot\$\sqrt{¥}$, $R = \$\cdot\sqrt{¥}\cdot C\cdot\$\Omega = \$\cdot U\cdot D$ and $R^{-1} = (\Omega\cdot¥)^{-1}\cdot G\cdot\sqrt{¥}$.

MATLAB program `chohilbl` below computes U; `ichohilb` computes U^{-1}; `choihilb` computes R; and `ichihilb` computes R^{-1}.

All these formulas have been adapted more or less directly from formulas found in the literature cited under *Further Reading*. For big dimensions N these formulas seem at first to entail work proportional to N^3, but in fact H, U, R and their inverses can all be computed from recurrences that cost work proportional to N^2; these appear in our MATLAB programs. And most of the arithmetic, if precise enough, produces intermediate results computable exactly as integers. Of course, for any chosen precision, like MATLAB's 53 sig. bits, roundoff will corrupt at least some of those huge integers when N and/or K gets too big.

Also cited under *Further Reading* are formulas and programs more general than ours published by Prof. Plamen Koev for accurate triangular factors and singular values of *Cauchy* matrices, of which Hilbert matrices are instances.

Tests of `hilbl`, `invhilbl`, and `dethilbl`

These programs' own correctness has to be assessed before they are used to generate data to test other programs' accuracies. A test of `[Y, L] = hilbl(N, K)` confirmed that all the elements of `V = L./Y` were the correct small integers, the reciprocals of the elements of $H_{N,K}$, except when some of the biggest values of the scale factor L got rounded back to 53 sig. bits, in which case some of the entries in V differed from integers in their 53rd (last) sig. bit.

`W = invhilbl(N, K)` was tested by computing `norm([Y, L*eye(N)]*[W; -eye(N)])` to obtain $\| Y \cdot W - L \cdot I \|$ as one matrix product accumulated extra-precisely in 64 sig. bits on an old 68040-based Macintosh Quadra 950. Both norms vanished for small values of N and K but, beyond these, 64 sig. bits were too few to hold the intermediate products of elements of W and Y, so roundoff contaminated Y·W. Almost the same results were obtained from MATLAB 6.5 on an IBM PC after the command `system_dependent('setprecision', 64)`, without which roundoff contaminated results sooner and worse. Another test better indicative of (in)correctness in `invhilbl` on other computers was needed; it was constructed from the following observation:

Let row N-vector $\mathbf{u}' := [1, 1, 1, \ldots, 1]$; then $\sigma_{N,K} := \mathbf{u}' \cdot H_{N,K}^{-1} \cdot \mathbf{u} = N \cdot (N+K)$ after massive cancellation, the more so as N and K increase; and the computation of $\sigma_{N,K}$ generates no intermediate sums bigger in magnitude than the biggest elements of $H_{N,K}^{-1}$ because their signs alternate. If $\mathbf{u}' \cdot W \cdot \mathbf{u} \neq N \cdot (N+K)$ then surely $W \neq H_{N,K}^{-1}$. This is how roundoff's interference was inferred to be recorded in the documentation of `invhilbl`.

`[dy, L, dhi] = dethilbl(N, K)` was tested for a few small integers N and K by comparing its outputs with values of the determinants computed exactly by the automated algebra system DERIVE 4.1 run on an IBM PC. The outputs matched perfectly until they got so big that only their rounded values displayed by MATLAB to 15 sig. dec. could be compared; these matched in all but at worst the last digit displayed.

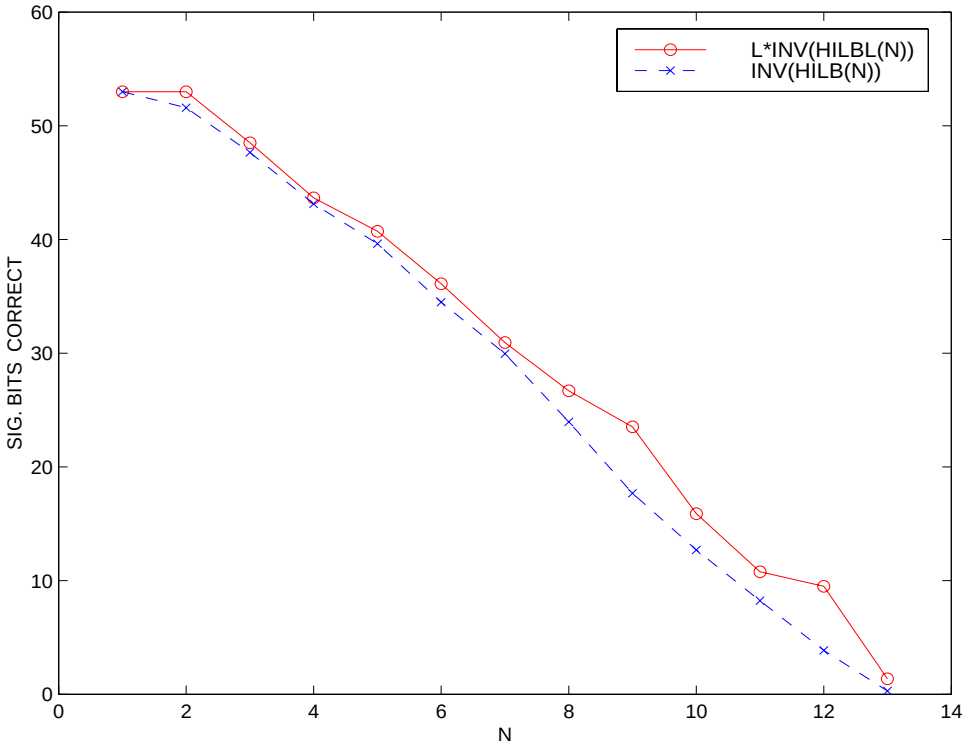
Tests of MATLAB's `inv`:

The accuracy of any estimate M of H_N^{-1}, no matter how MATLAB computes it, can now be assessed by comparing it with `W = invhilbl(N)` using arithmetic no more precise than was used to compute M. What measure of (in)accuracy is suitable? One possibility is *elementwise*:

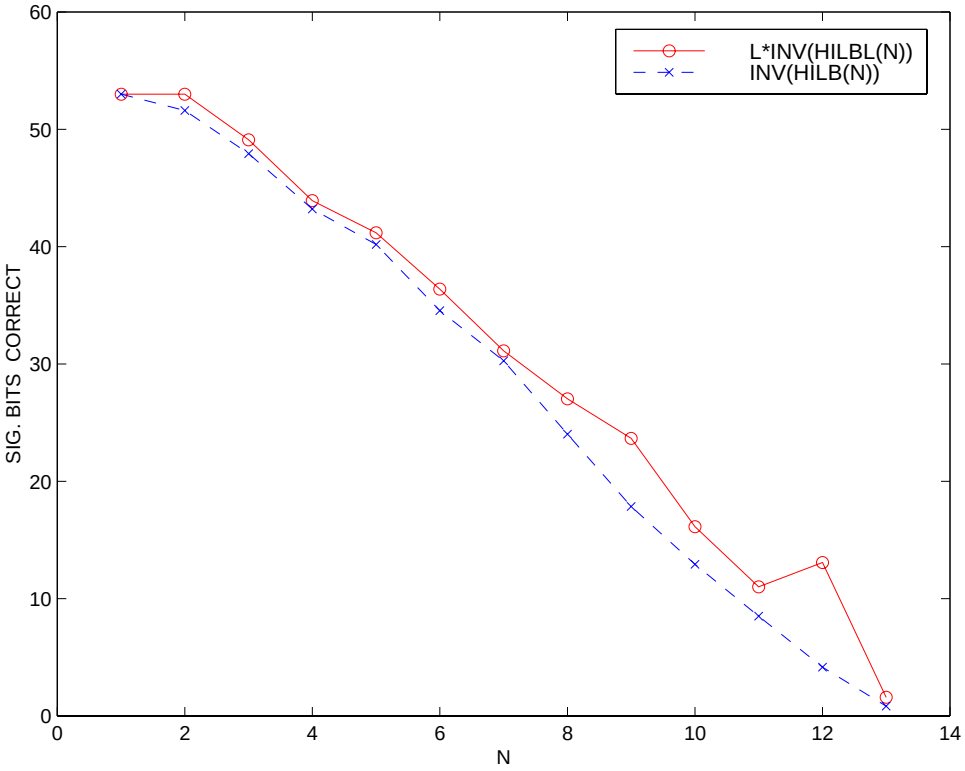
First compute `R = 2*(M-W)./(M+W)`, the array of symmetric differences from 1 of ratios of respective elements of M and W; then `m = -log(max( eps/2, max(abs(R(:))) ))/log(2)` is the least number m of matching sig. bits between respective elements of M and W. No more than 53 can match; this is why `eps/2` appears there. See the first graph below.

This elementwise measure m is appropriate for inverses of Hilbert matrices because they have no elements much closer to zero than their neighbors. There are matrices for which elementwise measures of (in)accuracy are inappropriate; for these a better measure may be *normwise*, using $r := 2 \cdot \|M-W\| / \|M+W\|$ in place of max|R(:)| above. The choice of norm ||…|| matters less for inverses of Hilbert matrices than for some others, in particular examples in course notes posted at `<…/Math128/FailMode.pdf>`.

Elementwise Accuracies of `L*inv(hilbl(N))` and `inv(hilb(N))`



Normwise Accuracies of `L*inv(hilbl(N))` and `inv(hilb(N))`



MATLAB's own biggest-singular-value `norm(…)` was chosen for the second graph above; it plots

```
-log(max( eps/2, 2*norm(M-W)/norm(M+W) ))/log(2)
```

against N for $1 \le N \le 13$, beyond which `hilb(N)` is too nearly singular for MATLAB's `inv(…)`.

The graphs corroborate that roundoff's effects inside `inv(…)` are comparable with the damage done by rounding the fractional elements of H_N to MATLAB's working precision of 53 sig. bits.

Exercise: Use tests like those above to compare with `invhilbl(N)` diverse estimates for H_N^{-1} computed by MATLAB from expressions like

```
inv(hilb(N)), round(inv(hilb(N)), L*inv(hilbl(N)), round(L*inv(hilbl(N))),
flipud(inv(fliplr(hilb(N))),  round(flipud(inv(fliplr(hilb(N)))), ...
```

for N = 1, 2, 3, … in turn to determine when each kind of estimate breaks down, and how much good or harm is done by different sources of roundoff, epilogs, column orders, … .

Exercise: Compare the output `dhi` of `[dy, L, dhi] = dethilbl(N)` with `1/det(hilb(N))` and its variations analogous to the previous exercise.

Variations Among Computed Inverses of a Near-Singular Matrix

Recall that the singular matrix S nearest H is distant from it by $||H–S|| = 1/||H^{-1}||$. This is proved as Theorem 7 in the course notes <…/MathH110/GIlite.pdf>. The norm ||…|| here is Matlab's `norm(…)` . Even a rough computation of H^{-1} reveals how near H is to singular; any estimate $M \approx H^{-1}$ provides an estimate $||H–S|| \approx 1/||M||$. How widely can estimates M vary?

