Study on Preconditioned Conjugate Gradient Methods for Solving Large Sparse Matrix in CSR Format

(Course Project of ECE697NA)

Lin Du, Shujian Liu

April 30, 2015

Contents

1 Introdution

1.1 Methods for Solving System of Linear Equations

A vast majority of problems in computational science and engineering are reduced to solving a system of linear algebraic equations [1], which can be written in a matrix form, $Ax = b$. There are basically two classes of methods for solving the linear system. The first class is known as direct methods, which theoretically give the exact solution in a finite number of steps. These methods have standard algorithms, are easy to incorporate in large softwares, and very efficient for solving small linear equation systems. For large systems, however, this is not always true due to the inevitable round-off errors. Round-off errors made in one step can be brought into the next step, and continue to propogate to the end of the computing. If the system is very large, the accumulated round-off error may be so significant that the solution is totally wrong. The direct solvers can not solve those nearly singular matrices. Even for those medium sized matrice, given the complexity of $O(N^3)$, the direct methods may be very time demanding, making them not the optimal option.

Though many modifications can be applied to the direct methods, to improve the problems and enhance efficience, it is always a good idea to consider another class of methods, known as iterative methods, especially for a large sparse matrix which contains many zeros. The multiplicate with those zeros can be bypassed in the iterative methods. Since during the iteration, the sturcture of the matrix does not change, iterative methods require less memory than direct methods which need extra memory for fill-in's. Last but not the least, the algorithms of iterative methods involve many matrix-vector multiplications, which is make it very easy to be parallelized.

There are two major classes of iterative methods: (1) statinary iterative methods or fix-point iterations, and (2) projection methods. The stationary methods are named so because the solution to the linear equation system is expressed as finding a stationary point for the iteration $x^{(k+1)} = F(x^{(k)})$. The way to establish a convergence is to design a contraction, a proper mapping satisfying $\|F(x) - F(y)\| < \alpha(x - y)$, and $\alpha < 1$. The convergence rate is determined by α. For a linear system $Ax = b$, it is easy to construct a iteration by split the coefficient matrix $A = M - N$, substituting which into the linear system gives the important *Residual Form*, $x^{(k+1)} = x^{(k)} + M^{-1}r^{(k)}$. The choices of the matrix M correspond to a family of standard iterative methods.

- Richardson ($M = \alpha I$)
- Jacobi ($M = D$, D is the diagonal matrix formed by the diagonal elements of A.)
- Gauss-Seidel ($M = (D + \tilde{L})$, or the lower triangular of matrix A)
- Successive Over-Relaxation ($M = (\omega^{-1}D + \tilde{L})$, and $\omega > 1$)
  a variant of the Gauss-Seidel method resulting in faster convergence. Values of $\omega < 1$ are often used to help establish convergence of a diverging iterative process or speed up the convergence of an overshooting process.
- Symmetric Successive Over-Relaxation ($M = \frac{\omega}{2-\omega}(\omega^{-1}D + \tilde{L})D^{-1}(\omega^{-1}D + \tilde{U})$)
  The symmetry of M permits the application of SSOR as a preconditioner for other

iterative schemes for symmetric matrices. This is the major motivation of the SSOR, since the convergence rate is usually slower than that of SOR.

The stationary iteration methods feature that the iterations are the same for each step, which makes the algorithms very simple. However, for real problems, the convergence of this classical iteration methods is usually very slow.

Another class of iterative methods are known as *projection methods*, the idea of which is to extract an approximate solution from a subspace. The requirement of this method is that the approximate solution x extracted from the subspace K should make the residual $r = b - Ax$ orthogonal to the subspace L. If $K = \text{span}(V)$ and $L = \text{span}(W)$, where $V = [v_1, v_2, \cdots, v_m]$ and $W = [w_1, w_2, \cdots, w_m]$, the expression of the approximate solution is $x = x_0 + V(W^T AV)^{-1} W^T r_0$. The simplest subspace is of course a straight line. For simplicity, we choose the V and W to be single vectors v and w, and successively approximate the solution by the iteration $x^{(k+1)} = x^{(k)} + \alpha v$, where $\alpha = \frac{w^T r^{(k)}}{w^T Av}$. Different choices of w and v correspond to different methods:

- steepest descent method ($v = r$, and $w = r$)
- minimal residual method ($v = r$, and $w = Ar$)
- conjugate gradient method ($v \in \mathcal{K}$, and $w \in \mathcal{K}$)
- GMRES ($v \in \mathcal{K}$, and $w \in A\mathcal{K}$)

where the $\mathcal{K} = \text{span}[v, Av, A^2 v, \cdots, A^{m-1} v]$ is the *Krylov subspace*, because of which , the last two methods are also known as *Krylov subspace methods*. Generally, the projection methods have better convergence properties compared with the classical stationary iteration methods. In the rest of this report, we will focus on the study of conjugate gradient method and the preconditioned conjugate gradient methods.

1.2 Background of Preconditioned Conjugate Gradient Methods

Conjugate gradient method was developed by Magnus R. Hestenes and Eduard Stiefel in 1952 [2], under their effort for finding a machine method that satisfies properties of (1) being simple, (2) requiring a minimum of storage space, (3) converging rapidly preferably to a exact solution if no rounding-off errors occur, (4) being stable and able to self-correct by repetition of the same routine, (5) using as many original data as possible in each step. They found that the CG method and the Gauss elimination method both satisfied the above criteria, and CG is superior to the elimination method as a machine method, for the reasons that

1. CG gives the exact solution in at most n steps if no rounding-off error occurs.
2. CG method is simple to code and less memory demanding
3. The matrix dose not change during the process, enabling the maximum usage of the original data, and the preservation of the structure of the sparse coefficient matrices.

4. approximate solution given at each step is closer to the exact solution than the one at preceding step

5. at each step one can start anew conveniently.

