Demonstration Of The Ill-Conditioned Nature Of Hilbert Matricies
An application of numerical analysis techniques

In this assignment I used LU decomposition to solve the matrix equation $H \cdot X = B$ where $H$ corresponds to a Hilbert matrix of order $n$, and all elements of $B$ are unity, and compare these results with those given by more accurate integer methods. I found that while the error from the LU decomposition increase exponentially with $n$, the routine still coped very well with quite extremely ill-conditioned matrices over the range of $n$ used (2 to 10). However, for characteristically ill-conditioned matrix systems, one should generally endeavour to find analytical solutions as opposed to utilising numerical methods when accuracy is of primary importance.

Andrew N Jackson                                                                                       17th February 1996
**Introduction**

As a computational experiment in the application of numerical methods, this assignment concerns the demonstration of the ill-conditioned nature of Hilbert matrices. This aim is effected by solving a matrix equation of the form:

\[ H \cdot X = B \]

for a range of order of Hilbert matrix (where all elements of \( B \) are unity). The resulting values contained within \( X \) can then be compared with those computed from formulae (via more accurate integer methods).

As a consequence of the above investigation, this assignment also serves as an introduction to the numerical solution of matrices, in this case by LU decomposition.
Theory

For a particular order, n, the Hilbert matrix is defined such that:

\[
a_{ij} = \frac{1}{i + j - 1}
\]  

(1)

For example, the 3rd order Hilbert matrix is defined as:

\[
\begin{bmatrix}
1 & \frac{1}{2} & \frac{1}{3} \\
\frac{1}{2} & 1 & \frac{1}{4} \\
\frac{1}{3} & \frac{1}{4} & 1
\end{bmatrix}
\]

As mentioned before, in this experiment LU decomposition was used to solve the matrix equation:

\[
H \cdot X = B
\]  

(2)

ie LU decomposition calculates the inverse of the matrix H and uses this inverted form to solve for X (where the elements of B are all unity.). The theory behind LU decomposition is as follows:

It is proposed that an arbitrary matrix A is transformed into a product of two matrices:

\[
L \cdot U = A
\]  

(3)

where \(L\) is a lower triangular matrix and \(U\) is an upper triangular matrix. For the case of a 3 × 3 matrix, equation (3) has this general form:

\[
\begin{bmatrix}
\alpha_{11} & 0 & 0 \\
\alpha_{21} & \alpha_{22} & 0 \\
\alpha_{31} & \alpha_{32} & \alpha_{33}
\end{bmatrix}
\begin{bmatrix}
a_{11} & a_{12} & a_{13} \\
a_{21} & a_{22} & a_{23} \\
a_{31} & a_{32} & a_{33}
\end{bmatrix}
\]

(4)

This means that the set of linear equation we wish to solve for (equation (2)) can be expressed as:

\[
(L \cdot U) \cdot X = L \cdot (U \cdot X) = B
\]

So that we can solve for \(X\) by first solving for \(Y\) such that

\[
L \cdot Y = B
\]  

(5)

and then solving

\[
U \cdot X = B
\]  

(6)

The advantage of breaking up one linear set into two successive ones is that the solution of a triangular set is quite trivial. Equation (5) can be solved by forward substitution, and equation (6) by back substitution, no other manipulation is required.

The question that remains is how to calculate the \(\alpha_{ij}\) and \(\beta_{ij}\) coefficients of the LU decomposition so that they correspond to the \(a_{ij}\) coefficients of the original matrix (see equation (4)). While the full
theory can be found in Numerical Recipies, it is worth noting here that the diagonal terms that appear in the \( L \) and \( U \) matrices mean that the system is over specified, and that we cannot solve for all the coefficients in equation (4). It can be shown that we are allowed to assume all the \( a_{ii} \) terms are equal to unity, so that the \( L \) and \( U \) matrices can be expressed in the combined form:

\[
\begin{bmatrix}
  a_{11} & b_{12} & b_{13} \\
  0 & a_{22} & b_{23} \\
  0 & 0 & a_{33}
\end{bmatrix}
\]

where all the \( \alpha \) and \( \beta \) coefficients are straightforward to calculate.

The ill-conditioned nature of the Hilbert matrices is demonstrated by the calculation of \( \Delta x_n \) (for a range of \( n \)):

\[
\Delta x_n = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{\sum_{i=1}^{n} x_i^2}}
\]

(7)

where \( n \) is the order of the Hilbert matrix being examined and \( x_i \) corresponds to the exact solution as computed by integer methods. It has been shown in the lecture notes that the value of the determinant of a Hilbert matrix becomes closer to zero as \( n \) increases, and so we should find that the error in our calculation also increases with \( n \).