Programs like MATLAB's `inv(…)` compute inverses by Gaussian Elimination or, equivalently, triangular factorization whose roundoff can cause `inv(H)` to differ from H^{-1} by a little more in norm than $(H+\Delta H)^{-1}$ can differ if $||\Delta H|| \approx N\cdot\varepsilon\cdot||H||$, though usually $(\texttt{inv(H)})^{-1} \ne H+\Delta H$ for any such tiny perturbation ΔH . Here $\varepsilon = \texttt{eps} = 2^{-52}$ is MATLAB's roundoff threshold. In other words, though a computed `M = inv(H)` need not be the inverse of any matrix $H+\Delta H$ differing from H by at most about N rounding errors in each element of H , the computed inverse M is almost never much farther from H^{-1} than the inverse $(H+\Delta H)^{-1}$ of such a perturbed H . And

$$(H+\Delta H)^{-1} – H^{-1} = –H^{-1}\cdot\Delta H\cdot(H+\Delta H)^{-1} , \quad \text{so}$$

$$||(H+\Delta H)^{-1} – H^{-1}|| \le ||H^{-1}||\cdot||\Delta H||\cdot||(H+\Delta H)^{-1}|| \approx N\cdot\varepsilon\cdot||H^{-1}||\cdot||H||\cdot||(H+\Delta H)^{-1}|| .$$

With very rare exceptions, an estimate M of H^{-1} obtained from `inv(…)` should satisfy roughly

$$||M – H^{-1}|| \le N\cdot\varepsilon\cdot||H||\cdot||M|| \quad \text{provided} \quad N\cdot\varepsilon\cdot||H|| << 1/||M|| \approx ||H–S|| .$$

The last proviso means " H is much farther than N rounding errors from its nearest singular matrix S ." "Rare exceptions" mean pathologies like those in <…/Math128/FailMode.pdf>.

But if ||H–S|| is not much bigger than $N\cdot\varepsilon\cdot||H||$, then `M = inv(H)` may well be computed far too inaccurately for the bound upon $||M – H^{-1}||$ above to be trusted. Usually Matlab issues a

```
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 3.572099e-17."
```

The displayed value RCOND ≈ 1/(||H||·||M||) is MATLAB's estimate of ||H–S||/||H|| , but it may be wrong by an order of magnitude because ||H+ΔH – S|| may vary by that much or more as ΔH varies by several rounding errors per element of H . Consequently $||(H+\Delta H)^{-1}|| = 1/||H+\Delta H - S||$ may vary that much, and therefore so may ||M|| , and consequently so may M , depending upon the rounding errors occurring during its computation.

Exercise: The last "consequently …" follows from the observation that the norm of a matrix can change by no more than the norm of its change: $|\,||M+\Delta M|| - ||M||\,| \le ||\Delta M||$. Can you explain why?

In short, if H is too nearly singular, different computations may yield estimates M of H^{-1} that appear utterly different. However, provided all the estimates are scarcely worse than $(H+\Delta H)^{-1}$ for roundoff-like perturbations ΔH , the diverse estimates M usually have this in common:

They are all approximately scalar multiples one of another.

This is explained on pp. 3-4 of the course notes `<…/MathH110/jacobi.pdf>` . An alternative explanation in terms of the *Singular Value Decomposition* of H+ΔH is more illuminating but requires enough additional technical machinery to be a story for another day.

Exercise: Use different formulas, like those in two previous exercises, to compute diverse estimates M_1, M_2, … for inverses of Hilbert matrices $H_{N,K}$ with N barely big enough to elicit MATLAB's "close to singular" warning. To see how nearly two such estimates, say M and W , come to scalar multiples of each other, estimate first a scalar multiplier ζ , say ζ := trace(W'·M)/trace(W'·W) , and then compute τ := || M – ζ·W ||/|| M + ζ·W || . If N is so big that τ is not much smaller than 1 , can you explain why? Reviewing the hypotheses from which the aforementioned note's conclusions were drawn may help; or else consider MATLAB's estimate of N – rank(H) .

Tests of Triangular Factors of $H_{N,K}$ and of $H_{N,K}^{-1}$

MATLAB's function `chol(…)` implements Cholesky's method, which can be used to compute upper-triangular factors U and R satisfying U'·U = H and R'·R = H^{-1} . No; R ≠ U^{-1} . Instead, R = $·U·D wherein integer diagonal D is the same as figures in the formula H^{-1} = D·H·D used by `invhilbl(…)` , and integer diagonal $:= Diag([1, –1, 1, –1, …, $(-1)^{N-1}$]) . See p. 6.

Assays of erosion by roundoff of `chol(…)`'s accuracy require accurate formulas for U , U^{-1}, R and R^{-1}. Despite that most of their elements are irrational, they can be computed in floating-point arithmetic from hitherto unpublished algorithms accurate in all but their last few sig. bits; this accuracy far surpasses `chol(…)`'s unless dimensions are small. These algorithms have been implemented below in MATLAB programs each of which produces an N-by-N upper triangle:

`U = chohilbl(N,K)` for U satisfying U'·U = H ; *cf.* `chol(hilbl(N,K))` .

`UI = ichohilb(N,K)` for U^{-1} satisfying UI·UI' = H^{-1} .

`R = choihilb(N,K)` for R satisfying R'·R = H^{-1} ; *cf.* `chol(invhilbl(N,K))` .

`RI = ichihilb(N,K)` for R^{-1} satisfying RI·RI' = H .

MATLAB's `chol(…)` issues a fatal error-message "`Matrix must be positive definite`" whenever its argument is indefinite or so nearly so that the factorization's rounding errors make the argument seem indefinite though they damage it no more than would altering its last few sig. bits. On a 68040-based Mac Quadra 950 and on a PC that message appeared in these cases:

`chol(hilb(14))`, `chol(hilbl(14))`, `chol(invhilb(13))` and `chol(invhilbl(15))`. These impose upper bounds upon the dimensions of the Hilbert matrices and their inverses for which the accuracy of MATLAB's `chol(…)` can be tested. Besides, for N > 12 MATLAB can compute neither `invhilb(N)` nor `invhilbl(N)` exactly before `chol(…)` operates upon them.

The correctness of `U = chohilbl(N,K)`, `UI = ichohilb(N,K)`, `R = choihilb(N,K)` and `RI = ichihilb(N,K)` cannot be assessed simply by checking that residuals like

`U'·U - hilbl(N,K)`, `UI·UI' - invhilbl(N,K)`,
`R'·R - invhilbl(N,K)` and `RI·RI' - hilbl(N,K)`

are relatively tiny. Residuals computed from

$\overline{\texttt{U}}$ `= chol(Y)/sqrt(L)`, $\overline{\texttt{UI}}$ `= X*chol(X*invhilbl(N,K)*X)'*X`,
$\overline{\texttt{R}}$ `= chol(invhilbl(N,K))` and $\overline{\texttt{RI}}$ `= X*chol(X*Y*X)'*X/sqrt(L)`

for `[Y,L] = hilbl(N,K)` (so Y := L·H is all integers exactly) and `X = fliplr(eye(N))` (it reverses the order of rows or columns) are similarly tiny elementwise and in norm though the two versions of U, UI, R and RI respectively can differ in norm in as much as about half their sig. bits. Relative elementwise differences are far bigger near those diagonal ends of these triangular matrices at which their elements become smallest, namely the bottoms of U and R, and the tops of UI and RI. Here is what happens when N = 9, K = 13: The foregoing residuals are about as tiny as if caused solely by rounding off the elements of U, UI, R, RI, $\overline{\text{U}}$, $\overline{\text{UI}}$, $\overline{\text{R}}$ and $\overline{\text{RI}}$ to store them; but $\overline{\text{U}}$'s smallest elements disagree almost entirely with U's, and likewise for the other pairs. However, $\|\overline{\text{U}} - \text{U}\|/\|\text{U}\| \approx 1/\sqrt{\text{eps}}$, and likewise for the other psirs, which means that `chol(…)` has lost about half the arithmetic's sig. bits when error is measured normwise.

Whenever I could check, MATLAB's `chol(H)` has always produced an error message or else a relatively tiny residual even for arguments H so nearly singular that κ(H) was of the order of `1/eps`. Then `chol(H)`'s result was often wrong normwise in about half the sig. bits carried by the arithmetic. Why were only about half the sig. bits lost instead of all of them? This can be explained when the last diagonal element of `U = chol(H)` is much smaller than all others, in which case the normwise loss of sig. bits is governed mainly by $\kappa(\text{U}) = \sqrt{\kappa(\text{H})} \approx 1/\sqrt{\text{eps}}$. Otherwise a loss of almost all sig. bits is to be expected according to error-analyses like Ji-Guang Sun's "Perturbation Bounds for the Cholesky and QR Factorizations" in pp. 341-352 of ***BIT* 31** (1991).

More persuasive corroboration of the correctness of our MATLAB programs that compute `U = chohilbl(N,K)`, `UI = ichohilb(N,K)`, `R = choihilb(N,K)` and `RI = ichihilb(N,K)` comes from the relative tinyness elementwise and normwise of differences like `[Y, -L*I]*[UI; U']`, `[R, -L*I]*[Y; RI']`, `[U, -I]*[HI; UI']`, `[HI, -I]*[RI; R']` and `[U, -UI']*[R'; RI]` (with `I = eye(N)`, `[Y, L] = hilbl(N,K)`, `HI = invhilbl(N,K)`) which turn out to be sensitive to uncorrelated errors in U, UI, R and RI. Here is what happens when N = 9, K = 13: For U, UI, R and RI the differences are barely bigger elementwise and normwise than if attributable solely to rounding off those matrices' elements to store them. For $\overline{\text{U}}$, $\overline{\text{UI}}$, $\overline{\text{R}}$ and $\overline{\text{RI}}$ the differences between smaller elements indicate they lost almost all their sig. bits; the norms of the differences indicate a loss of almost half these matrices' sig. bits.

The numerical results summarized above were obtained from MATLAB programs `choteste` and `chotestn` provided below and run originally under MATLAB 3.5 on a 386/7-based Intel 302 PC and a 68040-based Apple Macintosh Quadra 950 in the early 1990s, and more recently under MATLAB 6.5 on an IBM T21 *ThinkPad* laptop after executing `system_dependent('setprecision', 64)`.

Difficult Eigenproblems

The Generalized Symmetric Definite Eigenproblem asks for eigenvectors $\mathbf{b} \neq \mathbf{o}$ and eigenvalues λ that satisfy $A\cdot\mathbf{b} = \lambda\cdot M\cdot\mathbf{b}$ for given real symmetric (or complex Hermitian) matrices $A = A'$ and $M = M'$ of which the latter must be positive definite; $\mathbf{x}'\cdot M\cdot\mathbf{x} > 0$ for all $\mathbf{x} \neq \mathbf{o}$. Since its version 3, MATLAB's function `eig(A, M)` has offered solutions for such problems. They suffer from some pathologies among which is the case that A and M share a *near-nullspace*: If this contains a $\mathbf{z}$ with $\|A\cdot\mathbf{z}\| << \|A\|\cdot\|\mathbf{z}\|$ and $\|M\cdot\mathbf{z}\| << \|M\|\cdot\|\mathbf{z}\|$ then roundoff can contaminate some eigenvalue(s) and eigenvectors severely; for instance, eigenvector $\mathbf{b}$ can be miscomputed as $\mathbf{b} + \zeta\cdot\mathbf{z}$ for a wide range of scalars ζ and still satisfy $A\cdot(\mathbf{b} + \zeta\cdot\mathbf{z}) \approx \lambda\cdot M\cdot(\mathbf{b} + \zeta\cdot\mathbf{z})$ very nearly.

One way to solve the Generalized Symmetric Definite Eigenproblem starts from the Cholesky factorization of $M = U'\cdot U$. It is followed by the computation of $W := U'^{-1}\cdot A\cdot U^{-1} = W'$ whose eigendecomposition $W = Q\cdot\Lambda\cdot Q'$, with an orthogonal matrix $Q = Q'^{-1}$ of W's eigenvectors and real diagonal Λ of W's eigenvalues λ, is computed from `[Q, Λ] = eig(W)` quickly and reliably. Then the desired eigenvectors $\mathbf{b}$ are the columns of $B := U^{-1}\cdot Q$ ordered the same way as are the desired eigenvalues λ on the diagonal of Λ; now $A\cdot B = M\cdot B\cdot\Lambda$ except for roundoff's effect. It is exacerbated thrice by the aforementioned pathology, first in the factorization that gets `U = chol(A)`, second in the application of U^{-1} to get `W = U'\A/U`, and third in `B = U\Q`.

Whatever the way chosen to to solve the Generalized Symmetric Definite Eigenproblem, the accuracy of its program must be tested by applications to data-sets {A, M} with known solutions computable accurately some other way. Nearly pathological test-data-sets should be included.