They also found interestingly that the CG and Gauss elimination are two special cases of a even broader method, *conjugate direction method*, which we will demonstrate in next chapter.

CG method and its variants came into wide use only in the mid-1970, when vector computers and massive computer memories were manufactured, which made it possible to use the CG methods to solve the problems including not only the linear equation systems but also nonlinear systems of equations and optimization problems. Since then, the CG algorithm became an important basic tool for solving a wide variety problems [3].

1.3 Purpose of This Study

The purpose of this study is to demonstrate the derivation of the conjugate gradient method, the preconditioned CG, and the algorithms, to carry out numerical experiments by applying the CG and PCG to three linear systems with large sparse coefficient matrices, to discuss the efficiency of the CG and PCG by comparing the experiment results. Also discussed is the Compress Row Storage of large sparse matrices and its incomplete Cholesky factorization. As this is the course project of ECE697NA, we consider it a good opportunity to review all the methods, direct or iterative, we have learned in this class. Thus we summarized the major methods for solving linear system of equations at the beginning of the report, in the introduction section.

The structure of this report is following: Chapter 1 is the brief introduction of the major numerical methods and the background of the CG and PCG methods; Chapter 2 presents the derivation of the CG methods and the algorithms; Chapter 3 discusses the two preconditioners for the CG methods; Chapter 4 is the experiments of solving large sparse linear system with the algorithms developed in sections 2 and 3; Chapter 5 is the summary and conclusions.

We (Lin Du and Shujian Liu) cooperated closely on this project and had many discussions together or online, solved the difficulties together. It is hard to distinguish our efforts, to tell who did what. We contributed equally to this project.

2 Conjugate Gradient Methods

2.1 Derivation of the Conjugate Gradient Methods

From the name of CG method we know there must be two key concepts involved in the method: *conjugate* and *gradient*. Given a symmetric matrix A, two vectors d_1 and d_2 are said to be A-orthogonal or conjugate with respect to A, if $d_1^T A d_2 = 0$. A finite set of vectors $\{d_i\}$ is a conjugate set if $d_i^T A d_j = 0$ for all $i \neq j$. It is also easy to prove that if A is symetric positive definit (spd), the conjugate set of vectors $\{d_i\}$ are linearly independent [4]. We would see later on that the set of search directions in CG methods are a set of conjugate vectors with respect to the coefficent matrix A. Before introducing the concept of *gradient*, we need to introduce a functional

$$\mathcal{F} = \frac{1}{2} x^T A x + b^T x. \tag{1}$$

We observe that the original equation

$$Ax + b = 0 \tag{2}$$

is equivalent to finding a point x that can minimize the functional of Eq. (1). The residual of the original equation is the gradient of the functional. This is where the concept *gradient* comes into the picture.

Suppose the exact solution of Eq. (2) is x^*, and since $\{d_i\}$ form a complete basis for the space of $\mathbb{R}^n$, we can express x^* as

$$x^* = \alpha_0 d_0 + \alpha_1 d_1 + \cdots + \alpha_{n-1} d_{n-1}, \tag{3}$$

where α_i are real numbers. Substituting Eq. (3) into Eq. (2) gives,

$$A(\alpha_0 d_0 + \alpha_1 d_1 + \cdots + \alpha_{n-1} d_{n-1}) + b = 0,$$

which multiplied with d_i gives

$$d_i^T A(\alpha_0 d_0 + \alpha_1 d_1 + \cdots + \alpha_{n-1} d_{n-1}) + d_i^T b = 0.$$

Due to the properties of conjugate vectors, we have

$$\alpha_i = -\frac{d_i^T b}{d_i^T A d_i}, \tag{4}$$

and

$$x^* = -\sum_{i=0}^{n-1} \frac{d_i^T b}{d_i^T A d_i} d_i. \tag{5}$$

From computing α_i one can see why we need conjugate set of vectors to form a basis, not merely orthogonal ones. Upto this step, we should be very happy to see that if we can find $\{d_i\}$, then we can find the exact solution conviniently. The question now is how to construct the mutually conjugate vectors.

We can start with a set of n linearly independent vectors $u_0, u_1, \cdots, u_{n-1}$ and construct $\{d_i\}$ by a successive A-orthogonalization process [2]. We use the formulas

$$\begin{aligned} d_0 &= u_0, \\ d_1 &= u_1 - \alpha_{10} u_0, \\ d_2 &= u_2 - \alpha_{20} u_0 - \alpha_{21} u_1, \\ &\vdots \\ d_i &= u_i - \alpha_{i0} u_0 - \alpha_{i1} u_1 - \cdots - \alpha_{i,i-1} u_{i-1} \\ &\vdots \end{aligned}$$

The coefficient $\alpha_{ij}, (i > j)$ is to be chosen so that d_i is conjugate to d_j. The formula for α_{ij} is evidently

$$\alpha_{ij} = \frac{(u_i, Ad_j)}{(d_j, Ad_j)} \qquad (j < i). \tag{6}$$

If we select successively $u_0 = r_0$, $u_1 = r_1$, $\cdots$, $u_{n-1} = r_{n-1}$, using the equations above we can obtain a sequence of d_i, and x_i. This procedure describes the conjugate gradient method. Interestingly, if one selects $u_0 = (1, 0, \cdots, 0)$, $u_1 = (0, 1, \cdots, 0)$, $\cdots$, $u_{n-1} = (0, 0, \cdots, 1)$, the procedure describes the Gauss elimination method. In next section, we will see how the above analysis can lead to the CG algorithm.

2.2 Derivation of the Algorithm

From the previous section we can see the idea behind the CG algorithm. The algorithm written in pseudocode is given below [5].

