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** • **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 matricies. This aim is effected by solving a matrix equation of the form:

$$\mathbf{H} \bullet \mathbf{X} = \mathbf{B}$$

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 matricies, in this case by LU decomposition.

Theory

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

$$a_{ij} = \frac{1}{i + j - 1} \qquad (1)$$

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

$$\begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \end{bmatrix}$$

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

$$\mathbf{H} \bullet \mathbf{X} = \mathbf{B} \qquad (2)$$

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 matricies:

$$\mathbf{L} \bullet \mathbf{U} = \mathbf{A} \qquad (3)$$

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:

$$\begin{bmatrix} \alpha_{11} & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} \end{bmatrix} \bullet \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} \\ 0 & \beta_{22} & \beta_{23} \\ 0 & 0 & \beta_{33} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \qquad (4)$$

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

$$(\mathbf{L} \bullet \mathbf{U}) \bullet \mathbf{X} = \mathbf{L} \bullet (\mathbf{U} \bullet \mathbf{X}) = \mathbf{B}$$

So that we can solve for **X** by first solving for **Y** such that

$$\mathbf{L} \bullet \mathbf{Y} = \mathbf{B} \qquad (5)$$

and then solving

$$\mathbf{U} \bullet \mathbf{X} = \mathbf{B} \qquad (6)$$

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 α_{ij} and β_{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** matricies 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:

$$\begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} \\ \alpha_{21} & \beta_{22} & \beta_{23} \\ \alpha_{31} & \alpha_{32} & \beta_{33} \end{bmatrix}$$

where all the α and β coefficients are straightforward to calculate.

The ill-conditioned nature of the Hilbert matricies is demonstrated by the calculation of Δx_n (for a range of n):

$$\Delta x_n = \sqrt{\frac{\sum_{i=1}^{n}(x_i-\bar{x}_i)^2}{\sum_{i=1}^{n}\bar{x}_i^2}} \tag{7}$$

where n is the order of the Hilbert matrix being examined and $\bar{x}_i$ 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 $\bar{x}_i$ 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 matricies

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:

Order (n)	Error
2.0	3.1402e-16
3.0	1.5922e-15
4.0	1.5305e-13
5.0	5.9349e-13
6.0	9.1771e-11
7.0	2.7273e-09
8.0	1.3587e-08
9.0	2.6362e-06
10.0	1.5018e-04

Table 1: Error against order.

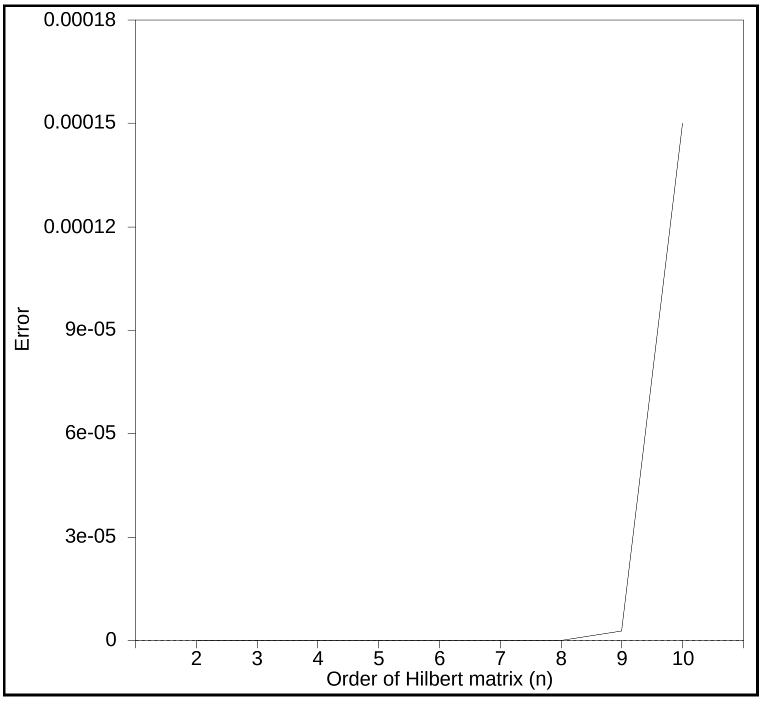

Figure 1: Error against order.