Offered below is a family of integer-valued data-sets $\{A := Y_{N,K+1},\ M := Y_{N,K}\}$ that approach pathological quickly as N+K increases. Here $Y_{N,K} = L_{N,K}\cdot H_{N,K}$ is a Hilbert matrix scaled up to have integer elements. For this family of data-sets {A, M}, computing the eigenvalues λ quickly and accurately is easy because they are rational multiples by $L_{N,K+1}/L_{N,K}$ of the squares of the singular values of upper-triangular bidiagonal matrices $F_{N,K} := U_{N,K+1}\cdot U_{N,K}^{-1}$; here $U_{N,K}$ is the upper-triangular Cholesky factor of $H_{N,K}$. And $F_{N,K}$ is computed directly, faster and more accurately than from `chohilbl(N,K+1)*choihilb(N,K)`, from the simpler formula $F_{N,K} = \sqrt{¥}_{N,K+1}^{-1}\cdot(J_N + K\cdot I)\cdot\sqrt{¥}_{N,K}^{-1}$ in which N-by-N diagonal matrix $\sqrt{¥}_{N,K}$ has elements $\{\sqrt{¥}_{N,K}\}_{jj} = \sqrt{K+2j-1}$ and bidiagonal J_N has elements $\{J_N\}_{ij} := (\text{if } i \le j \le i+1 \text{ then } i \text{ else } 0)$. For instance,

$$J_6 + K\cdot I = \begin{bmatrix} 1+K & 1 & 0 & 0 & 0 & 0 \\ 0 & 2+K & 2 & 0 & 0 & 0 \\ 0 & 0 & 3+K & 3 & 0 & 0 \\ 0 & 0 & 0 & 4+K & 4 & 0 \\ 0 & 0 & 0 & 0 & 5+K & 5 \\ 0 & 0 & 0 & 0 & 0 & 6+K \end{bmatrix} \quad ; \quad \sqrt{¥}_{6,K} = \begin{bmatrix} \sqrt{K+1} & 0 & 0 & 0 & 0 & 0 \\ 0 & \sqrt{K+3} & 0 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{K+5} & 0 & 0 & 0 \\ 0 & 0 & 0 & \sqrt{K+7} & 0 & 0 \\ 0 & 0 & 0 & 0 & \sqrt{K+9} & 0 \\ 0 & 0 & 0 & 0 & 0 & \sqrt{K+11} \end{bmatrix}.$$

Since singular values of bidiagonal matrices are computable almost as accurately as the precision of the arithmetic, all the eigenvalues λ of the Generalized Symmetric Definite Eigenproblem with integer-valued data-set $\{A := Y_{N,K+1},\ M := Y_{N,K}\}$ are computed far more accurately from the squared singular values of $F_{N,K}$ than from any other program under test unless N is small.

MATLAB program `[A, M, v] = amvhilb(N, K)` below delivers this data-set and its column **v** of eigenvalues λ. As K increases this data-set approaches another pathology: Closely clustered eigenvalues undermine the accuracies of computed eigenvectors; this is a story for another day.

… quis custodiet ipsos Custodes ? (Juvenal)
How shall `amvhilb`'s test-data itself be tested?

To that end, the eigenvalues in column **v** produced by `[A, M, v] = amvhilb(N, K)` have been compared with the columns `u = sort(eig(A, M))` and `w = sort(eig(X*A*X, X*M*X))` in which `X = flipud(eye(N))` is obtained from the identity matrix by reversing its rows (or its columns). In the absence of rounding or other error, **u** , **v** and **w** should all be the same; and except for their last few sig. bits they were the same when computed by MATLAB 6.5 on an IBM T21 laptop PC for smaller values N. As N was increased all the columns diverged, presumably impelled by roundoff. To test this presumption, another program had to be written:

MATLAB 6.5 program `rndir` supplied below redirects the arithmetic's rounding in three more ways besides the default "TO NEAREST"; the three are "TOWARD ZERO", "UP" and "DOWN" as prescribed by IEEE Standard 754. Rerunning the computations of **u**, **v** and **w** under the three directed roundings provided four values, one per direction, of each column. All four values **v** agreed in all but at most the last few of the 53 sig. bits computed, which tends to corroborate their accuracies. As N increased, directed roundings increasingly scattered the columns **u** and **w** from each other and from **v** by roughly similar amounts, thus corroborating how inaccurate **u** and **w** were, until N grew so big that `[A, M, v] = amvhilb(N, K)` no longer generated integer-valued data-sets $\{ A := Y_{N,K+1}, M := Y_{N,K} \}$ exactly regardless of rounding's direction.

Here for example are results for N = K = 10 . Columns **u**, **v** and **w** were computed with arithmetic rounded the default way TO NEAREST. Column $\Delta\mathbf{u}_o = \mathbf{u}_o - \mathbf{u}$ shows how **u** changed when computed with rounding directed TOWARD ZERO. Similarly $\Delta\mathbf{u}_\uparrow$ shows how rounding UP changed **u** , and $\Delta\mathbf{u}_\downarrow$ is for rounding DOWN. Likewise for $\Delta\mathbf{v}_{...}$ and $\Delta\mathbf{w}_{...}$.

u	$\Delta\mathbf{u}_o$	$\Delta\mathbf{u}_\uparrow$	$\Delta\mathbf{u}_\downarrow$	**v**	$\Delta\mathbf{v}_o$	$\Delta\mathbf{v}_\uparrow$	$\Delta\mathbf{v}_\downarrow$	**w**	$\Delta\mathbf{w}_o$	$\Delta\mathbf{w}_\uparrow$	$\Delta\mathbf{w}_\downarrow$
0.255	-0.007	-0.004	-0.389	0.2095058938478430	-3e-16	3e-16	-3e-16	0.247	-0.029	0.002	-0.001
0.386	-0.060	-0.006	-0.136	0.3239813175038243	-9e-16	7e-16	-9e-16	0.377	-0.101	0.001	-0.000
0.512	-0.133	-0.006	-0.133	0.4391226809250292	-12e-16	12e-16	-12e-16	0.502	-0.137	0.001	0.001
0.631	-0.126	-0.006	-0.126	0.5528261852845718	-19e-16	22e-16	-19e-16	0.622	-0.129	0.002	0.002
0.740	-0.114	-0.005	-0.115	0.6612493756197405	-22e-16	26e-16	-22e-16	0.731	-0.115	0.003	0.004
0.833	-0.098	-0.004	-0.099	0.7603044306722687	-26e-16	36e-16	-26e-16	0.825	-0.098	0.003	0.005
0.908	-0.078	-0.002	-0.079	0.8461150279850096	-33e-16	36e-16	-33e-16	0.903	-0.077	0.003	0.005
0.962	-0.056	-0.001	-0.056	0.9152685078254560	-39e-16	40e-16	-39e-16	0.959	-0.055	-0.052	0.003
0.993	-0.031	-0.000	-0.032	0.9649935940457747	-40e-16	42e-16	-40e-16	0.992	-0.032	-0.031	0.001
5.724	-4.732	-3.016	-4.732	0.9932996529571477	-41e-16	44e-16	-41e-16	1.151	-0.159	-0.159	-0.005

Repetitions among the Δ…s suggest that often a few, perhaps as few as one or two, rounding errors overwhelmingly dominated the others in their effect upon computed eigenvalues. This phenomenon renders the *Central Limit Theorem* impotent to justify estimating errors from the Δ…s' spreads; they are too likely to be too small when eigenvalues' errors are exceptionally big. More about such phenomena is posted at `<www.eecs.berkeley.edu/~wkahan/improber.pdf>`.

Condition Numbers

Until further notice, drop the integer subscripts $N \ge 1$ and $K \ge 0$ and use simple abbreviations $H := H_{N,K}$, *etc*. The *Condition Number* $\kappa(H) := \|H\|\cdot\|H^{-1}\|$ is an inverse measure of the relative distance between H and its nearest singular matrix; in fact, the line joining H to zero matrix O makes an angle $\arcsin(1/\kappa(H))$ with the cone through O of singular matrices. This is explained in web-posted class notes `GIlite.pdf` and `NORMlite.pdf` cited below. Besides subscripts N and K now suppressed, the choice of norm $\|\ldots\|$ can affect $\kappa(H)$, perhaps drastically, as happens to the first example in `<…/Math128/FailMode.pdf>`.

The choice matters to us because the condition number reveals how much at worst (or at least) an infinitesimal perturbation δH in H can perturb H^{-1} when measured by the chosen $\|\ldots\|$:
$\delta(H^{-1}) := (H+\delta H)^{-1} - H^{-1} = -H^{-1}\cdot\delta H\cdot H^{-1}$ so $1/\kappa(H) \le (\|\delta(H^{-1})\|/\|H^{-1}\|)/(\|\delta H\|/\|H\|) \le \kappa(H)$, and each "$\le$" can be made "$=$" by an appropriate choice of δH. Moreover, error-analyses of the programs most often invoked to invert a matrix H in floating-point show why, with very rare exceptions, the program's roundoff harms its result by little more than if H had been perturbed first by a pseudo-random δH comparable in norm to roundoff in the elements of H, and then inversion had been performed exactly. In short, with very rare exceptions, we can expect the number of sig. bits lost to roundoff during the inversion program to approximate $\log_2(\kappa(H))$, roughly. How roughly depends upon details including the dimension N and the chosen $\|\ldots\|$.

For norm $\|\ldots\|$ let us choose first MATLAB's `norm(…)`, which is the biggest *Singular Value*:
$\|B\| := \max_{\mathbf{x}\neq\mathbf{0}} \|B\cdot\mathbf{x}\|_2/\|\mathbf{x}\|_2 = \max_{\mathbf{x}\neq\mathbf{0}} \sqrt{((B\cdot\mathbf{x})'\cdot B\cdot\mathbf{x}/(\mathbf{x}'\cdot\mathbf{x}))} = \sqrt{(}$ biggest eigenvalue of $B'\cdot B$).
Tabulated below are computed values of $\log_2(\kappa(H_{K,N}))$ for some small integers K and N.

K	N=2	3	4	5	6	7	8	9	10	11	12	13	14	15	20	30	50	80	120	170
0	4.27	9.03	13.92	18.86	23.83	28.82	33.83	38.84	43.87	48.89	53.93	58.97	64.01	69.05	94.31	144.92	246.33	398.62	601.81	855.90
1	5.27	10.40	15.49	20.55	25.61	30.66	35.71	40.77	45.82	50.88	55.94	61.00	66.05	71.11	96.43	147.11	248.58	400.92	604.14	858.25
3	6.67	12.60	18.18	23.58	28.88	34.12	39.33	44.50	49.65	54.79	59.92	65.04	70.15	75.26	100.75	151.62	253.25	405.69	608.97	863.12
7	8.35	15.49	21.98	28.09	33.95	39.65	45.23	50.73	56.15	61.53	66.86	72.16	77.44	82.69	108.72	160.18	262.35	415.12	618.60	872.87
15	10.18	18.86	26.66	33.90	40.75	47.30	53.64	59.80	65.83	71.74	77.56	83.31	88.98	94.60	122.08	175.31	279.21	433.11	637.28	891.98
31	12.09	22.52	31.93	40.65	48.87	56.69	64.20	71.45	78.50	85.36	92.07	98.64	105.10	111.45	142.03	199.55	308.24	465.66	672.15	928.36
63	14.04	26.35	37.55	47.98	57.84	67.24	76.26	84.97	93.40	101.61	109.60	117.42	125.07	132.58	168.34	233.98	353.40	520.07	733.40	994.44

For any fixed $K \ge 0$ and all N big enough, $\log_2(\kappa(H_{K,N})) \approx$??????????? within ?????????.

WHY ARE THESE CONDITION NUMBERS IRRELEVANT TO THE LEAST-SQUARES FIT OF A POLYNOMIAL $\Xi(\tau)$ TO A GIVEN FUNCTION $y(\tau)$? See Orthogonal Polynomials.
STILL TO COME: Correlations of condition numbers with actual computations of inverses and factors.