1. $x_0 = 0$, $r_0 = b - Ax_0$, $d_0 = r_0$.
2. For $j = 0, 1, \cdots$, until convergence Do:
3. $\alpha_j = (r_j, r_j)/(Ad_j, d_j)$
4. $x_{j+1} = x_j + \alpha_j d_j$
5. $r_{j+1} = r_j - \alpha_j A d_j$
6. $\beta_{j+1} = (r_{j+1}, r_{j+1})/(r_j, r_j)$
7. $d_{j+1} = r_{j+1} + \beta_{j+1} d_j$
8. EndDo

3 Preconditioned Conjugate Gradient Methods

3.1 Diagonal Matrix as the Preconditioner

If the coefficient matrix A is ill-conditioned, the CG algorithm will still converge very slow. A work-around is to precondition the matrix A, with the purpose to reduce its condition number. Below we give the preconditioned conjugate gradient method with the diagonal matrix D being the preconditioner. D is formed by the diagonal elements of matrix A. The algorithm written in pseudocode is given below [5].

1. $x_0 = 0$, $r_0 = b - Ax_0$, $z_0 = D^{-1}r_0$, and $d_0 = z_0$.
2. For $j = 0, 1, \cdots$, until convergence Do:
3. $\alpha_j = (r_j, z_j)/(Ad_j, d_j)$
4. $x_{j+1} = x_j + \alpha_j d_j$
5. $r_{j+1} = r_j - \alpha_j A d_j$
6. $z_{j+1} = D^{-1} r_{j+1}$
7. $\beta_{j+1} = (r_{j+1}, z_{j+1})/(r_j, z_j)$
8. $d_{j+1} = z_{j+1} + \beta_{j+1} d_j$
9. EndDo

3.2 Split Preconditioner using Incomplete Cholesky Factorization

We can also use split preconditioning technique to modify the matrix A. Assuming M is factorized as: $M = M_L M_R$, the split preconditioning is

$$M_L^{-1} A M_R^{-1} u = M_L^{-1} b, \quad \text{with} \quad x = M_R^{-1} u.$$

Since the CG method required the coefficient matrix to be spd, a practical factorization is *Cholesky factorization*, which is

$$M = LL^T,$$

where L is a lower triangular matrix. In this project, we will use the CG method to solve a large sparse linear system, meaning the matrix A will contain a large number of zeros. However, the result of Cholesky factorization L is not necessarily to be sparse. This may require extra memories to store the fill-ins generated in the process of factorization. Another down side is the factorization will take much time, slowing the algorithm. An alternative approach is to use *incomplete Cholesky factorization* method. This method and the algorithm will be illustrated in next section. Here we assume that the matrix A has been factorized, and the lower triangular matrix L has been obtained. The corresponding split precondtioning conjugate gradient algorithm written in pseudocode is given below [5].

1. $x_0 = 0$, $r_0 = b - Ax_0$, $\hat{r}_0 = L^1 r_0$, and $d_0 = L^{-T}\hat{r}_0$.

2. For $j = 0, 1, \cdots,$ until convergence Do:
3. $\alpha_j = (\hat{r}_j, \hat{r}_j)/(Ad_j, d_j)$
4. $x_{j+1} = x_j + \alpha_j d_j$
5. $\hat{r}_{j+1} = \hat{r}_j - \alpha_j L^{-1} A d_j$
6. $\beta_j = (\hat{r}_{j+1}, \hat{r}_{j+1})/(\hat{r}_j, \hat{r}_j)$
7. $d_{j+1} = L^{-T}\hat{r}_{j+1} + \beta_{j+1} d_j$
8. EndDo

4 Numerical Experiments

4.1 Matrices in Compress Row Storage Format

Many scientific or engineering problems, like boundary element method, will result in a system of linear equation with a large sparse coefficient matrix, which contains a lot of zeros. Storing these zeros need a large amount of memories. However, these zeros literaly give no information. An efficient way to store the matrix is to keep only the non-zero elements in a vector a, and record their coordinates in another two vectors ja and ia. Basically, there are three ways of compress storage:

- COO ja stores the column numbers, ia stores the row numbers.
- CSR Compressed Sparse Row, ia stores only the beginning number of the first element of each row.
- CSC Compressed Sparse Column, ja stores only the beginning number of the first element of each column.

Within above three approaches, the CSR is most efficient, and most popular. Thus, the three matrices we will be used for numerical experiment are stored in this way. Though the CSR format can save a great amount of memories, the price is that more calculations will be involved, and the data locality is not as good as the traditional storage. One evidence is that we can not obtain the coordinate of element $a(k)$ directly, or locate the $a(k)$ for the element of (i, j) directly. Mapping between k and (i, j) is needed. In next section, we shall write a subroutine to do this mapping, which will be used in the incomplete Cholesky factorization algorithm.

4.2 Illustration of the Algorithms

Up to now, we derived the algorithm of CG and PCG, which can be converted into Fortran code easily. However, we haven't obtained the algorithm of the incomplete Cholesky factorization applied on sparse matrix in CSR format. Incomplete Cholesky factorization is quite a standard one, and below is the algorithm written in Matlab code [6]:

```
function a = ichol(a)
        n = size(a,1);

        for k=1:n
                a(k,k) = sqrt(a(k,k));
                for i=(k+1):n
                    if (a(i,k)!=0)
                        a(i,k) = a(i,k)/a(k,k);
                    endif
                endfor
                for j=(k+1):n
                    for i=j:n
                        if (a(i,j)!=0)
                            a(i,j) = a(i,j)-a(i,k)*a(j,k);
                        endif
```

```
                endfor
            endfor
        endfor

    for i=1:n
        for j=i+1:n
            a(i,j) = 0;
        endfor
    endfor
endfunction
```

However, this code is for the matrix of dense storage. To convert it to a version suitable for the CSR matrix, we need to consider two things: (1) how to map $a(k)$ with $a(i,j)$, and (2) how to take advantage of the symmetry properties of the matrix A. For the question (1), we write a small subroutine `iefinder(i,j,ie,ia,ja)`, which will give the corresponding element number `ie` in the vector a, when the original coordinate (i,j) is given. The Fortran code is:

```
subroutine iefinder(i,j,ie,ia,ja)
! Purpose: find the value of ie for sa(ie)=a(i,j)
  implicit none
  Integer :: i, j, k,ie, ia(*), ja(*)

  ie=0
  do k = ia(i), ia(i+1)-1
     if(ja(k) .eq. j) then
       ie = k
       exit
     endif
  enddo

  return
end subroutine iefinder
```