While we have been given the 'integer methods' mentioned above, due to time restrictions I shall not write a general program to solve for any order of \( n \), but instead only for the required range of \( n = 2 \) to 10. This restriction allows me to use values for \( x_i \) in my program which have been calculated by Mapel, and avoids the need to implement the general integer method solution.
Method

The structure of the program which I assembled for this assignment can be broken down into three main sections:

1 • Definition of matrices

Before any calculation can be attempted, we need to define the matrices we wish to solve. This breaks down into three steps; given an order, n:

- Define the Hilbert matrix according to equation (1).
- Define the correct solution matrix from data transferred from Mapel.
- Define the unity matrix $B$.

2 • Decomposition of matrix and solving for $X$

The routines for this section were taken from Numerical Recipes, and work broadly as defined in the theory chapter above. The only difference being that this routine uses pivoting to help stabilise the solution, and so an array is required to keep count of the pivoting moves the program makes.

So, in order to solve the matrix equation my code:

- Calls the decomposition routine LUDCMP, passing the required parameters, and then
- Calls the back/forward substitution routine LUDKSB, transforming the unity matrix $B$ into the solution matrix $X$.

3 • Calculation of the deviation from the real solution

This section of the code simply compares the calculated solution to that from the more accurate integer methods, using equation (7) above. By doing this for the range of $n$ from 2 to 10, we can demonstrate the effect of ill conditioning in the solution of matrices by LU decomposition.

The full code for the above scheme is included in the appendix at the back of this report.
Results
The results from my program are summarised in table 1 and figure 1 below:

<table>
<thead>
<tr>
<th>Order (n)</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>3.1402e-16</td>
</tr>
<tr>
<td>3.0</td>
<td>1.5922e-15</td>
</tr>
<tr>
<td>4.0</td>
<td>1.5305e-13</td>
</tr>
<tr>
<td>5.0</td>
<td>5.9349e-13</td>
</tr>
<tr>
<td>6.0</td>
<td>9.1771e-11</td>
</tr>
<tr>
<td>7.0</td>
<td>2.7273e-09</td>
</tr>
<tr>
<td>8.0</td>
<td>1.3587e-08</td>
</tr>
<tr>
<td>9.0</td>
<td>2.6362e-06</td>
</tr>
<tr>
<td>10.0</td>
<td>1.5018e-04</td>
</tr>
</tbody>
</table>

Table 1: Error against order.

The plot in figure 1 really doesn’t do the data justice, and so figure 2 below shows a plot of $\log_{10}$ error against $n$. 
The points of figure 2 are the results from my program, and the line is a line of best fit through the data, such that:

\[ \log_{10} \Delta x_n = 1.4655 n - 18.928 \]

or:

\[ \Delta x_n = 10^{(1.4655 n - 18.928)} \]

Thus, for a 0.1% error, we need:

\[ n = \frac{\log_{10} 0.001 + 18.928}{1.4655} \]

\[ = 10.87 \]

ie we need an order between \( n = 10.0 \) and \( 11.0 \) to get a 0.1% error from the LU decomposition.

It should be noted that an order 10 Hilbert matrix corresponds to:

\[ \det |H_{10}| \sim 1.0 \times 10^{-58} \]
Conclusion

While the inability of the LU decomposition method to cope with Hilbert matrices increases in a very rapid exponential trend, it should be noted that even with \( \det |H| \sim 10^{-58} \) the routine still did not introduce error of the order of 1%.

In other words, while we should find alternative methods to solve characteristically ill-conditioned matrices (like solving the higher order Hilbert matrices by the integer methods mentioned earlier), the LU decomposition method will, in general, be very reliable for systems where ill-conditioning is not an integral characteristic, as well as for systems of a moderate degree of ill-conditioning (Hilbert up to order \( \sim 10 \)). Of course, the particular choice of method depends on the degree of accuracy that is required from the solution.

If one must apply LU decomposition to very ill-conditioned matrices, it would be possible to use the iterative form of LU decomposition, where the result is run backwards through the calculation and compared with the initial equation in order to improve the algorithm's accuracy.