Orthogonal Polynomials linked to Hilbert Matrix $H_{N,K}$

The elements υ_{ij} in columns of U^{-1} provided by `ichohilb(N,K)` are the coefficients of the polynomials $\pi_{j-1}(\tau) = \sum_{1\le i\le j} \upsilon_{ij}\cdot\tau^{i-1}$ orthogonal with weight τ^K on $0 \le \tau \le 1$ normalized so

$$\int_0^1 \tau^K\cdot\pi_{i-1}(\tau)\cdot\pi_{j-1}(\tau)\cdot d\tau \;=\; (\text{if } i = j \text{ then } 1 \text{ else } 0)\,.$$

$\pi_n(\tau)$ differs only by a scalar factor from the more convenient-to-compute monic *Shifted Jacobi Polynomial* called $G_n(K+1, K+1, \tau)$ in line **22.2.2** of the *Handbook* … by Stegun and Abramowitz cited under *Further Reading*. We shall call this polynomial $G_{n,K}(\tau)$; then

$$\pi_n(\tau) = {}^{2n+K}\!\varsigma_n\cdot\sqrt{2n+K+1}\cdot G_{n,K}(\tau)\,.$$

But those coefficients υ_{ij} alternate in sign and grow too fast to provide a numerically satisfactory way to compute values of $\pi_{j-1}(\tau)$ unless τ or j is small. Better ways use either the three-term recurrence in lines **22.1.4** and **22.7.2** or a scheme like the one in §**22.18** of the *Handbook* … .

For a given $K \ge 0$, the simplest less inaccurate way to compute $G_{n,K}(\tau)$ by itself at each of a diverse multiplicity of values τ is the expansion $G_{n,K}(\tau) := \sum_{0\le j\le n} (\tau - 1)^{n-j}\cdot({}^n\!\varsigma_j)^2 \,/\, {}^{2n+K}\!\varsigma_j$. MATLAB program `g1(n, K, t)` below does this.

To best compute sequence $\{G_0(\tau), G_{1,K}(\tau), G_{2,K}(\tau), G_{3,K}(\tau), \ldots\}$, three-term recurrence $\mathbb{G}$ sets

$G_{-1}(\tau) := 0$; $G_0(\tau) := 1$; and for $n = 0, 1, 2, 3, \ldots$ in turn,

$G_{n+1,K}(\tau) := (\tau - a_{n,K})\cdot G_{n,K}(\tau) - c_{n,K}\cdot G_{n-1,K}(\tau)$ wherein $c_0 := 0$ and otherwise

$$c_{n,K} := \big(n\cdot(n+K)/(2n+K)\big)^2/\big((2n+K)^2 - 1\big) \le \tfrac{1}{12}\,,\quad \text{and}$$

$$a_{n,K} := \tfrac{1}{2} + \tfrac{1}{2}\cdot K^2/\big((2n+K)\cdot(2n+K+2)\big) < 1\,.$$

Recurrence $\mathbb{G}$ seems numerically stable over the interval $0 \le \tau \le 1$ since every $a_{n,K} + c_{n,K} \le 1$; but $\mathbb{G}$ too loses *relative* accuracy to cancellation since $G_{n,K}(\tau)$ dwindles and oscillates faster as n increases. Starting at $G_0(\tau) \equiv 1$, the magnitude of $G_{n,K}(0) = (-1)^n\cdot((n+K)!)^2/\big((2n+K)!\cdot K!\big)$ plummets as n increases unless K is huge; and if $K > 0$ then as $G_{n,K}(\tau)$ oscillates it decays to below $G_{n,K}(1) = n!\cdot(n+K)!/(2n+K)!$. Maybe $\mathbb{G}$ should be run in extra-precise arithmetic, but it was unavailable when $\mathbb{G}$'s MATLAB program `gr(n, K, t)` below was first written.

Recurrence $\mathbb{G}$'s $[G_0(\tau), G_{1,K}(\tau), G_{2,K}(\tau), \ldots, G_{N-1,K}(\tau)]$ serves in lieu of $[1, \tau, \tau^2, \ldots, \tau^{N-1}]$ to obtain the least-squares-fitted polynomial $\Xi(\tau) = \sum_{1\le j\le N} \chi_j\cdot G_{j-1,K}(\tau) = \sum_{1\le j\le N} \xi_j\cdot\tau^{j-1}$ of degree less than N that minimizes $\int_0^1 \tau^K\cdot(\Xi(\tau) - y(\tau))^2\cdot d\tau$. The coefficients $\{\chi_j\}$ are obtained by solving a linear system of *Normal Equations* with $\mathrm{Diag}\{1/\big({}^{2j+K-2}\!\varsigma_{j-1}\cdot\sqrt{2j+K-1}\,\big)^2\}$ in lieu of the Hilbert matrix $H_{N,K}$, and with $\int_0^1 \tau^K\cdot G_{i-1,K}(\tau)\cdot y(\tau)\cdot d\tau$ in lieu of $\int_0^1 \tau^{i-1+K}\cdot y(\tau)\cdot d\tau$ on the right-hand side. Consequently $\chi_j := \big({}^{2j+K-2}\!\varsigma_{j-1}\big)^2\cdot(2j+K-1)\cdot\int_0^1 \tau^K\cdot G_{j-1,K}(\tau)\cdot y(\tau)\cdot d\tau$.

With coefficients $\{a_{n,K}\}$, $\{c_{n,K}\}$ and $\{\chi_j\}$ in hand, computations of $\Xi(\tau) = \sum_{1\le j\le N} \chi_j\cdot G_{j-1,K}(\tau)$ at diverse arguments τ in $0 \le \tau \le 1$ are probably best performed without first computing the polynomials $G_{\ldots}(\tau)$ explicitly; instead use C.W. Clenshaw's recurrence thus:

Start $\eta_N := 0$, $\eta_{N-1} := \chi_N$, and for $m = N-2, N-3, \ldots, 2, 1, 0$ in turn compute $\eta_m := (\tau - a_{m,K})\cdot\eta_{m+1} - c_{m+1,K}\cdot\eta_{m+2} + \chi_{m+1}$ to obtain $\Xi(\tau) := \eta_0$.

(Variants of Clenshaw's recurrence are explored in §5.5 of *Numerical Recipes in Fortran* 2d ed. (corrected) by W.H. Press, S.A. Teukolsky, W.T. Vetterling & B.P. Flannery (1994), or in §5.4.2 of the 3d ed. (2007), Cambridge Univ. Press.)

$G_{n,K}(\tau)$ has n simple zeros $\{\tau = \zeta_j\}$ all between 0 and 1 . As K increases they crowd near 1 . If $n > 2$ they are computed easiest as the n eigenvalues $\{\zeta_j\}$ of a symmetric tridiagonal matrix

$$T_{n,K} := \begin{bmatrix} a_{0,K} & \sqrt{c_{1,K}} & \cdot & \cdot & \cdot & \cdot & \cdot \\ \sqrt{c_{1,K}} & a_{1,K} & \sqrt{c_{2,K}} & \cdot & \cdot & \cdot & \cdot \\ \cdot & \sqrt{c_{2,K}} & a_{2,K} & \cdots & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdots & \cdots & \cdots & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdots & a_{n-3,K} & \sqrt{c_{n-2,K}} & \cdot \\ \cdot & \cdot & \cdot & \cdot & \sqrt{c_{n-2,K}} & a_{n-2,K} & \sqrt{c_{n-1,K}} \\ \cdot & \cdot & \cdot & \cdot & \cdot & \sqrt{c_{n-1,K}} & a_{n-1,K} \end{bmatrix}$$

assembled from the coefficients of the three-term recurrence $\mathbb{G}$. Program `zg(n,K)` below gets these eigenvalues and does so surprisingly accurately (only for some old MATLAB versions) by refining away most of the rounding errors committed within `eig(T)` . With accurate eigenvalues $\{\zeta_j\}$ in hand, $G_{n,K}(\tau) := \prod_{1\le j\le n}(\tau - \zeta_j)$ is by far the most accurate fast way to compute it.

==========

To be used for asymptotic estimates of Condition Numbers:
Stirling's well-known Asymptotic approximation to Ln(n**!**) accurate for $n >> 1$:
$\mathrm{Ln}(n!) \approx \frac{1}{2}\,\mathrm{Ln}(2\pi) + (n+\frac{1}{2})\cdot\mathrm{Ln}(n) - n + 1/(12n) - 1/(360n^3) + 1/(1260n^5) - 1/(1680n^7) + O(1/n^9)$.

==========

$\|H_{N,K}\|$ is the largest eigenvalue as well as singular value of $H_{N,K}$. For any fixed integer $N > 0$, obviously $\|H_{N,K}\| \searrow 0$ slowly as $K \nearrow +\infty$. For any fixed integer $K \geq 0$, unobviously $\|H_{N,K}\| \nearrow \pi$ slowly as $N \nearrow +\infty$; see Choi [1983].

==========

Let D be the diagonal matrix in `invhilbl`'s formula $H^{-1} = D\cdot H\cdot D$ and let $\$:= \mathrm{signum}(D)$ be a diagonal matrix of alternating ±1's, so that $|D| = \$\cdot D$; and set $A := \sqrt{|D|}\cdot H\cdot\sqrt{|D|}$. This has $A^{-1} = \$\cdot A\cdot\$$, which differs from A only in its checker-board pattern of ± signs. Consequently $\|A^{-1}\| = \|A\|$ for each matrix norm $\|\ldots\| =$ `norm(…, …)` that MATLAB offers, and $\kappa(A) = \|A\|^2$. Cholesky factors of A and of its inverse are the same except for the checkerboard sign pattern. Alas, A has irrational elements, so it cannot be stored exactly in floating-point numbers and consequently cannot serve as test data for matrix software.

==========

Further Reading

"On the Inverses of Certain Matrices" by Dr. Samuel Schechter, pp. 73-77 in *MTAC* **13** #66, Apr. 1959, supplied the algorithms used in `invhilbl` and `dethilbl` here.

"Tricks or Treats with the Hilbert Matrix" by Prof. Man-Duen Choi, pp. 301-312 in *Amer. Math. Monthly* **90** #5, May 1983, surveys much of the lore about Hilbert matrices of finite and infinite dimensions, and includes an extensive list of references.

Prof. D.E. Knuth's book *The Art of Computer Programming* 3rd. ed. (1997, Addison-Wesley) treats Hilbert matrices in §1.2.3 Ex. 45 on p. 38 as special cases of Cauchy matrices $\{1/(x_i + y_j)\}$ for which his Ex. 44 presents a precursor of our test formula $\mathbf{u}'\cdot H_{N,K}^{-1}\cdot\mathbf{u} = N\cdot(N+K)$.

"The condition numbers of real Vandermonde, Krylov and positive definite Hankel matrices" by Bernhard Beckermann, pp. 553-577 of *Numerische Mathematik* **85** (2000), assembles estimates of which one, in *Example 3.3* on p. 570, is $\kappa(H_N) \approx \text{Constant}\cdot(1+\sqrt{2})^{4N}/\sqrt{N}$ for large N .

"Accurate Eigenvalues and SVDs of Totally Nonnegative Matrices" by Prof. Plamen Koev in pp. 1-23 of *SIAM J. MATRIX ANAL. APPL.* **27** #1 (2005) discusses a peculiar bidiagonal factorization (his equations (3.6) on p. 7) of the Cholesky factor U of H (treated as a Cauchy matrix) from which fully accurate eigenvalues of H of widely disparate magnitudes are computed without recourse to more than MATLAB's working precision. Rectangular nonnegative matrices are treated in his paper "Accurate Computations with Totally Nonnegative Matrices" in pp. 731-751 of *ibid.* **29** #3 (2007). Prof. Koev is now in the Math. Dept. at San Jose State University; see `<www.math.sjsu.edu/~koev>` .