Before answering question (2), we observe that vector ia gives the ie of the non-zero elements on a specific row, and ja will give the column number of those elements. Since the matrix A is symmetric, we can obtain the row numbers for those non-zero elements in a specific column as well. This information implies that we do not need to loop over all the rows or columns to update the elements in each outer loop, as in the MATLAB code does, which has a computational cost of $O(N^3)$. We can actually use only one full loop (from 1 to N) to fulfill the incomplete Cholesky factorization. The Fortran code is:

```
subroutine ichol(sl,ja,ia,N)
! Purpose: incomplete factorization of CRS sparse matrix. L is stored in the
!          lower triangular part of sl and L^T is stored in the upper
!          triangular part of sl.
!
! Arguments: akk(*) stores the number of diagonal element, diag(k) the index of
!            of the diagonal element a(k,k).
!            ja, ia and N are the same with those used in the main program.

  implicit none
  Integer :: N, i, j, k, ie,ieei, ieej, ie1, ie2, ja(*), ia(*), diag(1:N)
```

```
  Double precision :: sl(*), akk

  ie=0
  ie1=0
  ie2=0
  akk=0.d0
! find the diag element index ie
  do k=1,N
     call iefinder(k,k,ie,ia,ja)
     diag(k)=ie
  enddo

  do k=1, N
     sl(diag(k))=sqrt(sl(diag(k)))
     akk=1.d0/sl(diag(k))
     do ieei=diag(k)+1,ia(k+1)-1 ! from the non-zero column # to
        i=ja(ieei)               ! locate the non-zero row #
        call iefinder(i,k,ie,ia,ja)
        sl(ie)=sl(ie)*akk
        do ieej=ie+1,diag(i)  ! update the row elements of lower triangular
           j=ja(ieej)
           call iefinder(i,k,ie1,ia,ja)
           call iefinder(j,k,ie2,ia,ja)
           if ( (ie1 .ne. 0) .and. (ie2 .ne. 0)) then
                 sl(ieej)=sl(ieej)-sl(ie1)*sl(ie2)
           endif
        enddo
     enddo
  enddo

  ! fill the upper triangular of the matrix

  do i=1, N
     do ie1=diag(i)+1,ia(i+1)-1
        j=ja(ie1)
        call iefinder(j,i,ie2,ia,ja)
        sl(ie1)=sl(ie2)
     enddo
  enddo

  return
end subroutine ichol
```

The inner loops over *ieei* and *ieej* take advantage of the symmetry property of the matrix A, thus they do not need to go from 1 to N. This Fortran90 code can factorize the matrices very quickly.

4.3 Comparison of Two Incomplete Cholesky Factorization Subroutines

A subroutine of `ICholeski` has been provided to perform the incomplete Cholesky factorization of the cofficient matrix A. It is interesting to compare the efficiency of the `ICholeski`

with our own subroutine, `ichol`. In the code, we add a case4 for this comparison purpose, and perform the decomposition by the two subroutines sequentially and record the time. The results are listed in the table below

Table 1: The comparison of the efficiency of subroutines

Matrices	N	nnz	`ICholeski` (s)	`ichol` (s)
gsmat1	432000	7955116	0.3734	0.9004
laplace3d	180000	1211400	4.69×10^{-2}	6.39×10^{-2}
c6h6-vh	48913	1447514	0.1100	0.3317

We see that the provided subroutine `ICholeski` requires less time for the decomposition of the matrices. However, the performance of our subroutine `ichol` is also acceptable. We used only one loop from 1 to N, which makes it of complexity of $O(N)$. We do not know the source code of the subroutine `ICholeski`. From its description, we can see that it deals with only the lower triangular of the orignal sparse matrices. In our subroutine, we deal with the whole original matrices. Probably, this difference contributes to the difference in the computing time. Though `ichol` needs a little more time, it also has some merits: (1) it dose not need the pre-treatment of the matrix, extracting the lower triangular of A; (2) it is simple and dose not need extra libraries; (3) it dose not need the extra memory for the extracted lower triangular matrix, thus is more memory efficient; (4) it outputs a matrix with the same structure with the original one, thus vectors *isa*, *jsa* can be used directly; (5) it stores both the L and L^T in the output matrix, which is very convenient in the PCG algorithm, where L^T is also needed. Considering those merits, we will use our own subroutine `ichol` for the decomposition tasks in the following numerical experiments.

4.4 Results and Discussion

To investigate the efficiency of the algorithms of CG, the left preconditioned CG and the plit preconditioned CG, we carried out numerical experiments upon three different large sparse matrices stored in the CSR format, which are *gsmat1*, *laplace3d* and *c6h6-vh*. The efficiency is characterized by the number of iterations required for the convergence and the time it takes. Once the normalized residual $\|r\|_2/\|b\|_2 < 10^{-14}$, the convergence is considered to be reached.

Figure 1 shows the normalized residual as a function of the iteration numbers. The black line with open square markers is the result of conjugate gradient method, denoted by CG, the blue line with open circle markers is the result of preconditioned CG with the diagonal matrix being the preconditioner, denoted by CG-D, and the red line with open triangle markers is for the split precondtioned conjugate gradient with the incomplete Cholesky factorization matrix L as the preconditioner, denoted by CG-S. We observed that the line of CG-D is almost overlapped with that of CG, and they require the same number of iterations to achieve the convergence, indicating that the preconditioning with the diagonal matrix has no effect on the coefficient matrix A. However, the CG-S line has a more steep slop than that of the CG and CG-D, and it reached 10^{-14} using 19 iterations, less than half of the iteration numbers for CG and CG-D. This shows that the split preconditioning with the matrix L improved the coefficient matrix A greatly, making it better conditioned.