Note:

Wherever I refer to Numerical Recipes, I am referring to:

Appendix: FORTRAN code implementation of the assignment.

program Hilbert
  c Computes the solutions to the matrix equation \( H X = B \)
  c where \( H \) is the Hilbert
  c matrix of order \( n \), and all elements of \( B \) are unity.
  c An LU decomposition
  c method is employed, using routines from Numerical
  c Recipies.
  c
  c The ill-conditioned nature of Hilbert matrices is
  c demonstrated by
  c calculating the difference between the \( L U \)decomp
  c results with the
  c known from formulae.
  c
  c Andrew Jackson, 1996.
  INTEGER np
  PARAMETER (np=10)
  DOUBLE PRECISION d, dx, h(np, np), soln(np), b(np)
  INTEGER n, index(np)

  c Display program header and get a value for \( n \):
  CALL progheader(n)

  c Define all the required matrices for the analysis:
  CALL defineH(n, np)
  CALL defineSoln(soln, n, np)
  CALL outputmat1D(soln, n, np)
  CALL defineunity(b, n, np)

  c Decompose and solve the \( H X = B \) equation:
  CALL LUDECOMP(H, n, np, index, d)
  CALL LUDKS(H, n, np, Index, B)
  CALL outputmat1D(B, n, np)

  c Analyse error:
  CALL erccalc(n, np, b, soln, dx)

  c Give results:
  WRITE(*,*) 'Error between the methods = \', dx
  stop
end

  c Calculate error between 1D matrices:
  c
  SUBROUTINE erccalc(n, np, est, true, err)
    INTEGER n, np, i
    DOUBLE PRECISION er
    INTEGER n, np, i
    DOUBLE PRECISION B(n, np)

  c Define the solution matrix for order \( n \):
  c
  SUBROUTINE defineSoln(soln, n, np)
    INTEGER n, np
    DOUBLE PRECISION Soln(np)

  c currently using results from Maple until the formulae
  c are implemented

  IF (n.EQ.1) THEN
    Soln(1) = 1.0d0
  ENDIF
  IF (n.EQ.2) THEN
    Soln(1) = 2d0
    Soln(2) = 4d0
  ENDIF
  IF (n.EQ.3) THEN
    Soln(1) = 1s0d0
    Soln(2) = 24s0d0
    Soln(3) = 30s0d0
  ENDIF
  IF (n.EQ.4) THEN
    Soln(1) = 4s0d0
    Soln(2) = 60s0d0
    Soln(3) = 180s0d0
    Soln(4) = 400s0d0
  ENDIF
  IF (n.EQ.5) THEN
    Soln(1) = 50d0
    Soln(2) = 120d0
    Soln(3) = 630d0
    Soln(4) = 1120d0
    Soln(5) = 630d0
  ENDIF
  IF (n.EQ.6) THEN
    Soln(1) = 6d0
    Soln(2) = 210d0
    Soln(3) = 680d0
    Soln(4) = 5040d0
    Soln(5) = 6300d0
    Soln(6) = 2772d0
  ENDIF
  IF (n.EQ.7) THEN
    Soln(1) = 7d0
    Soln(2) = 336d0
    Soln(3) = 3780d0
    Soln(4) = 16800d0
    Soln(5) = 34560d0
    Soln(6) = 33264d0
    Soln(7) = 12012d0
  ENDIF
  IF (n.EQ.8) THEN
    Soln(1) = 8d0
    Soln(2) = 504d0
    Soln(3) = 7560d0
    Soln(4) = 46200d0
    Soln(5) = 138600d0
    Soln(6) = 216216d0
    Soln(7) = 168168d0
    Soln(8) = 51480d0
  ENDIF
  IF (n.EQ.9) THEN
    Soln(1) = 9d0
    Soln(2) = 720d0
    Soln(3) = 13860d0
    Soln(4) = 110880d0
    Soln(5) = 450450d0
    Soln(6) = 1009008d0
    Soln(7) = 2162160d0
    Soln(8) = 823560d0
    Soln(9) = 218790d0
  ENDIF
  IF (n.EQ.10) THEN
    Soln(1) = 10d0
    Soln(2) = 990d0
    Soln(3) = 23760d0
    Soln(4) = 240240d0
    Soln(5) = 1261260d0
    Soln(6) = 3783780d0
    Soln(7) = 6726720d0
    Soln(8) = 7001280d0
    Soln(9) = 3939220d0
    Soln(10) = 923780d0
  ENDIF

  c Define a 1D unity matrix:
  c
  SUBROUTINE defineunity(A, n, np)
    INTEGER n, np, i
    DOUBLE PRECISION A(np)