Orthogonal polynomials are surveyed in ch. 22 of the *Handbook of Mathematical Functions* … ed. by Milton Abramowitz† & Irene A. Stegun, Nat'l Bureau of Standards (now N.I.S.T.) *Appl. Math. Series* #55 (1964), reprinted in paperback with *many* corrections by Dover, N.Y.

Prof. Nicholas J. Higham's book *Accuracy and Stability of Numerical Algorithms* 2d. ed. (2002, SIAM, Philadelphia) is a well-written 680 page encyclopedia about the intricacies of rounding-error-analysis including a vast bibliography. Hilbert matrices appear in pp. 512-515 .

Relevant Course Notes Posted on my Web Pages `<www.cs.berkeley.edu/~wkahan/...>:`

`<…/MathH110/GCD5.pdf>` "Euclid's GCD Algorithms *vs.* Programs" uses matrices to simplify explanations of the Extended Euclidean Algorithm, continued fractions, Lamé's Theorem and a process attributed to Hermite that speeds the evaluation of integer matrices' determinants using only integer arithmetic. Then a few MATLAB programs are presented to compute GCDs and LCMs and test their correctness for some version of MATLAB on each of the computers most popular in their time. These programs are complicated by MATLAB's denial of access to the *Inexact Flag* mandated by IEEE Standard 754 for floating-point arithmetic.

`<…/MathH110/GIlite.pdf>` "Huge Generalized Inverses of Rank-Deficient Matrices" explores the reciprocal relation between the norm of a possibly generalized inverse of a matrix and its distance from the nearest matrix of the same dimensions but lower rank.

`<…/MathH110/jacobi.pdf>` "Jacobi's Formula for the Derivatice of a Determinant" proves it and applies it to the *Wronskian*, to the derivative of a simple eigenvalue, and to the peculiarities of the inverses of almost all nearly singular matrices.

`<…/MathH110/LstSqrs.pdf>` "Least-Squares Approximation and Bilinear Forms" explains, among other things, why the symmetric positive definite matrix of a least-squares problem's *Normal Equations* represents a linear operator not from a vector space to itself but rather from a vector space to its dual space. This influences our choices of norms and of diagonal scaling.

`<…/MathH110/NORMlite.pdf>` "Notes on Vector and Matrix Norms" explains their properties deemed most important for their applications to analysis in finite-dimensional linear spaces.

`<…/Math128/FailMode.pdf>` "Do MATLAB's `lu(…)`, `inv(…)`, `/` and `\` have Failure Modes ?" Yes; and the simplest way to cope with them is routinely to invoke *Iterative Refinement* with residuals computed extra-precisely if this is feasible for your hardware and version of MATLAB.

Discrepancies

Despite that floating-point hardware and programming languages' compilers may conform to all applicable standards, these are not tight enough to prevent troublesome discrepancies from being observed among results from the same program run on different versions of MATLAB or the same version on different hardware. Usually these discrepancies are so tiny that they are ignored by all but the conscientious individual who worries that they may portend a fatal flaw somewhere in her software or hardware. What *else* could generate these discrepancies? … *Optimizations* :

Optimizations are intended to enhance speed without degrading accuracy excessively, we hope.

Since MATLAB 6.x, its matrix multiplication operations have been optimized to nearly minimize the incidence of cache misses and page faults that severely retarded previous versions' operations upon matrices too big to fit comfortably into the microprocessor's cache(s). Now big matrices get broken into blocks (submatrices) of sizes that depend upon the parameters of the computer's memory hiearchy. These blocks are multiplied and their products added in an order influenced also, perhaps, by the hardware's capacity for concurrency. Discrepancies induced by these influences are usually ignored; they are perceptible only so far as roundoff makes floating-point addition depart slightly from associativity. Discrepancies are noticed most often in a residual like $R := A \cdot X - B \cdot Y$, computed perhaps as one matrix product `R = [A, -B]*[X; Y]`, that so nearly cancels to zero as leaves the computed R dominated by the rounding errors committed during its computation. This is how iterative refinement programs, which depend crucially upon residuals, can get discrepant results from different computers and different versions of MATLAB.

The *Math Library* of programs to compute functions like log and cos is another source of small discrepancies. Usually this library is incorporated in the compiler for whatever language, *C* or *Java*, the developers of MATLAB used to program it. The speeds of computers are gauged by benchmarks that exercise the Math Library, so it gets optimized by the vendors of hardware and compilers who exploit every advantageous hardware feature. Thus sped up, the library's Math functions may produce results different in their last bits for some relatively few input arguments.

Optimizations intended to enhance accuracy may introduce discrepancies too. Intel processors and their clones can, if so enabled, accumulate matrix products in a few registers carrying 64 sig. bits before MATLAB stores them rounded again to 53 sig. bits. Some compilers achieve concurrency by computing a quotient and sometimes a product in one of these few wider registers while computing other products in registers with only 53 sig. bits. This is how a quotient and/or product may get rounded twice, which will change its last (53rd) sig. bit stored with probability roughly 1/4000 if all but its leading few bits are random and independent. Something else can happen with IBM *Power-PC* processors used also for a decade in Apple *Power-Mac*s, *G4*s, *G5*s and *iMac*s. These processors' *Fused Multiply-Add* operation can compute expressions like $x \pm y \cdot z$ with at most one rounding error instead of two. Whether such extra-precise capabilities have been enabled and where they have been employed by a MATLAB program can be difficult to ascertain before they affect `[Y, L] = hilbl(N, K)` for good or ill at the largest N and/or K at which `hilbl` does not balk. The MATLAB program `dblrnd` supplied here ???? reveals how many times some artfully chosen products and quotients got rounded during its explorations.

MATLAB™ Programs

= =

```
function  [Y, L] = hilbl(N,K)
%HILBL  is a  Hilbert matrix,  or one scaled up to integers.
%  H = hilbl(N, K)  is an  N-by-N  Hilbert  matrix:  H(i,j) = 1/(i+j+K-1)
%    but rounded to  53 sig. bits.  If omitted,  K  defaults to  K = 0 ,
%    and then  hilbl(N) = hilb(N) .  If  K  is a nonnegative integer,
%  [Y, L] = hilbl(N,K)  produces  Y = L*H  with  L = lcm([K+1: 2N+K-1]')
%    so all elements  Y(i,j) = L/(i+j-1+K)  are integers computed exactly.
%    Again,  if  K  is omitted it defaults to  K = 0 .
%  Ideally,  if  N  or  K  is too big,  [Y, L] = hilbl(N,K)  should balk
%    rather than deliver any element of  Y  wrong because of roundoff;
%    but some versions of  Matlab  on some computers are not ideal.  See
%    W.K.'s  enhanced versions of  gcd,  lcm  and perhaps  r0und.  Also
%  see  DETHILBL  and  INVHILBL.         W. Kahan,  1996 - 13 Jan. 2009.

L = 1 ;  bigL = 0 ;  if  (nargin < 2),  K = 0 ;  end
if  (nargout > 1)  % ... we need  L = lcm(…) > 1 :
if  (K < 0) | (K ~= round(K)), bad_K = K
    error( '  hilbl(N, K)  requires a nonnegative integer  K .'),  end
%  Compute scale-factor  L = lcm([K+1:2N-1+K]')  accurately enough:
if (N<2),  Y = ones(N) ;  L = (K+1)*Y ;  return, end
L = lcm([K+1: 2*N-1+K]') ;  %... only for  W.K.'s  version of  lcm(...)
bigL = ( isinf(L) & (K > 0) ) ;  if bigL
    L = lcm(lcm([K+1: 2*N-2+K]'), 2*N-1+K) ;  end %...  maybe rounded.
if  isinf(L),  N_K = [N, K]
    disp(' N or K  is too big to compute  hilbl(N,K)  accurately.') %<<<<<<<<
  end,  end  %...  of  nargout > 1  and computation of  L > 1 .
y = [1:N] + (K-1)*0.5 ;  Y = y(ones(N,1),:) ;  Y = L./(Y + Y') ;
if bigL,  Y = round(Y) ;  end %... tries to compensate for rounded  L .
%  Replace buggy  round  by  r0und  in  386-Matlab 3.5 & PC-Matlab 4.2 !
```

= =

```
function  W = invhilbl(N, K)
%INVHILBL  accurate inverse of a segment of the  Hilbert  matrix
%  invhilbl(N, K)  is the inverse of an  N-by-N  matrix  H  whose elements
%    are  H(i,j) = 1/(i+j-1+K) EXACTLY.  ( Rounded elements may be obtained
%    from  [Y,L] = hilbl(N,K)  via  H = Y/L .)  If omitted,  K = 0 ;  then
%    H  is the notoriously ill-conditioned  Hilbert  matrix.  Like  invhilb,
%    invhilbl  is computed not from  Matlab's  inv(H)  but from an elegant
%    (and faster)  formula published by  Sam Schechter  in  MTAC (1959)  to
%    compute the diagonal matrix  D  from which we get  invhilbl = D*H*D .
%    Its result is accurate despite ill-condition.  If  K  is a nonnegative
%    integer,  all  invhilbl(N, K)'s  elements are correct nonzero integers,
%    except perhaps for roundoff that must interfere if  (N,K)  lies beyond
%        (12,0) - (12,2) - (11,5) - (10,8) - (9,10) - (8,15) - (7,27) -
%        (6,39) - (5,73) - (4,195) - (3,1287) - (2,262142) - (1, 2^53)
%    for  Matlab's  53-sig.-bits IEEE 754  floating-point arithmetic.
%    See also  hilb,  invhilb,  hilbl  and  dethilbl .
%                                            W. Kahan,  1994 - 28 Oct. 2008
if  ( nargin < 2 ),  K = 0 ;  end
```

```
u = K+1 ;
if  (N == 1),  W = u ;  return,  end
if  (N == 2),  %...  avoid unnecessarily big intermediate integers
    p = (u+1)*(u+1) ;  q = -u*(u+1)*(u+2) ;
    W = [p*u, q;  q, p*(u+2)] ;  return,  end
x = [0:N-1] ;  y = (x+u).' ;
if ( K == 0 ) ,
      p = N ;  %  ...  The most common case is the fastest:
   elseif ((0 < K) & (K == round(K)) & (K < N)),
      p = N*(N+1) ;  for  j = 2:K ,  p = (p/j)*(N+j) ;  end
   else,  % ...  The most general case is the slowest:
      p = u ;  for  j = 1:N-1,  p = (p/j)*(u+j) ;  end
   end
d = [ p ,  zeros(1,N-1) ] ;  %...  to become  diag(D)
for  j = 1:N-1,  d(j+1) = (( (d(j)/(j+K))*(j-N) )/j)*(N+j+K) ;  end
W = (d.'*d)./( x(ones(N,1),:) + y(:,ones(1,N)) ) ;  % = D*H*D
if ((0 <= K) & (K == round(K))),  W = round(W) ;  end
```