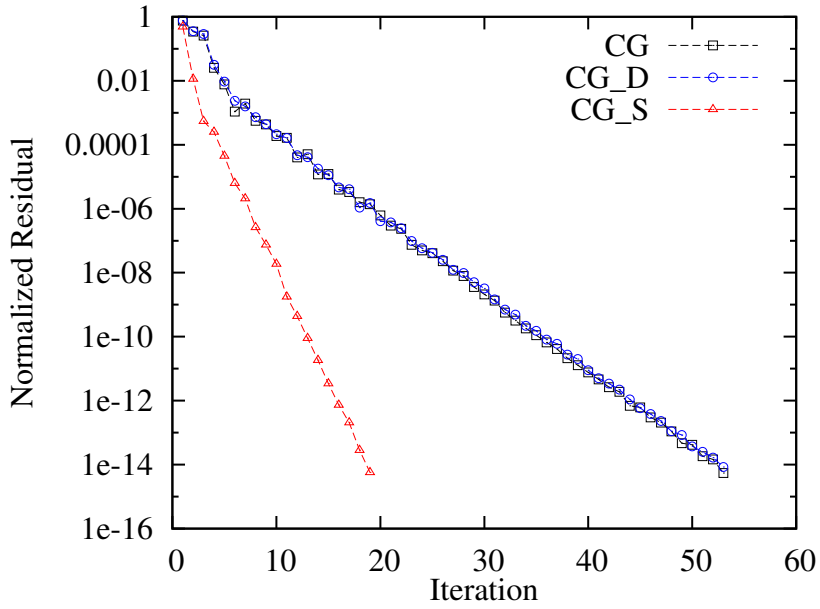

Figure 1: The relative residual as a function of iteration numbers for matrix *gsmat1*.

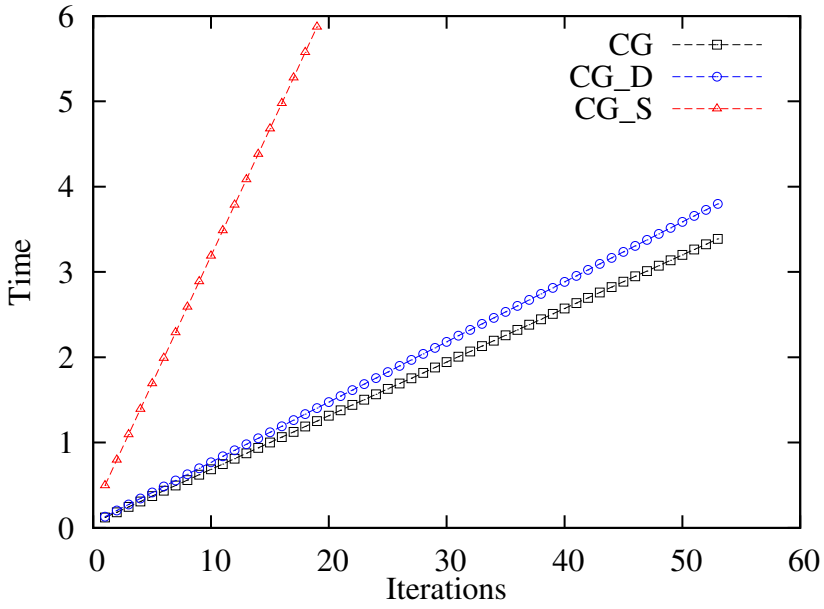

Figure 2: The computing time as a function of iteration numbers for matrix *gsmat1*.

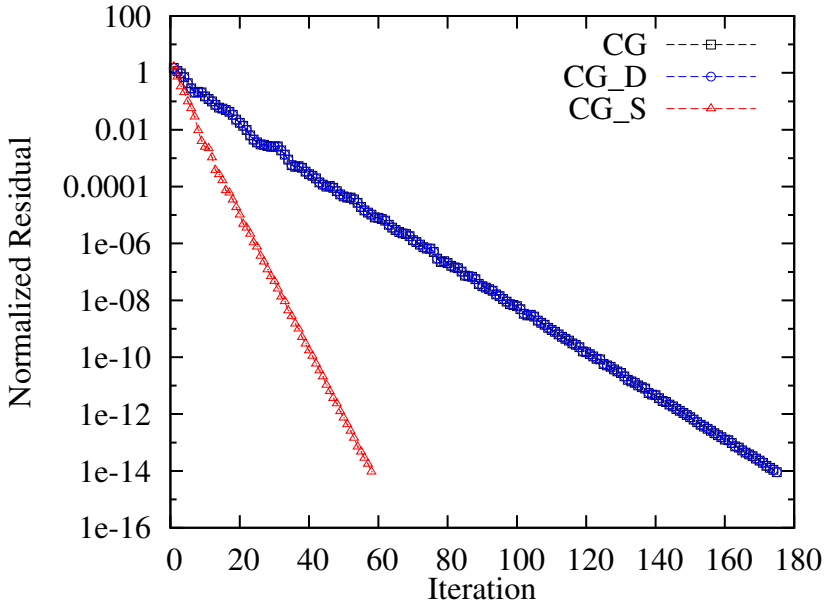

Figure 3: The relative residual as a function of iteration numbers for matrix *laplace3d*.

The computation time for CG, CG-D and CG-S is plotted as a function of iteration in the Fig. 2. We see that though the CG-S required more iteration numbers to converge, it took the least time to reach the convergence. CG-D required the same number of iterations but took a little more time to converge. This is because the preconditioning brought more calculations into the algorithm. Since the multiplication with diagonal matrix is quite simple, the extra time it takes is also just a bit. However, the CG-S, though needed only 19 iterations to converge, took almost twice the time of CG. This because the CG-S algorithm involves in many more complicated calculations, like computing $\alpha_j L^{-1} A d_j$ and $L^{-T}\hat{r}_{j+1}$ which need to be done by solving a system of linear equation. The matrix L has more complicated structure than D, which has also contributed to the long computation time.