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.

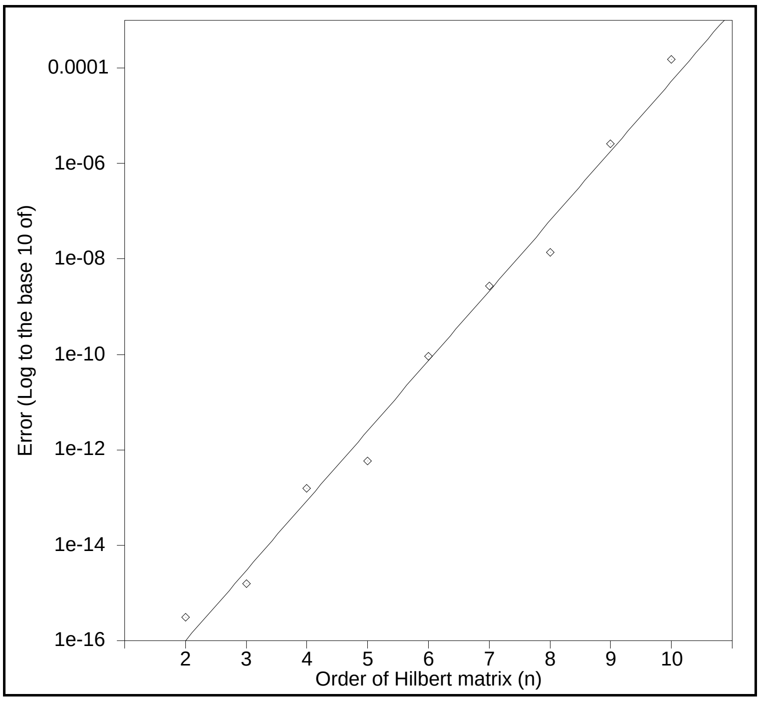

Figure 1: Log_{10} error against order.

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:

$$n = (\log_{10} 0.001 + 18.928)/1.4655$$

$$= 10.87$$

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| ~ 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 ~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 matricies, 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:

- Press, W.H., Teukolsky, S.A., Vetterling, W.T., and Flannery, B.P. 1994, Numerical Recipes in FORTRAN, 2nd Ed. (Cambridge University Press), Chapter 2.