= =

```
function  [dy, L, dhi] = dethilbl(N, K)
%DETHILBL  integer determinants of a scaled  Hilbert  matrix and inverse
%  [dy, L, dhi]  computes integer-valued determinants related to the
%  N-by-N  scaled  Hilbert matrix  Y  produced by  [Y,L] = hilbl(N,K) :
%  dy = det(Y)  and  dhi = det(inv(Y/L))  except that  dethilbl  uses,
%  instead of  Matlab's  det  and  inv,  a method far less vulnerable
%  to roundoff based upon a formula published by  Sam Schechter  in
%  MTAC (1959).  K  must be a nonnegative integer which,  if omitted,
%  defaults to  K = 0 ,  and then  H = Y/L  is the familiar  Hilbert
%  matrix with elements  H(i,j) = 1/(i+j-1) .  The integer scale factor
%  L = lcm([K+1:2*N+K-1]')  is computed accurately only if  N  and  K
%  are not too big;  otherwise  L  may be wrong but  dhi  should be
%  approximately right and then  dy = (L^N)/dhi .  See also  INVHILBL.
%  Needs  W. K.'s  modified  gcd and lcm,  and maybe  r0und  too.
%                                      W. Kahan,  1996 - 26 Oct. 2008

if  ( nargin < 2 ),  K = 0 ;  end
if  (K < 0) | (K ~= round(K)),  N__K = [N, K]
    error( ' dethilbl(N,K)  requires a nonnegative integer  K .'), end

u = K+1 ;  m = 2*N+K-1 ;  L = lcm([u: m]') ;  %...  Correct if finite.
if  isinf(L)  %...  compute instead an approximate  L :
    L = u*(u+1) ;
    for  j = u+2:m ,  L = j*(L/gcd(L,j)) ;  end
    % Now  L = lcm(K+1, K+2, ..., 2*N+K-1) but for roundoff if too big.
    N__K__L = [N, K, L]
  disp('WARNING: N and/or K  are so big in  [dy,L,dhi] = dethilbl(N, K)  that')
  disp('L  may be wrong because of roundoff though  dhi  is approximately right.')
  %  This is the best I can do without  IEEE Standard 754's Inexact Flag.
  end
%  x=[1:N];  H=x(ones(N,1),:);  hilbl(N,K) = round(L./( H+H' + (K-1) )) ;

if ( K == 0 ) ,
      p = N ;  %  ...  The most common case is the fastest:
   elseif (K < N),
```

```
      p = N*(N+1) ;  for  j = 2:K ,  p = (p/j)*(N+j) ;  end
   else,  % ...  The most general case is the slowest:
      p = u ;  for  j = 1:N-1,  p = (p/j)*(u+j) ;  end
   end
d = [ p, zeros(1,N-1) ] ;  %...  to become  |diag(D)|
for  j = 1:N-1,  d(j+1) = (( (d(j)/(j+K))*(N-j) )/j)*(N+j+K) ;  end
dhi = round(prod(d)) ;  dy = round(prod(L./d)) ;
%  Replace buggy  round  in  386-Matlab 3.5  &  PC-Matlab 4.2  by  r0und .

===========================================================================

function  [U, L] = chohilbl(N, K)
%CHOHILBL  Cholesky factor of a Hilbert matrix, or it scaled up to integers.
%  U = chohilbl(N,K)  is the upper-triangular  Cholesky  factor of  Hilbert
%   matrix  H = hilb(N,K)  that  Matlab  would get from  U = chol(H)  if
%   only the last one or two of the  53  sig. bits of  U  suffered from the
%   effects of roundoff in  hilb()  and  chol() .  Instead a neat algorithm
%   delivers  U = chohilbl(N,K)  faster and far more accurately than can
%   U = chol(hilb(N,K))  unless dimension  N  is small.  Both versions of
%   U  have a residual  U`*U - H  so tiny it drowns in its own roundoff.
%  [Ul, L] = chohilbl(N,K)  produces the upper-triangular  Cholesky  factor
%   Ul  of the scaled  Hilbert  matrix  Y = L*H  whose elements are integers
%   obtained exactly from  [Y, L] = hilbl(N,K) .  Ul = U*sqrt(L)  comes out
%   faster and far more accurately than  chol(Y)  can unless  N  is small.
%  Nonnegative integer  K  defaults to zero if omitted.
%  Floating-point operations are so ordered as to generate exact integer
%   intermediate results  (no rounding errors)  about as often as possible.
%  RESTRICTION:  If  N+K  is too big,  [Ul, L] = chohilbl(N,K)  should balk
%   rather than deliver scale factor  L  very wrong because of roundoff.
%  See also  DETHILBL,  HILBL,  INVHILBL  ICHOHILB  and  CHOIHILB.
%  L  needs  W. Kahan's  improved  GCD and LCM.          17 Sept. 2010.

if  ( nargin < 2 ),  K = 0 ;  end
if ~((N==round(N))&(N>0)&(K==round(K))&(K>=0)),  N_K = [N, K]
    error(' chohilbl(N,K)  needs integers  N > 0  and  K >= 0 '),  end
if (nargout == 1),  L = 1 ;  else
%  First compute scale-factor  L = lcm([K+1:2N+K-1]') > 1  accurately:
   L = lcm([K+1: 2*N+K-1]') ; %... works only with  W.K.'s  lcm(...)
   if  isinf(L)
       L = lcm(lcm([K+1: 2*N+K-2]'), 2*N+K-1) ;  end %...  maybe rounded
   if  isinf(L),  bigNplusK = N+K
       error(' N+K is too big to compute  [Ul,L] = chohilbl(N,K)  accurately.')
     end,  end  %... of computation of  L > 1

dr = sqrt( L*[K+1:2:K+2*N-1]' ) ;  %... column
f = ones(1,N) + K ;
for i = 1:N-1,  f(i+1) = (((f(i)/i)*(K+2*i))/(K+i))*(K+2*i+1) ;  end
f = 1.0./f ;  %... row
U = eye(N) ;
for  j = 2:N
    g = U(:,j) ;
    for  i = j-1:-1:1,  g(i) = (g(i+1)/(j-i))*(K+i+j) ;  end
    U(:,j) = g ;  end
U = U.*(dr*f) ;
```

= =

```
function  UI = ichohilb(N, K)
%ICHOHILB  Inverse of the  Cholesky  factor of a  Hilbert  matrix.
%  UI = ichohilb(N,K)  is the inverse of the upper-triangular  Cholesky
%   factor of an  N-by-N  Hilbert  matrix  H = hilbl(N,K)  that  Matlab
%   would get from  UI = inv(chol(H))  if only the last one or two of
%   the  53  sig. bits of  UI  suffered from the effects of roundoff in
%   hilbl(),  chol()  and  inv() .  Instead a neat algorithm delivers
%   UI = ichohilb(N,K)  far faster and far more accurately than can
%   inv(chol(H))  unless  N  is small.  Consequently,  compared with
%   UI*UI' ,  the residual  UI*UI'- invhilbl(N,K)  is very tiny.
%  Nonnegative integer  K  defaults to zero if omitted.
%  NOTE:  Generally  UI ~= choihilb(N,K)  even if computed exactly.
%  Floating-point operations are so ordered as to generate exact integer
%   intermediate results  (no rounding error)  about as often as possible.
%  See also  W.K's  DETHILBL,  HILBL,  INVHILBL,  CHOHILBL  and  CHOIHILBL.
%                                              W. Kahan,   19 Sept 2010.

if (nargin < 2),  K = 0 ;  end
if ~((N==round(N))&(N>0)&(K==round(K))&(K>=0)),  N_K = [N, K]
    error(' ichohilb(N,K)  needs integers  N > 0  and  K >= 0 '),  end
p = -cumprod(-ones(N,1)) ;  %... column  [1, -1, 1, -1, ...]'
dr = sqrt( [K+1:2:K+2*N-1] ).*p' ;  %... row of square roots
for i = 1:N-1,  Ki = K+i ;  Ki2 = Ki+i ;
    p(i+1) = round(( round((p(i)*(1-Ki2))/i)*Ki2 )/Ki) ;
   end  %... of column of integers
UI = eye(N) ;
for  i = 1:N-1 ,  g = UI(i,:) ;  i1 = i-1 ;
    for  j = i:N-1,  g(j+1) = round((g(j)*(K+j+i1))/(j-i1)) ;
       end  %...  of row of integers
    UI(i,:) = g ;  end  %... of upper triangle of integers
UI = UI.*(p*dr) ;
```

= =

```
function  R = choihilb(N, K)
%CHOIHILB  Cholesky  factor of the inverse of an  N-by-N  Hilbert  matrix.
%  R = choihilb(N,K)  is the upper-triangular  Cholesky  factor of the
%   inverse of a  Hilbert  matrix  H = hilbl(N,K)  that  Matlab  would get
%   from  R = chol(invhilbl(N,K))  if only the last one or two of its  53
%   sig. bits suffered from the effects of roundoff in  chol() .  Instead
%   a neat algorithm produces  R = choihilb(N,K)  faster and far more
%   accurately than  R = chol(invhilbl(N,K))  can unless dimension  N  is
%   small.  Both versions of  R  make residual  R`*R - invhilbl(N,K)  tiny
%   enough to drown in roundoff accumulated in the huge computed  R'*R .
%  Nonnegative integer  K  defaults to zero if omitted.
%  NOTE:  R  and  chohilbl(N,K)  are  NOT  each the inverse of the other.
%  Floating-point operations are so ordered as to generate exact integer
%   intermediate results  (no rounding error)  about as often as possible.
%  See also  W.K's  DETHILBL,  HILBL,  INVHILBL,  CHOHILBL  &  ICHOHILB.
%                                               W. Kahan,   17 Sept. 2010
```

```
if (nargin < 2),  K = 0 ;  end
if ~((N==round(N))&(N>0)&(K==round(K))&(K>=0)),  N_K = [N, K]
    error(' choihilb(N,K)  needs integers  N > 0  and  K >= 0 '),  end
q = -cumprod(-ones(1,N)) ;  %... row  [1, -1, 1, -1, ...]
dr = sqrt( [K+1:2:K+2*N-1]' ).*q' ;  %... column of square roots
for j = N:-1:2 ,  j2 = j+j ;
    q(j-1) = round((round((q(j)*(K+j2-1))/(N+K+j-1))*(j2+K-2))/(j-N-1)) ;
   end  %...  of row  q  of integers alternating in sign
R = eye(N) ;
for  j = 2:N ;  g = R(:,j) ;
    for  i = j-1:-1:1,  g(i) = round((g(i+1)*(i+j+K))/(j-i)) ;  end
    R(:,j) = g ;  end  %... of upper triangle of integers
R = R.*(dr*q) ;
```

= =

```
function  RI = ichihilb(N, K)
%ICHIHILB  Inverse of  Cholesky  factor of an inverse  Hilbert  matrix
%  RI = ichihilb(N,K)  is the inverse of the upper-triangular  Cholesky
%   factor of the inverse of  N-by-N  Hilbert  matrix  H = hilbl(N,K)
%   that  Matlab  would get from  RI = inv(chol(inv(H)))  if only the
%   last one or two of the  53  sig. bits of  RI  suffered from the
%   effects of roundoff in  hilbl(),  chol()  and  inv() .  Instead a
%   neat algorithm delivers  RI = ichihilb(N,K)  far faster and far
%   more accurately than  inv(chol(inv(H)))  can unless  N  is small.
%   Consequently residual  RI*RI'- hilbl(N,K)  is very tiny.
%  Nonnegative integer  K  defaults to zero if omitted.
%  NOTE:  Generally  RI ~= chohilbl(N,K)  even if computed exactly.
%  Floating-point operations are so ordered as to generate exact integer
%   intermediate results  (no rounding error)  about as often as possible.
%  See too  W.K's  DETHILBL, HILBL, INVHILBL, CHOHILBL, CHOIHILB, ICHOHILB
%                                              W. Kahan,   19 Sept 2010.

if (nargin < 2),  K = 0 ;  end
if ~((N==round(N))&(N>0)&(K==round(K))&(K>=0)),  N_K = [N, K]
    error(' ichihilb(N,K)  needs integers  N > 0  and  K >= 0 '),  end

dr = sqrt( [K+1:2:K+2*N-1] ) ;  %... row of square roots
p = [zeros(N-1,1); K+2*N-1] ;
for i = N-1:-1:1,  K2i = K+2*i ;
    p(i) = round( round(p(i+1)/(K+N+i))*K2i/(N-i) )*(K2i-1) ;
   end  %... of column of integers
p = 1.0./p ;  %...  column of integers' reciprocals
RI = eye(N) ;
for  i = 1:N-1 ,  g = RI(i,:) ;  i1 = i-1 ;
    for  j = i:N-1,  g(j+1) = round((g(j)*(K+j+i1))/(j-i1)) ;
       end  %...  of row of integers
    RI(i,:) = g ;  end  %... of upper triangle of integers
RI = RI.*(p*dr) ;
```