We then carried out the numerical experiment on the matrix *laplace3d*. The residual *v.s.* iteration number is plotted in Fig. 3. Similar with the Fig. 1, the CG and CG-D completely overlap with each other, and took nearly 180 iterations to converge. The CG-S took only 60 iterations to converge, showing a faster convergence speed. Compare with the result in Fig. 1, CG-S saved more iterations for the matrix *laplace3d*.

The computation time is plotted as a function of the iteration in Fig. 4. We can see that though the CG has the least computation time, the CG-S took only a little more time. This implies that when the matrix A is more ill-preconditioned, the advantage that requiring fewer iterations to converge may make CG-S converge finally faster than CG.

Finally we used our three algorithms to solve the third large sparse matrix *c6h6-vh*. We plot the residual as a function of iteration in Fig. 5. This matrix is much larger and more complicated. The CG method took more than 6000 iterations to converge. The plot of CG fluctuates greatly, which actually wastes a lot of iterations and computation time. The preconditioning with diagonal matrix has greatly improved the convergence. CG-D took 453 iterations to converge. The CG-S took only 152 iterations to converge.

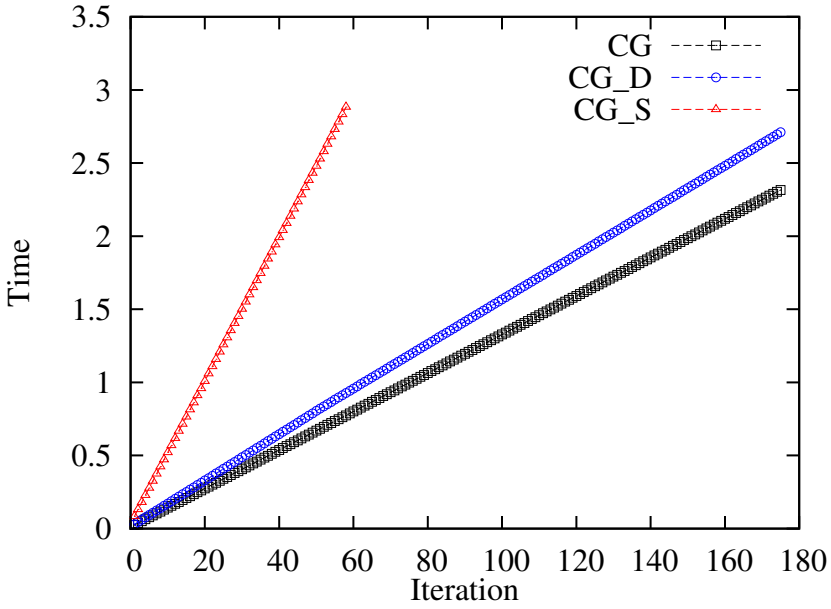

Figure 4: The computing time as a function of iteration numbers for matrix *laplace3d*.

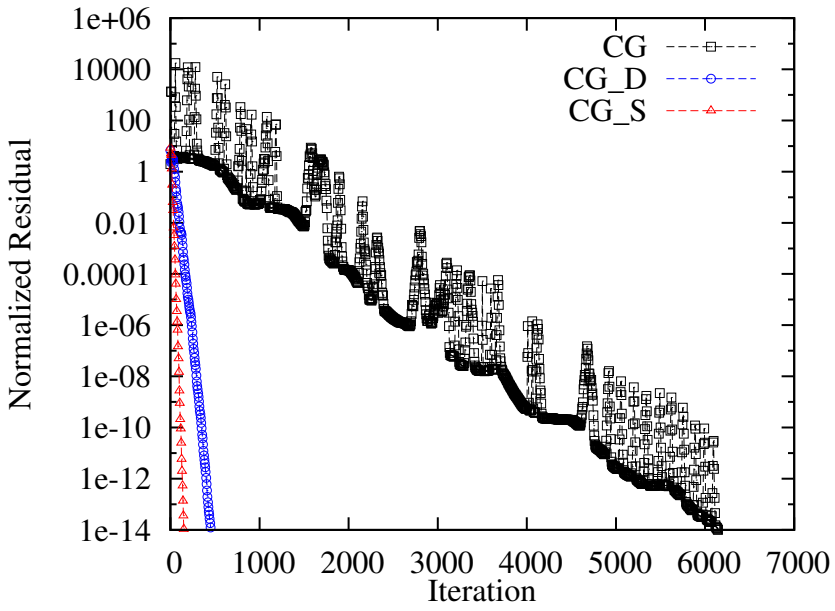

Figure 5: The relative residual as a function of iteration numbers for matrix *c6h6-vh*.

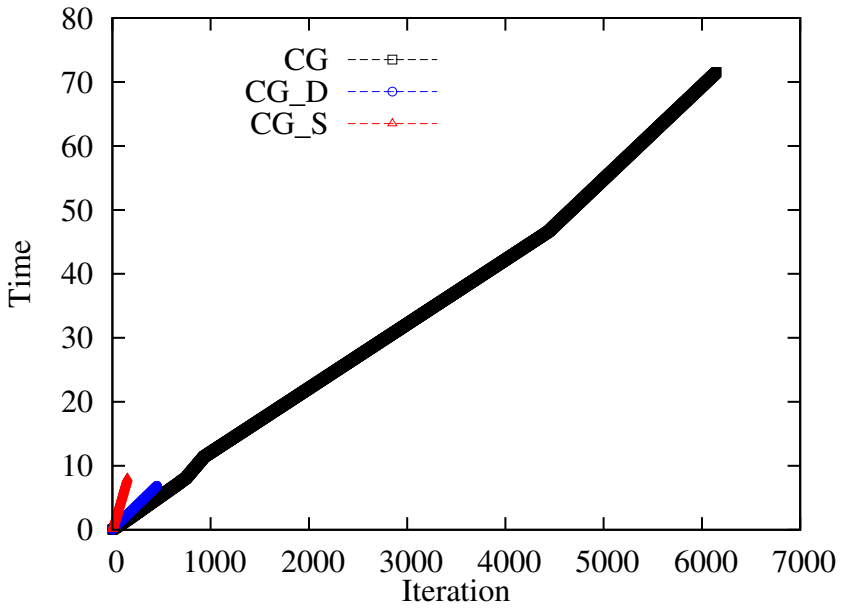

Figure 6: The computing time as a function of iteration numbers for matrix *c6h6-vh*.

