ICCGLU

A Fortran IV subroutine to solve large sparse general systems of linear equations.

J.J. Dongarra, G.K. Leaf and M. Minkoff

July, 1982

1. Purpose

The Fortran program ICCGLU solves a linear system of equations

$$A*x = b ,$$

where A is a large sparse real gereral matrix. The solution is found through an iterative procedure. The iterative method is based on the conjugate gradient algorithm applied to the implicitly formed normal equations. The method uses a preconditioner to accerlerate convergence. The preconditioner is based on an incomplete LU factorization. This routine can be used to solve symmetric systems as well as non-symmetric.

2. Usage

A. Calling Sequence.

The subroutine statement is:

```
SUBROUTINE ICCGLU(A,N,IA,JA,AF,B,X,R,P,S,TOL,MAXITS,ITS,INFO)
```

where

On input:

A double precision(nmat)
contains the non-zero elements of the matrix, stored row-wise. nmat is the number of locations supplied to represent the array. (See section 3 for more details.)

N integer
is the order of the matrix.

IA integer(N+1)
contains pointers to the start of the rows in the arrays A and JA.

JA integer(nmat)
contains the column indices of the corresponding elements in the array A.

nmat is the number of locations supplied to represent
the array.

B double precision (N)
contains the right hand vector.

X double precision (N)
contains an estimate of the solution, the closer the
estimate is to the true solution the faster the method
will converge.

TOL double precision
is the accuracy desired in the solution.

MAXITS integer
is the maximum number of iterations to be taken
by the routine.

On output

AF double precision (nmat)
contains the incomplete LU factorization of the matrix
contained in the array A.
nmat is the number of locations supplied to represent
the array.

X double precision (N)
contains the solution.

R,P,S double precision (N)
these are scratch work arrays.

ITS integer
contains the number of iterations needed to converge.

INFO integer
signals nature of termination.
INFO = 0 method converged in ITS iterations
INFO = -999 method did not converge in MAXITS iterations.

B. Error conditions and returns.

The conjugate Gradient iteration may require more
steps to attain the desired accuracy then has been
allowed by MAXITS. In this case the subroutine returns
with INFO set to -999.

C. Applicability and restrictions.

This routine is intended for use with very large sparse
non-symmetric matrices. There exist excellent routines in
LINPACK for dense matrices of general form, banded form and
tridiagonal form, ICCGLU should not be used in these
instances. (See the LINPACK package which is on our computers).
There are also special routines which deal with very large
sparse symmetric positive definite matrices which should
be used where applicable. (See the SPARSPAK package which is
on our computers or its description in [2].)

3. Discussion of method and algorithm.

This program grew out of a need to solve large scale linear systems

where the coefficient matrix arose from a finite difference approximation (seven point) to an elliptic partial differential equation in three dimensions. The matrices associated with three dimensional problems tend to be large and sparse. For example, a finite difference mesh of 15x15x30 cells leads to a linear system of order 6750 with about 45,000 non-zero coefficients, for a density of about .001.

The conjugate gradient algorithm has been known for some time. This algorithm has finite termination after n steps when the matrix is symmetric positive definite. The rate of convergence can often be accelerated by multiplying the problem by an approximation to the inverse of the matrix. The approximation used for sparse matrices often is derived from an incomplete factorization of the matrix. By incomplete factorization we mean that the sparsity structure of the original matrix is retained in its factors. This requires ignoring or skipping certain operations during the factorization. This incomplete factorization then serves to precondition the original matrix, in some sense, by clustering the eigenvalues. For a complete description see [1]. In order to that the non-symmetric matrix problem be handled by the conjugate gradient algorithm, the problem must be further transformed into a symmetric positive definite one. This is accomplished by implicitly solving the normal equations. The incomplete factorization and implicit solution of the normal equations are transparent to the user.

The algorithm has the following form.
(inv = inverse, trans = transpose, invtrans = inverse transpose)

```
form incomplete factors L and U
x(0) = initial solution estimate
r(0) = b - A*x(0)
R(0) = trans(A)*invtrans(L)*invtrans(U)*inv(U)*inv(L)*r(0)
p(0) = R(0)
i    = 0
while norm(r)/(norm(a)*norm(x)) > tol do
   s      = inv(U)*inv(L)*A*p(i)
   a(i)   = trans(r(i))*r(i)/(trans(s)*s)
   x(i+1) = x(i) + a(i)*p(i)
   r(i+1) = r(i) - a(i)*trans(A)*invtrans(L)*invtrans(U)*s
   b(i)   = trans(r(i+1))*r(i+1)/(trans(r(i))*r(i))
   p(i+1) = r(i+1) + b(i)*p(i)
   i      = i + 1
end
```