= =

```
function  Res = choteste(N,K)
%  Res = choteste(N,K)  tests  chohilb, ichohilb, choihilb, ichihil
%   and MATLAB's chol  for elementwise accuracy on  N-by-N  Hilbert
%   matrices and their inverses.  The upper-triangular matrices tested are
%   U = chohilbl(N,K)  vs.  Ub = chol(Y)/sqrt(L)  at  [Y,L] = hilbl(N,K)
%   UI = ichohilb(N,K)  vs  UIb = X*chol(X*HI*X)'*X
%   R = choihilb(N,K)  vs.  Rb = chol(HI)  at  HI = invhilbl(N,K)
%   RI = ichihilb(N,K)  vs.  RIb = X*chol(X*Y*X)'*X/sqrt(L)
%   where  X = flipud(eye(N))  reverses rows or columns,  and
%   Y = hilbl(N,K)*L  is all integers exactly.  If omitted,  K = 0 .
%   Then these residuals' worst elements are computed at  H = Y/L :
%     rU = |U'*U - H|./H ,        rUb = |Ub'*Ub - H|./H ,
%   rUI = |UI*UI' - HI|./(|UI|*|UI|') ,  rUIb = |UIb*UIb' - HI|./(|UI|*|UI|') ,
%     rR = |R'*R - HI|./(|R|'*|R|) ,   rRb = |Rb'*Rb - HI|./(|R|'*|R|) ,
%     rRI = |RI*RI' - H|./H ,     rRIb = |RIb*RIb' - H|./H .
%   Computed also are worst elements of differences     (I = eye(N))
%     dU = |U - Ub|./U ,        dUI = |UI - UIb|./|UI| ,
%     dR = |R - Rb|./|R| ,        dRI = |RI - RIb|./RI ,
%     dYUI = |[Y, -L*I]*[UI; U']|./(Y*|UI|) ,
%                       dYUIb = |[Y, -L*I]*[UIb, Ub']|./(Y*|UIb|) ,
%     dRY = |[R, -L*I]*[Y; RI']|./(|R|*Y) ,
%                       dRYb = |[Rb, -L*I]*[Y; RIb']|./(|Rb|*Y) ,
%     dUHI = |[U, -I]*[HI; UI']|./(U*|HI|) ,
%          dUHIb = |[Ub, -I]*[HI; UIb']|./(|Ub|*|HI|) ,
%     dHIRI = |[HI, -I]*[RI; R']|./(|HI|*RI) ,
%           dHIRIb = |[HI, -I]*[RIb; Rb']|./(|HI|*|RIb|) ,
%     dUR = |[U, -UI']*[R', RI]|./(U*|R|' + |UI|'*RI) ,
%          dURb = |[Ub, -UIb]*[Rb', RIb]|./(|Ub|*|Rb|' + |UIb|'*|RIb|) .
%   Finally  Res = worst elements of  [rU,    rUb;
%                                      rUI,   rUIb;
%                                      rR,    rRb;
%                                      rRI,   rRIb;
%                                      dU,    dUI;
%                                      dR,    dRI;
%                                      dYUI,  dYUIb;
%                                      dRY,   dRYb;
%                                      dUHI,  dUHIb;
%                                      dHIRI, dHIRIb;
%                                      dUR,   dURb]/eps
%  Limits:  Under each  N  is the biggest  K  this program has accepted.
%    N:      2       3    4    5    6   7   8   9  10  11  12  13
%    K:  67145771  9262  535  129  51  40  18  11   7   4   2   0
%   (Programmed originally for a  Mac  with a small memory,  by  W. Kahan)

if (nargin < 2),  K = 0 ;  end
I = eye(N) ;  X = flipud(I) ;  z = 0.5^1023 ;  %... to prevent .../0
[Y, L] = hilbl(N,K) ;  H = Y/L ;  HI = invhilbl(N,K) ;

U = chohilbl(N,K) ;   Ub = chol(Y)/sqrt(L) ;
UI = ichohilb(N,K) ;  UIb = X*chol(X*HI*X)'*X ;
R = choihilb(N,K) ;   Rb = chol(HI) ;
RI = ichihilb(N,K) ;  RIb = X*chol(X*Y*X)'*X/sqrt(L) ;

rU = abs(U'*U - H)./H ;  mrU = max(rU(:)) ;                clear rU
rUb = abs(Ub'*Ub - H)./H ;  mrUb = max(rUb(:)) ;           clear rUb
```

```
aUI = abs(UI) ;  UIUI = aUI*aUI' ;
rUI = abs(UI*UI' - HI)./UIUI ;  mrUI = max(rUI(:)) ;       clear rUI
rUIb = abs(UIb*UIb' - HI)./UIUI ;  mrUIb = max(rUIb(:)) ;  clear rUIb UIUI

aR = abs(R)+z ;  RR = aR'*aR ;
rR = abs(R'*R - HI)./RR ;  mrR = max(rR(:)) ;              clear rR
rRb = abs(Rb'*Rb - HI)./RR ;  mrRb = max(rRb(:)) ;         clear rRb RR
rRI = abs(RI*RI' - H)./H ;  mrRI = max(rRI(:)) ;           clear rRI
rRIb = abs(RIb*RIb' - H)./H ;  mrRIb = max(rRIb(:)) ;      clear rRIb

aUI = abs(UI)+z ;
dU = abs(U - Ub)./(U+z) ;  mdU = max(dU(:)) ;              clear dU
dUI = abs(UI - UIb)./aUI ;  mdUI = max(dUI(:)) ;           clear dUI
dR = abs(R - Rb)./aR ;  mdR = max(dR(:)) ;                 clear dR
dRI = abs(RI - RIb)./(RI+z) ;  mdRI = max(dRI(:)) ;        clear dRI

aHI = abs(HI) ;  mLI = -L*I ;
dYUI = abs([Y, mLI]*[UI; U']).|/(Y*aUI) ;
                          mdYUI = max(dYUI(:)) ;      clear dYUI
dYUIb = abs([Y, mLI]*[UIb; Ub']).|/(Y*abs(UIb)) ;
                        mdYUIb = max(dYUIb(:)) ;      clear dYUIb
dRY = abs([R, mLI]*[Y; RI']).|/(abs(R)*Y) ;
                        mdRY = max(dRY(:)) ;          clear dRY
dRYb = abs([Rb, mLI]*[Y; RIb']).|/(abs(Rb)*Y) ;
                       mdRYb = max(dRYb(:)) ;         clear dRYb
dUHI = abs([U, -I]*[HI; UI']).|/(U*aHI) ;
                          mdUHI = max(dUHI(:)) ;      clear dUHI
dUHIb = abs([Ub, -I]*[HI; UIb']).|/(abs(Ub)*aHI) ;
                         mdUHIb = max(dUHIb(:)) ;     clear dUHIb
dHIRI = abs([HI, -I]*[RI; R']).|/(aHI*RI) ;
                         mdHIRI = max(dHIRI(:)) ;     clear dHIRI
dHIRIb = abs([HI, -I]*[RIb; Rb']).|/(aHI*abs(RIb)) ;
                       mdHIRIb = max(dHIRIb(:)) ;     clear dHIRIb
dUR = abs([U, -UI']*[R'; RI]).|/([U, abs(UI)']*[abs(R)'; RI]) ;
                          mdUR = max(dUR(:)) ;        clear dUR
dURb = abs([Ub, -UIb']*[Rb'; RIb]).|/(abs([Ub, UIb'])*abs([Rb'; RIb])) ;
                         mdURb = max(dURb(:)) ;       clear dURb

Res1 = [mrU, mrUb;  mrUI, mrUIb;  mrR, mrRb;  mrRI, mrRIb;  mdU, mdUI] ;
Res2 = [mdR, mdRI;  mdYUI, mdYUIb;  mdRY, mdRYb;  mdUHI, mdUHIb] ;
Res3 = [mdHIRI, mdHIRIb;  mdUR, mdURb] ;
Res = ceil([ Res1; Res2; Res3 ]/eps) ;
```

= =

```
function  Res = chotestn(N,K)
%  Res = chotestn(N,K)  tests  chohilb, ichohilb, choihilb, ichihil
%   and MATLAB's chol  for normwise accuracy on  N-by-N  Hilbert matrices
%   and their inverses.  The upper-triangular matrices tested are
%   U = chohilbl(N,K)  vs.  Ub = chol(Y)/sqrt(L)  at  [Y,L] = hilbl(N,K)
%   UI = ichohilb(N,K)  vs  UIb = X*chol(X*HI*X)'*X
%   R = choihilb(N,K)  vs.  Rb = chol(HI)  at  HI = invhilbl(N,K)
%   RI = ichihilb(N,K)  vs.  RIb = X*chol(X*Y*X)'*X/sqrt(L)
%   where  X = flipud(eye(N))  reverses rows or columns,  and
%   Y = hilbl(N,K)*L  is all integers exactly.  If omitted,  K = 0 .
%   In what follows we abbreviate  {B} = norm(B)  and  |B| = abs(B) .
%   Then these residual norms' ratios are computed at  H = Y/L :
%     rU = {U'*U - H}/{H} ,         rUb = {Ub'*Ub - H}/{H} ,
%   rUI = {UI*UI' - HI}/{|UI|*|UI|'} ,  rUIb = {UIb*UIb' - HI}/{|UI|*|UI|'} ,
%     rR = {R'*R - HI}/{|R|'*|R|} ,   rRb = {Rb'*Rb - HI}./{|R|'*|R|} ,
%     rRI = {RI*RI' - H}./{H} ,     rRIb = {RIb*RIb' - H}/{H} .
%   Computed also are ratios of norms of differences     (I = eye(N))
%     dU = {U - Ub}/{U} ,         dUI = {UI - UIb}/{UI} ,
%     dR = {R - Rb}/{R} ,         dRI = {RI - RIb}/{RI} ,
%     dYUI = {[Y, -L*I]*[UI; U']}/{Y*|UI|} ,
%                       dYUIb = {[Y, -L*I]*[UIb, Ub']}/{Y*|UIb|} ,
%     dRY = {[R, -L*I]*[Y; RI']}/{|R|*Y} ,
%                       dRYb = {[Rb, -L*I]*[Y; RIb']}/{|Rb|*Y} ,
%     dUHI = {[U, -I]*[HI; UI']}/{U*|HI|} ,
%          dUHIb = {[Ub, -I]*[HI; UIb']}/{|Ub|*|HI|} ,
%     dHIRI = {[HI, -I]*[RI; R']}/{|HI|*RI} ,
%          dHIRIb = {[HI, -I]*[RIb; Rb']}/{|HI|*|RIb|'} ,
%     dUR = {[U, -UI']*[R'; RI]}/{U*|R|' + |UI|'*RI} ,
%    dURb = {[Ub, -UIb']*[Rb'; RIb]}/{|Ub|*|R|' + |UIb|'*|RIb|} .
%   Finally  Res = ceil( [rU,    rUb;
%                         rUI,   rUIb;
%                         rR,    rRb;
%                         rRI,   rRIb;
%                         dU,    dUI;
%                         dR,    dRI;
%                         dYUI,  dYUIb;
%                         dRY,   dRYb;
%                         dUHI,  dUHIb;
%                         dHIRI, dHIRIb;
%                         dUR,   dURb]/eps ) .
%  Limits:  Under each  N  is the biggest  K  this program has accepted.
%    N:      2       3    4    5    6   7   8   9  10  11  12  13
%    K:  67145771  9262  535  129  51  40  18  11   7   4   2   0
%                                                 W. Kahan,  28 June 2011

if (nargin < 2),  K = 0 ;  end
I = eye(N) ;  X = flipud(I) ;
[Y, L] = hilbl(N,K) ;  H = Y/L ;  HI = invhilbl(N,K) ;  sL = sqrt(L) ;

U = chohilbl(N,K) ;   Ub = chol(Y)/sqrt(L) ;
UI = ichohilb(N,K) ;  UIb = fliplr(flipud(chol(fliplr(flipud(HI)))')) ;
R = choihilb(N,K) ;   Rb = chol(HI) ;
RI = ichihilb(N,K) ;  RIb = fliplr(flipud(chol(fliplr(flipud(Y)))'))/sL ;

nU = norm(U) ;  nRI = nU ;  nH = nU*nU ;  nY = L*nH ;
```

```
nUI = norm(UI) ;  nR = nUI ;  nHI = nR*nR ;  % ...= n|HI| etc.

rU = norm(U'*U - H)/nH ;        rUb = norm(Ub'*Ub - H)/nH ;
rUI = norm(UI*UI' - HI)/nHI ;   rUIb = norm(UIb*UIb' - HI)/nHI ;

rR = norm(R'*R - HI)/nHI ;      rRb = norm(Rb'*Rb - HI)/nHI  ;
rRI = norm(RI*RI' - H)/nH ;     rRIb = norm(RIb*RIb' - H)/nH ;

dU = norm(U - Ub)/nU ;
dUI = norm(UI - UIb)/nUI ;
dR = norm(R - Rb)/nR ;
dRI = norm(RI - RIb)/nRI ;

aUI = abs(UI) ;  aHI = abs(HI) ;  mLI = -L*I ;
aUIb = abs(UIb) ;  aUb = abs(Ub) ;
dYUI = norm([Y, mLI]*[UI; U'])/norm(Y*aUI) ;
dYUIb = norm([Y, mLI]*[UIb; Ub'])/norm(Y*aUIb) ;
dRY = norm([R, mLI]*[Y; RI'])/norm(abs(R)*Y) ;
dRYb = norm([Rb, mLI]*[Y; RIb'])/norm(abs(Rb)*Y) ;
dUHI = norm([U, -I]*[HI; UI'])/norm(U*aHI) ;
dUHIb = norm([Ub, -I]*[HI; UIb'])/norm(aUb*aHI) ;
dHIRI = norm([HI, -I]*[RI; R'])/norm(aHI*RI) ;
dHIRIb = norm([HI, -I]*[RIb; Rb'])/norm(aHI*abs(RIb)) ;
dUR = norm([U, -UI']*[R'; RI])/norm([U, abs(UI)']*[abs(R)'; RI]) ;
dURb = norm([Ub, -UIb']*[Rb'; RIb])/norm(abs([Ub, UIb'])*abs([Rb'; RIb])) ;

Res1 = [rU, rUb;  rUI, rUIb;  rR, rRb;  rRI, rRIb;  dU, dUI;  dR, dRI] ;
Res2 = [dYUI, dYUIb; dRY, dRYb;  dUHI, dUHIb;  dHIRI, dHIRIb;  dUR, dURb] ;
Res = ceil([ Res1; Res2 ]/eps) ;
```