The computation time the three algorithms took for solving the matrix *c6h6-vh* is plotted as a function of iteration in Fig. 6. In this figure we see that though the slop of CG remains the smallest, meaning it tooks least time for a single iteration, it needs 6148 iterations to converge. The total computation time for CG is 7 times as long as that for CG-D and CG-S. Though each iteration needs less time, CG-D needs 3 times as many as the iterations needed for CG-S, which makes the final computation time of CG-D and CG-S comparable.

5 Summary and Conclusions

In this project, we reviewed the major methods for solving systems of linear equations (nonlinear equations can be reduced into linear equations as well). We focused our study on the conjugate gradient (CG) method and different types of preconditioned conjugate gradient (PCG) methods. We derived mathematically the algorithm of the CG and PCG. We wrote Fortran90 code to implement the algorithms. We derived two types of PCG algorithms: (1) CG-D, the PCG with preconditioner being the diagonal matrix D, and (2) CG-S, the PCG with split preconditioners being the lower triangular matrix L (left preconditioner) and its transpose L^T (right preconditioner), where the L is obtained from the incomplete Cholesky factorization of the coefficient matrix A.

We also discussed the three kinds of storage approaches for the large sparse matrix, and wrote a subroutine `ichol` in Fortran90 code for the incomplete Cholesky factorization for matrix in CSR format. We compare the efficiency of `ichol` with that of the provided subroutine `ICholeski` by comparing their computation time for decomposing three different large sparse matrices. The results show that `ICholeski` needs less time for the decomposition, and thus is more efficient. `ichol` is also very efficient. Considering other factors, we use our own subroutine `ichol` to perform the factorization of the three matrices. Finally, we carried out numerical experiments, using the three algorithms to solve three different large sparse matrices in CRS format, and compared the convergence speeds and the computation time of the three algorithms.

Based on the analysis on the algorithm and the numerical experiment results, we draw the major conclusions below:

1. Preconditioning usually can improve the convergence speed, making the algorithm converge over fewer iterations.

2. Preconditioning makes the algorithm more complicated and require more calculations for a single iteration, thus increases the computation cost for one single iterations.

3. For well-conditioned matrix, both CG and PCG converge very quickly. CG is cheaper in overall computation cost than PCG. However, if the matrix is very ill-conditioned, PCG can converge much faster, and needs much less computation time than CG.

4. The split preconditioning with L and L^T resulting from incomplete Cholesky factorization has better preconditioning effect than left preconditioning with diagonal matrix D.

Acknowledgements

We would like to express our deep appreciation to Professor Eric Polizzi, the instructor of this very interesting and extremely useful course *ECE697NA*, who has taught us many important numerical methods and inspired us to learn more, who spent much time on answering our questions and gave us many helpful suggestions on this project.

We also want to thank each other. We both work actively on this project, making the cooperation an enjoyable one.

References

[1] Constantine Pozrikidis. *Numerical Computation in Science and Engineering*. Oxford, 2008.

[2] Magnus Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, 1952.

[3] Gene H. Golub and Dianne P. O'Leary. Some history of the conjugate gradient and lanczos algorithms:1948-1976*. *Siam Reivew*, 31(1):50–102, 1989.

[4] Yanbin Jia. Lecture note on conjugate gradient method, 2007.

[5] Yousef Saad. *Iterative methods for sparse linear systems*. Siam, 2003.

[6] Wikipedia. Incomplete cholesky factorization, 2015.

Appendix A: Code

```
program hw4

!!!!!!! variable declaration
  implicit none
  integer :: i,j,N,it,itmax,k,Nx,Ny,e,nnz, nl
  integer :: ie
  integer :: t1,t2,tim
  double precision :: hx,hy,L,alpha,beta,pi,normr,normb,nres,err
  double precision,dimension(:),allocatable :: sa,b,r,dummy,x,sl,z,diag,lsa
  integer,dimension(:),allocatable :: isa,jsa, lisa, ljsa
  character(len=100) :: name,algo
  !! for banded solver
  double precision,dimension(:,:),allocatable :: ba
  double precision :: nzero,norm
  integer :: kl,ku,info
  !!for pardiso
  integer(8),dimension(64) :: pt
  integer,dimension(64) :: iparm
  integer :: mtype
  integer :: MAXFCT,MNUM,PHASE,MSGLVL
  integer :: idum
  double precision,dimension(:),allocatable :: usa,d
  integer,dimension(:),allocatable :: uisa,ujsa


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  !! read command line argument name="1","2","3","4","5", "6" or "7"
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  call getarg(1,name)
  call getarg(2,algo)


!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!! Read matrix "name" is csr format (3 arrays/3 files) !!!!!!!!!!!!!!!
!!!!!!!!! Create isa, jsa, sa arrays !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  open(10,file=trim(name)//'.isa',status='old')
  read(10,*) N
  allocate(isa(1:N+1))
  do i=1,N+1
     read(10,*) isa(i)
  end do
  close(10)


  open(10,file=trim(name)//'.jsa',status='old')
  read(10,*) nnz
  allocate(jsa(1:nnz))
  do i=1,nnz
     read(10,*) jsa(i)
  end do
  close(10)


  open(10,file=trim(name)//'.sa',status='old')
  read(10,*) nnz
  allocate(sa(1:nnz))
  do i=1,nnz
     read(10,*) sa(i)
  end do
  close(10)

  print *,'#','Matrix:  ',trim(name)
  print *,'#','N=',N
  print *,'#','nnz=',nnz
  print *,''
!!!!!!!!!!!!!


  allocate(b(1:N))! ! Right hand-side
  b=1.0d0

  allocate(x(1:N)) !! solution
  x=0.0d0 ! initial starting vector


  allocate(r(1:N)) !! residual array

  itmax=7000 ! iteration maximal allowed
  err=1d-14 !convergence criteria


  allocate(dummy(1:N))!! dummy array if additional storage is needed

  select case(algo)

  case("1")
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