The matrix must be stored in a sparse storage scheme to minimize the amount of memory required. The storage scheme used by ICCGLU is standard in dealing with sparse matrices [2] and is designed to facilitate row operations on the matrix; for this reason the storage is row oriented. The non-zero elements of a row are stored consecutively. To locate and identify these elements, we need to know where the row starts in storage, how long it is and in what columns the elements of the row lie. The scheme requires three arrays, A, IA, and JA. The array A contains non-zero elements of the matrix. The array JA contains the column indices which correspond to the entries of A. If for example, A(K) contains the value of the i,j element of the matrix then JA(K) contains j. The array IA contains pointers to the start of rows in A and JA. IA(1) = 1 and IA(I+1) is defined so that IA(I+1) - IA(I) is the number of non-zero elements in the i-th row of the matrix. The elements themselves are stored in locations IA(I) through IA(I+1)-1 of the array A.

```
a11  a12  0    0    0    0
a21  a22  a23  a24  0    a26
0    a32  a33  0    a35  0
0    0    0    a44  0    0
```

```
                a51  0    0     0     a55   a56
                0    0    0     0     0     a66


A:   a11 a12 a21 a22 a23 a24 a26 a32 a33 a35 a44 a51 a55 a56 a66

JA:   1   2   1   2   3   4   6   2   3   5   4   1   5   6   6

IA:   1   3   8   11  12  15  16
```

We have provided utility subroutines prepare the input arrays.
Subroutine INITS initializes A, IA, and JA in the appropriate manner.
Subroutines PUT and PUTSUM are provided to insert information about the
elements of the matrix into the appropriate arrays. The call to PUT
has the form:

```
       CALL PUT(T,A,N,IA,JA,I,J)
```

where T is the numerical value to be inserted into the structure
and I and J are its row and column in the matrix. Subroutine PUT
will then insert T or overwrite an existing value if one has previously
been defined at that location. Subroutine PUTSUM performs the same function
as PUT except that if a value already exists PUTSUM will add T to it.
The following program initializes the structure, sets up the matrix,
and finds a solution for an example problem.

```
      double precision a(100),af(100),b(10),s(10),x(10),r(10),p(10)
      integer ia(11),ja(100)
c
c     this is a test main program for the incomplete factorization conjugate
c     gradient algorithm which uses an incomplete lu decomposition.
c
      lda = 10
      n = 10
c
c     initialize the structure for the matrix.
c
      call inits(a,n,ia,ja)
c
c     setup the matrix and the right hand side.
c
      do 10 i = 2,n
         call put(1.0d0,a,n,ia,ja,i,i-1)
         call put(1.0d0,a,n,ia,ja,i-1,i)
         call put(4.0d0,a,n,ia,ja,i,i)
         b(i) = 6.0d0
   10 continue
      call put(1.0d0,a,n,ia,ja,1,n)
      call put(1.0d0,a,n,ia,ja,n,1)
      b(1) = 5.0d0
      b(n) = 5.0d0
      call put(4.0d0,a,n,ia,ja,1,1)
c
c     estimate the solution
c
      x(1) = 1.0d0
      do 20 i = 2,n
         x(i) = 0.0d0
   20 continue
c
c     solve the system.
c
      maxits = 20
```

```
      tol = 1.0d-10
      call iccglu(a,n,ia,ja,af,b,x,r,p,s,tol,maxits,its,info)
      write(6,30) its
   30 format(' number of iterations taken',i5)
      write(6,40)(x(i),i=1,n)
   40 format(' solution',5e16.8)
      stop
      end
```

4. References.

1) J.J. Dongarra, G.K. Leaf and M. Minkoff, A Preconditioned Conjugate Gradient Method for Solving a Class of Non-Symmetric Linear Systems, ANL-81-71.

2) Alan George and Joseph W. Liu, Computer Solution of Large Sparse Positive Definite Systems, Prentice-Hall, New Jersey 1981.