Appendix: FORTRAN code implementation of the assignment.

```
      program Hilbert
c  Computes the solutions to the matrix equation HX=B
where H is the Hilbert
c  matrix of order n, and all elements of B are unity.
An LU decomposition
c  method is employed, using routines from Numerical
Recipies.
c
c  The ill-conditioned nature of Hilbert matricies is
demonstrated by
c  calculating the difference between the LUdecomp
results with the results
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 matricies for the analysis:
      CALL defineH(H,n,np)
      CALL defineSoln(Soln,n,np)
      CALL outputmatrix1D(Soln,n,np)
      CALL defineunity(B,n,np)
c Decompose and solve the HX=B equation:
      CALL LUDCMP(H,n,np,Index,d)
      CALL LUDKSB(H,n,np,Index,B)
      CALL outputmatrix1D(B,n,np)
c Analyse error:
      CALL errcalc(n,np,B,Soln,Dx)
c Give results:
      WRITE(*,*) ‘Error between the methods = ‘,Dx
      stop
      end
c
c Calculate error between 1D matricies:
c
      SUBROUTINE errcalc(n,np,Est,True,err)
      INTEGER n,np,i
      DOUBLE PRECISION
Est(np),True(np),err,diffsum,truesum
      diffsum=0.0d0
      truesum=0.0d0
      DO i=1,n
       diffsum=diffsum+(Est(i)-True(i))**2
       truesum=truesum+True(i)**2
      ENDDO
      err=dsqrt(diffsum/truesum)
      return
      end
c
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 Mapel until the formulea
are implemented
      IF (n.EQ.1) THEN
       Soln(1)=1.0d0
      ENDIF
      IF (n.EQ.2) THEN
       Soln(1)=-2.0d0
       Soln(2)=6.0d0
      ENDIF
      IF (n.EQ.3) THEN
       Soln(1)=3.0d0
       Soln(2)=-24.0d0
       Soln(3)=30.0d0
      ENDIF
      IF (n.EQ.4) THEN
       Soln(1)=-4.0d0
       Soln(2)=60.0d0
       Soln(3)=-180.0d0
       Soln(4)=140.0d0
      ENDIF
      IF (n.EQ.5) THEN
       Soln(1)=5.0d0
       Soln(2)=-120.0d0
       Soln(3)=630.0d0
       Soln(4)=-1120.0d0
       Soln(5)=630.0d0
      ENDIF
      IF (n.EQ.6) THEN
       Soln(1)=-6.0d0
       Soln(2)=210.0d0
       Soln(3)=-1680.0d0
       Soln(4)=5040.0d0
       Soln(5)=-6300.0d0
       Soln(6)=-2772.0d0
      ENDIF
      IF (n.EQ.7) THEN
       Soln(1)=7.0d0
       Soln(2)=-336.0d0
       Soln(3)=3780.0d0
       Soln(4)=-16800.0d0
       Soln(5)=34650.0d0
       Soln(6)=-33264.0d0
       Soln(7)=12012.0d0
      ENDIF
      IF (n.EQ.8) THEN
       Soln(1)=-8.0d0
       Soln(2)=504.0d0
       Soln(3)=-7560.0d0
       Soln(4)=46200.0d0
       Soln(5)=-138600.0d0
       Soln(6)=216216.0d0
       Soln(7)=-168168.0d0
       Soln(8)=51480.0d0
      ENDIF
      IF (n.EQ.9) THEN
       Soln(1)=9.0d0
       Soln(2)=-720.0d0
       Soln(3)=13860.0d0
       Soln(4)=-110880.0d0
       Soln(5)=450450.0d0
       Soln(6)=-1009008.0d0
       Soln(7)=1261260.0d0
       Soln(8)=-823680.0d0
       Soln(9)=218790.0d0
      ENDIF
      IF (n.EQ.10) THEN
       Soln(1)=-10.0d0
       Soln(2)=990.0d0
       Soln(3)=-23760.0d0
       Soln(4)=240240.0d0
       Soln(5)=-1261260.0d0
       Soln(6)=3783780.0d0
       Soln(7)=-6726720.0d0
       Soln(8)=7001280.0d0
       Soln(9)=-3939220.0d0
       Soln(10)=923780.0d0
      ENDIF
      return
      end
c
c Define a 1D unity matrix:
c
      SUBROUTINE defineunity(A,n,np)
      INTEGER n,np,i
      DOUBLE PRECISION A(np)
      DO i=1,n
       A(i)=1.0d0
      ENDDO
      return
      end
c
c Define a Hilbert matrix of order n:
c
      SUBROUTINE defineH(H,n,np)
      INTEGER n,np,i,j
      DOUBLE PRECISION H(np,np)
c Go through element applying a(ij)=1/(i+j-1) formula:
      DO i=1,n
       DO j=1,n
        H(i,j)=1.0d0/(i+j-1.0d0)
```

```
        ENDDO
       ENDDO
       return
       end
c
c Output a 2D matrix
c
       SUBROUTINE outputmatrix2D(A,n,np)
       INTEGER n,np,i,j
       INTEGER*2 k
       DOUBLE PRECISION A(np,np)
       DO i=1,n
        DO j=1,n
         WRITE(*,*) A(i,j)
        ENDDO
        WRITE(*,*) ' '
       ENDDO
       WRITE(*,*) ' '
       WRITE(*,*) 'Press any key...'