= =

= =

```
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 .
```

= =

```
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,  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.
%  For  386-Matlab 3.5,  use  r0und  instead of buggy  round .
end  %...  of  gcd

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

function  y = r0und(x)
%  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. 2008
y = round(x) ;  J = (abs(x) > 1/eps) ;
if any(J(:)),  y(J) = x(J) ;  end

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

function  [A, M, v1, v2, v3, v4] = amvhilb(N,K)
%AMVHILB  N-by-N Hilbert matrix test data for  eig(A,M)
%  [A,M,v] = amvhilb(N,K)  invokes  [A,L1] = hilbl(N,K+1)  and
%   [M,L0] = hilbl(N,K)  to generate integer-valued test data
%   for the Generalized Symmetric Definite Eigenproblem
%   A*b = lambda*M*b  solved by  Lambda = sort(eig(A, M)) .
%   Its computed column  Lambda  of approximate eigenvalues is
%   to be compared with the fairly accurate column  v .
%
%  [A,M, v1,v2,v3,v4] = amvhikb(N,K)  computes four versions of
%   v  stemming from the  SVD  of a bidiagonal matrix,  its
%   transpose,  and their reversals.  The spread among these
%   four reflects effects of roundoff upon Matlab's  svd(...) .

%                                    W. Kahan    15 Jan. 2011
```

```
[A,L1] = hilbl(N,K+1) ;  [M,L0] = hilbl(N,K) ;  LL = L1/L0 ;
J2 = [1:N] ;  J1 = J2 + K ;  %... = [1+K, 2+K, 3+K, ...]
Y1 = J2 + J1 ;  Y = Y1 - 1 ; %... = [1+K, 3+K, 5+K, ...]
J1 = J1./sqrt(Y.*Y1) ;
J2 = J2(1:N-1)./sqrt( Y(2:N).*Y1(1:N-1) ) ;
F = diag(J1) + diag(J2,1) ;   %...  bidiagonal upper triangle
v = sort(svd(F)) ;  v1 = v.*v*LL ;
if (nargout > 3)
    v = sort(svd(F')) ;  v2 = v.*v*LL ;
    F = flipud(fliplr(F)) ;
    v = sort(svd(F)) ;  v3 = v.*v*LL ;
    v = sort(svd(F')) ;  v4 = v.*v*LL ;  end
```

= =

```
function  [r,p] = rndir(R)
%  RNDIR  sets the direction of rounding for  MATLAB 6.5
%  r = rndir(R)  swaps out the old rounding direction  r
%   of  MATLAB 6.5's  floating-point arithmetic and then
%   replaces it by the new direction  R  chosen from one of
%       R = 0.5         round to nearest (the default),
%           0.0         round towards zero (chop),
%           +inf        round towards  +infinity  (up),
%           -inf        round towards  -infinity  (down).
%  Invoke  rndir(r)  to restore the old rounding direction.
%  Omit  R  to get the current rounding direction  r = rndir.
%  [r,p] = rndir(...)  reveals the precision  p  of  MATLAB's
%    arithmetic other than matrix multiply;  p = 24 or 53 .
%  Though  rndir  is unaffected by this  p ,  to sense it via
%   precn.m,  q.v.,  rounding must be the default to nearest.

%  Whether  rndir(R)  works correctly for  MATLAB 7  is unclear.
%  However,  [r,p] = rndir  works for earlier MATLAB versions.

%                                    W. Kahan  20 Dec. 2010

E = 8388608 ;  E3 = E*3 ;  E8 = E*8 ;  %... E = 2^23
precns = [24, 53] ;  dirns = [-inf, 0, 0.5, +inf] ;
p = E8/5 ;  m = (-E8)/5 ;  %...  both rounded,  but how?
p = p - E ;  m = m + E ;
p = (p*4 - E3) + p ;  m = (m*4 + E3) + m ;  %... EXACTLY!
if ((p==0)|(m==0))
   error(' Compiler over-optimization has ruined  rndir(...).')
  end
r = dirns( sign(p) + (sign(m) + 5)/2 ) ;  %... direction
p = precns( 1 + (abs(p)<1) ) ;            %... precision

if nargin > 0
  if ~((R==-inf)|(R==0)|(R==0.5)|(R==inf)),  R = R
    error('rndir(R)  takes  R  only from  {-inf, 0, 0.5, inf} .')
   else  %...  set the new direction of rounding:
    system_dependent('setround', R) ;  end,  end
```

= =

```
function  [G, dG] = g1(n,k,t)
%  G1 =  Shifted Jacobi Orthogonal Polynomial  from powers of  t - 1 .
%  [G, dG] = g1(n, k, t)  evaluates a Shifted Jacobi Orthogonal Polynomial
%  over a column  t  of real numbers typically all between  0  and  1  by
%  expanding  G(...)  in powers of  t - 1 .  G  is the column of values of
%  G[n](k+1,k+1,t),  the Jacobi polynomial of degree  n  in the notation
%  of line  22.2.2  of the  Handbook of Math. Functions ed. by Stegun &
%  Abramowitz.  dG*eps  is a very rough estimate of the uncertainty in
%  G  due to roundoff.

%                                                 W. Kahan,  10 Mar. 2011

t1 = t(:) - 1 ;  at = abs(t1) ;
if (n==0)
    G = ones(length(t1),1) ;  dG = zeros(length(t1),1) ;  return, end
C = cumprod([1, n:-1:1]) ;  C = C.*C ;
D = cumprod([1, [1:n].*[2*n+k:-1:n+k+1]]) ;
C = C./D ;  %... coefficients of polynomial  G
G = polyval(C,t1) ;  dG = polyval(C,at) ;

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

function  [G, dG, a, c] = gr(n,k,t)
%  GR =  Shifted Jacobi Orthogonal Polynomials  from a 3-term recurrence.
%  [G, dG] = gr(n, k, t)  evaluates  Shifted Jacobi Orthogonal Polynomials
%  over a column  t  of real numbers typically all between  0  and  1  by
%  running a recurrence  G(:,j+1) = (t - a(j)).*G(:,j) - c(j)*G(:,j-1) .
%  G(:,j+1)  for  0 <= j <= n  is the column of values of  G[j](k+1,k+1,t),
%  the Jacobi polynomial of degree  j  in the notation of line  22.2.2
%  of the  Handbook of Math. Functions ed. by Stegun & Abramowitz.  dG*eps
%  is a very rough estimate of the uncertainty in  G  due to roundoff.  The
%  coefficient rows  a = [a0,a1,a2,...,an-1]  and  c = [c1,c2,c3,...,cn-1]
%  are available too from  [G, dG, a, c] = gr(n, k, t) .

%                                                 W. Kahan,  13 Mar. 2011

t = t(:) ;  G = ones(length(t),n+1) ;  dG = zeros(length(t),n+1) ;
a = [0:n-1] ;  ak2n = a+a+k ;  c = [1:n-1] ;  ck2n = c+c+k ;
c = c.*(c+k)./ck2n ;  c = c.*c./( (ck2n+1).*(ck2n-1) ) ;
a = 0.5 + (0.5*k*k)./(ak2n.*(ak2n+2)) ;

G(:,2) = t - a(1) ;  dG(:,2) = abs(t - a(1)) + a(1) ;
for  j = 3:n+1
    G(:,j) = (t - a(j-1)).*G(:,j-1) - c(j-2)*G(:,j-2) ;
    dG(:,j) = a(j-1)*abs(G(:,j-1)) ;
    dG(:,j) = dG(:,j) + abs(t - a(j-1)).*dG(:,j-1) + c(j-2)*dG(:,j-2) ;
  end %... j

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = =
```

```
function  z = zg(n,k)
%  ZG = zeros of a shifted Jacobi orthogonal polynomial.
%  z = zg(n,k)  is a column of the  n  zeros of the  Shifted Jacobi
%  Orthogonal Polynomial  G[n](k+1,k+1,t)  found on line  22.2.2  of
%  the  Handbook of Math. Functions ed. by Stegun & Abramowitz,  and
%  computed by  Matlab  program  g1(n,k,z)  or else  gr(n,k,z),  q.v.
%  Zeros  z  are the eigenvalues of a tridiagnal matrix  T  drawn from
%  coefficient rows  a  and  c  generated in  gr.  Rounding errors in
%  eig(T)  are mostly eliminated when  zg  runs in  386 Matlab 3.5,
%  in  Matlab 5.2 on a Mac Quadra 950,  or in  PC Matlab 6.5  after
%  " system_dependent('setprecision', 64) "  has been executed.

%                                          W. Kahan,  15 March 2011

[g,dg,a,c] = gr(n,k,0) ;  c = sqrt(c) ;
T = diag(a) + diag(c,1) + diag(c,-1) ;
[Q,E] = eig(T) ; %... diagonals  E = Q'*T*Q  &  eye = Q'*Q  nearly
dz = sum(Q.*([T,Q]*[Q;-E]))./sum(Q.*Q) ;  %...  refinement
z = diag(E) + dz' ;
```

= =

```
function  p = p0(z, t)
%  P0:  a monic polynomial's values given its zeros.
%  p = p0(z, t)  is the column of values,  at each element of
%  column  t ,  of the monic polynomial whose zeros are in row
%  z ;  i.e.,  p = (t - z(1)).*(t - z(2)).*(t - z(3)).*(...) .

%                                     W. Kahan,  15 March 2011

t = t(:).' ;   z = z(:) ;
nt = length(t) ;  nz = length(z) ;
tz = t(ones(nz,1),:) - z(:,ones(1,nt)) ;
p = prod(tz).' ;
```

= =