5. Program Statistics.

```
      subroutine iccglu(a,n,ia,ja,af,b,x,r,p,s,tol,maxits,its,info)
c
      integer n,ia(1),ja(1),maxits,its,info
      double precision a(1),af(1),x(1),r(1),p(1),s(1),b(1),tol
c
c     this routine performs preconditioned conjugate gradient on a
c     sparse matrix. The preconditioner is an incomplete LU of
c     the matrix.
c
c     on entry:
c
c        a  double precision()
c           contains the elements of the matrix.
c
c        n  integer
c           is the order of the matrix.
c
c        ia integer(n+1)
c           contains pointers to the start of the rows in the arrays
c           a and ja.
c
c        ja integer()
c           contains the column location of the corresponding elements
c           in the array a.
c
c        b double precision (n)
c           contains the right hand side.
c
c        x double precision (n)
c           contains an estimate of the solution, the closer the
c           estimate is to the true solution the faster the method
c           will converge.
c
c        tol double precision
c           is the accuracy desired in the solution.
```

```
c
c        maxits integer
c           is the maximum number of iterations to be taken
c           by the routine.
c
c     on output
c
c        af double precision ()
c           contains the incomplete factorization of the matrix
c           contained in the array a.
c
c        x double precision (n)
c          contains the solution.
c
c        r,p,s double precision (n)
c          these are scratch work arrays.
c
c        its integer
c          contains the number of iterations need to converge.
c
c        info integer
c          signals if normal termination.
c          info = 0 method converged in its iterations
c          info = -999 method did not converge in maxits iterations.
c
c
c     the algorithm has the following form.
c
c        form incomplete factors L and U
c        x(0) <- initial estimate
c        r(0) <- b - A*x(0)
c        R(0) <- trans(A)*invtrans(L)*invtrans(U)*inv(U)*inv(L)*r(0)
c        p(0) <- R(0)
c        i    <- 0
c        while r(i) > tol do
c           s      <- inv(U)*inv(L)*A*p(i)
c           a(i)   <- trans(r(i))*r(i)/(trans(s)*s)
c           x(i+1) <- x(i) + a(i)*p(i)
c           r(i+1) <- r(i) - a(i)*trans(A)*invtrans(L)*invtrans(U)*s
c           b(i)   <- trans(r(i+1))*r(i+1)/(trans(r(i))*r(i))
c           p(i+1) <- r(i+1) + b(i)*p(i)
c           i      <- i + 1
c        end
c
      double precision ai,bi,rowold,rownew,xnrm,anrm
      double precision ddot
      anrm = dabs(a(idamax(ia(n+1)-1,a,1)))
c
c     form incomplete factors L and U
c
      call dcopy(ia(n+1)-1,a,1,af,1)
      call lu(af,n,ia,ja)
c
c     r(0) <- b - A*x(0)
c
      call res(a,n,ia,ja,x,b,r)
c
c     R(0) <- trans(A)*invtrans(L)*invtrans(U)*inv(U)*inv(L)*r(0)
c
c     inv(U)*inv(L)*r(0)
c
      call dcopy(n,r,1,s,1)
      call ssol(af,n,ia,ja,s,1)
      call ssol(af,n,ia,ja,s,2)
c
```

```
c     invtrans(L)*invtrans(U)*above
c
      call ssol(af,n,ia,ja,s,-2)
      call ssol(af,n,ia,ja,s,-1)
c
c     trans(A)*above
c
      call mmult(a,n,ia,ja,r,s,-1)
c
c     p(0) <- R(0)
c
      call dcopy(n,r,1,p,1)
      rowold = ddot(n,r,1,r,1)
      i = 0
c
c     while r(i) > tol do
c
      ai = 1.0d0
   10 continue
      xnrm = dabs(x(idamax(n,x,1)))
      if( dsqrt(rowold)/(anrm*xnrm) .le. tol ) go to 12
c
c        s      <- inv(U)*inv(L)*A*p(i)
c
         call mmult(a,n,ia,ja,s,p,1)
         call ssol(af,n,ia,ja,s,1)
         call ssol(af,n,ia,ja,s,2)
c
c        a(i)   <- trans(r(i))*r(i)/(trans(s)*s)
c
         ai = rowold/ddot(n,s,1,s,1)
c
c        x(i+1) <- x(i) + a(i)*p(i)
c
         call daxpy(n,ai,p,1,x,1)
c
c
c        r(i+1) <- r(i) - a(i)*trans(A)*invtrans(L)*invtrans(U)*s
c
         call ssol(af,n,ia,ja,s,-2)
         call ssol(af,n,ia,ja,s,-1)
         call mmult(a,n,ia,ja,b,s,-1)
         call daxpy(n,-ai,b,1,r,1)
c
c        b(i)   <- trans(r(i+1))*r(i+1)/(trans(r(i))*r(i))
c
         rownew = ddot(n,r,1,r,1)
         bi = rownew/rowold
c
c        p(i+1) <- r(i+1) + b(i)*p(i)
c
         call daxpy2(n,bi,p,1,r,1)
         rowold = rownew
c
c        i      <- i + 1
c
         i = i + 1
         if( i .gt. maxits ) go to 20
         go to 10
   12 continue
   15 continue
      its = i
      return
   20 continue
      info = -999
```

```
its = maxits
return
end
```