       WRITE(*,*) ' '
       CALL GET_KEY@(k)
       return
       end
c
c Output a 1D matrix
c
       SUBROUTINE outputmatrix1D(A,n,np)
       INTEGER n,np,i
       INTEGER*2 k
       DOUBLE PRECISION A(np)
       DO i=1,n
         WRITE(*,*) A(i)
       ENDDO
       WRITE(*,*) ' '
       WRITE(*,*) 'Press any key...'
       WRITE(*,*) ' '
       CALL GET_KEY@(k)
       return
       end
c
c Present user with program header and ask for order of
matrix to solve:
c
       SUBROUTINE progheader(n)
       INTEGER n
       WRITE(*,*)
'-------------------------------------------------'
       WRITE(*,*) 'HX=B matrix equation solver, where H
is a Hilbert'
       WRITE(*,*) 'matrix of order n and B is unity.'
       WRITE(*,*)
'-------------------------------------------------'
       WRITE(*,*) ' '
       WRITE(*,*) 'Enter order of matrix to solve: '
       READ(*,*) n
       return
       end
c
c  The following routines are copied from Numerical
Recipies. 2nd Ed.
c
       SUBROUTINE LUDCMP(a,n,np,indx,d)
c  Given an NxN matrix (a), this routine replaces it by
the LU decomposition
c  of a rowwise permutation of itself.
c
c     Input:    a     - the matrix
c               n     - `active' dimension of the
matrix, a.
c               np    - physical dimension of the
matrix, a.
c
c     Output:   a     - the matrix in LU form [two
matricies stored as one]
c               indx  - an output vector used to record
the row permutation
c                        as effected by partical
pivoting.
c               d     - output as +/-1 depending on
whether the number of
c                       row interchanges was even or
odd, respectively.
c
c  This routine is used in combination with LUBKSB to
solve linear equations
c  or 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
coeff of each row
c  normalised to unity
      d=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
      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,i-1
              sum=sum-a(i,k)*a(k,j)
           end do
           a(i,j)=sum
        end do
c  initialise the search for the largest pivot element
        aamax=0.0d0
        do i=j,n
           sum=a(i,j)
           do k=1,j-1
              sum=sum-a(i,k)*a(k,j)
           end do
           a(i,j)=sum
c  figure of merit for the pivot
           dum=vv(i)*dabs(sum)
c  is it better than the best so far?
           if(dum.ge.aamax)then
              imax=i
              aamax=dum
           end if
        end do
c  do we need to interchange rows?
        if(j.ne.imax)then
           do k=1,n
              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
        indx(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)then
           dum=1.0d0/a(j,j)
           do i=j+1,n
              a(i,j)=a(i,j)*dum
           end do
        end if
c  go back for the next column in the reduction
      end do
      return
      end

      SUBROUTINE LUDKSB(a,n,np,indx,b)
c  Solves the set of N linear equations AX=B.
c
c     Input:    a     - the LU decomposed matrix
c               n     - `active' dimension of the
matrix, a.
c               np    - physical dimension of the
```

```
matrix, a.
c               indx  - the permutation vecotr as
returned by LUDCMP.
c               b     - contains the RHS vector B.
c
c     Output:   b     - contains the result vector X.
c
      INTEGER i,ii,j,ll,n,np,indx(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
c nonvanishing element of b.
      ii=0
c do forward substitution - unscramble permutation as
we go
      do i=1,n
        ll=indx(i)
        sum=b(ll)
        b(ll)=b(i)
        if(ii.ne.0)then
           do j=ii,i-1
              sum=sum-a(i,j)*b(j)
           end do
        else if(sum.ne.0.0d0)then
c a non-zero element was encountered so have to do sums
in loop above
c from now on
           ii=i
        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
      end
```