  c Go through element applying \( a(i,j) = 1/(i+j-1) \) formula:
  DO i=1,n
    DO j=1,n
      A(i,j) = 1d0/(i+j-1)
    ENDDO
  ENDDO
```fortran
ENDDO
ENDDO
return
end

c Output a 2D matrix

SUBROUTINE outputMatrix2D(A,n,np)
INTEGER n,np,i,j
DOUBLE PRECISION A(np,np)
DO i=1,n
DO j=1,n
WRITE(*,*) A(i,j)
ENDDO
END

SUBROUTINE outputMatrix1D(A,n,np)
INTEGER n,np,i
DOUBLE PRECISION A(np)
DO i=1,n
WRITE(*,*) A(i)
ENDDO

SUBROUTINE proghdr(n)
~-------------------------------------------------------------~
WRITE(*,*), 'HX-B matrix equation solver, where H is a Hilbert'
WRITE(*,*), 'matrix of order n and B is unity.'
WRITE(*,*), ~-------------------------------------------------------------~
WRITE(*,*), 'Enter order of matrix to solve: '
READ(*,*), n
return
end

c The following routines are copied from Numerical Recipies. 2nd Ed.
c
SUBROUTINE LUDCMP(a,n,np,idx,nx)
c Given an N x N matrix (a), this routine replaces it by the LU
decomposition
c of a rowwise permutation of itself.
c
Input:  a - the matrix
n - 'active' dimension of the
matrix, a.
np - physical dimension of the
matrix, a.
c
Output:  a - the matrix in LU form [two
matrices stored as one]
c  indx - an output vector used to record
the row permutation
c as effected by partial
pivoting.
c  d - output as +/-1 depending on
whether the number of
row interchanges was even or
c odd, respectively.
c
This routine is used in combination with LUBKSB to
to invert a matrix.
c
INTEGER n,np,indx(n),NMAX
DOUBLE PRECISION d,a(np,np),TINY
PARAMETER (NMAX=500,TINY=1.0D-20)
INTEGER i,imax,j,k
DOUBLE PRECISION aamax, dum, sum, vv(NMAX)
c vv stores the implicit scaling of each row - largest
eff of each row
c normalised to unity
d write=1.0d0
c loop over rows to get implicit scaling information
do i=1,n
 aamax=0.0d0
do j=1,n
 if(dabs(a(i,j)).gt.aamax)aamax=dabs(a(i,j))
 end do
if(aamax.eq.0.0d0) pause 'LUDCMP: Singular matrix'
c save the scaling
 vv(i)=1.0d0/aamax
d end do
c loop over columns - Crout's method
do j=1,n
 do i=1,j-1
 sum=a(i,j)
do k=1,j-1
 sum=sum-a(i,k)*a(k,j)
 end do
 a(i,j)=sum
d end do
c initialise the search for the largest pivot element
 aamax=0.0d0
do i=j,n
 do k=1,j-1
 sum=a(i,k)*a(k,j)
 end do
 a(i,j)=sum
d end do
c figure of merit for the pivot
dum=vv(i)*dabs(sum)
c is it better than the best so far?
 if(dum.gt.aamax)

 imax=i
 aamax=dum
 end if
d end do
c do we need to interchange rows?
 if(j.ne.imax)

do k=1,i-1
 dum=a(imax,k)
 a(imax,k)=a(j,k)
a(j,k)=dum
 end do
c change parity of d and interchange the scale factor
d=-d
 vv(imax)=vv(j)
 end if
 idx(j)=imax

c matrix is singular in effect but substitute for zero
 if(a(j,j).eq.0.0d0)a(j,j)=TINY
c finally, divide by pivot element
 if(j.ne.n)

do k=1,j-1
 a(i,j)=a(i,j)*dum
 end do
 end if
c go back for the next column in the reduction
d end do
SUBROUTINE LUBKSB(a,n,np,idx,nx)
c Solves the set of N linear equations AX=B.
c
Input:  a - the LU decomposed matrix
n - 'active' dimension of the
matrix, a.
np - physical dimension of the
```
matrix, a.
c c - the permutation vector as returned by LUDCMP.
c c b - contains the RHS vector B.
c c Output: b - contains the result vector X.
c
INTEGER i,ii,j,ll,n,np,index(n)
DOUBLE PRECISION sum,a(np,np),b(n)
c when ii is set to a +ve value it becomes the index of the first
nonvanishing element of b.
ii=0
c do forward substitution - unscramble permutation as we go
do i=1,n
ll=index(i)
b(ll)=b(i)
if(ii.ne.0)then
 do j=ll,i-1
 sum=sum-a(i,j)*b(j)
 end do
 else if(sum.ne.0.0d0)then
 a non-zero element was encountered so have to do sums in loop above
 c from now on
 ii=1
 end if
 b(i)=sum
 end do
c now do back substitution
do i=n,1,-1
 sum=b(i)
do j=i+1,n
 sum=sum-a(i,j)*b(j)
 end do
c store a component of the solution vector X
 b(i)=sum/a(i,i)
 end do
 return