```fortran
!!!!!!!!   Conjugate Gradient !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

  call system_clock(t1,tim)

! compute initial residual
     r=b
     call dcsrmm('F',N,N,1,-1.0d0,sa,isa,jsa,x,1.0d0,r)
     nres=sum(abs(r))/sum(abs(b)) ! norm relative residual

     ! search direction initial
     allocate(d(1:N))
     d=r

     it=0
     do while ((nres>err).and.(it<itmax))
        it=it+1 ! # of iterations

       ! A*d
        call dcsrmm('F',N,N,1,1.0d0,sa,isa,jsa,d,0.0d0,dummy)

       ! alpha=r^t.r/(d^t.A.d)
        alpha=sum(r**2)/sum(d*dummy)

       ! x(k)=x(k-1)+alpha*d(k-1) (new iterate)
        do i=1,N
           x(i)=x(i)+alpha*d(i)
        enddo

        ! beta=1/(r^t.r)
        beta=1.d0/sum(r**2)

        ! new residual r=r-alpha*A*d
        do i=1,N
           r(i)=r(i)-alpha*dummy(i)
        enddo

        ! norm relative residual
        nres=sum(abs(r))/sum(abs(b))

        !beta value updated
        beta=beta*sum(r**2)

        !Update search direction d=r+beta*d
        do i=1,N
           d(i)=r(i)+beta*d(i)
        enddo

        call system_clock(t2,tim) ! final time

        print *,it,nres, (t2-t1)*1.0d0/tim
     end do



  case("2")
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!Preconditioned Conjugate Gradient (Diagonal) !!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!! extract diagonal matrix (store the inverse of the diagonal elements)
   allocate(diag(1:N))
   do k=1,N
      do i=isa(k), isa(k+1)-1
         if (jsa(i) == k) then
            diag(k)=1.d0/sa(i)
            exit
         else
            diag(k)=0.d0
         endif
      enddo
   enddo

  call system_clock(t1,tim)

! compute initial residual
     r=b
     call dcsrmm('F',N,N,1,-1.0d0,sa,isa,jsa,x,1.0d0,r)
     nres=sum(abs(r))/sum(abs(b)) ! norm relative residual
     ! compute z_0 stored in z
     allocate(z(1:N))
     z=r*diag

     ! search direction initial
     allocate(d(1:N))
     d=z

     it=0
     do while ((nres>err).and.(it<itmax))
        it=it+1 ! # of iterations

       ! A*d
```

```fortran
        call dcsrmm('F',N,N,1,1.0d0,sa,isa,jsa,d,0.0d0,dummy)

       ! alpha=r^t.z/(d^t.A.d)
        alpha=sum(r*z)/sum(d*dummy)

       ! x(k)=x(k-1)+alpha*d(k-1) (new iterate)
        x=x+alpha*d

        ! beta=1/(r^t.z)
        beta=1.d0/sum(r*z)

        ! new residual r=r-alpha*A*d
        r=r-alpha*dummy

        ! new z z=M^-1.r
        z=r*diag

        ! norm relative residual
        nres=sum(abs(r))/sum(abs(b))

        !beta value updated
        beta=beta*sum(r*z)

        !Update search direction d=z+beta*d
        d=z+beta*d

        call system_clock(t2,tim) ! final time

        print *,it,nres, (t2-t1)*1.0d0/tim
     end do


  case("3")
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!Preconditioned Conjugate Gradient (split preconditioner) !!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!! incomplete factorization of A and store L in sl
  allocate(sl(1:nnz))
  sl=sa

  call ichol(sl,jsa,isa,N)

  call system_clock(t1,tim)

! compute initial residual
     r=b
     call dcsrmm('F',N,N,1,-1.0d0,sa,isa,jsa,x,1.0d0,r)
     nres=sum(abs(r))/sum(abs(b)) ! norm relative residual

     ! compute r_hat=L^-1 r (residual initial)
     call dcsrsv('L',N,1,sl,isa,jsa,r)

     ! compute d_0 = L^-T r_hat (search direction initial)
     allocate(d(1:N))
     d=r
     call dcsrsv('U',N,1,sl,isa,jsa,d)

     it=0
     do while ((nres>err).and.(it<itmax))
        it=it+1 ! # of iterations

       ! A*d
        call dcsrmm('F',N,N,1,1.0d0,sa,isa,jsa,d,0.0d0,dummy)

       ! alpha=r^t.r/(d^t.A.d)
        alpha=sum(r**2)/sum(d*dummy)

       ! x(k)=x(k-1)+alpha*d(k-1) (new iterate)
        do i=1,N
           x(i)=x(i)+alpha*d(i)
        enddo

        ! beta=1/(r^t.r) before r being overwritten and for use later on
        beta=1.d0/sum(r**2)

        ! compute dummy=L^-1.A.d
        call dcsrsv('L',N,1,sl,isa,jsa,dummy)

        ! new residual r=r-alpha*A*d
        r=r-alpha*dummy

        ! compute the original residual dummy=L.r_hat
        call dcsrmm('L',N,N,1,1.0d0,sl,isa,jsa,r,0.0d0,dummy)

        ! norm relative residual
        nres=sum(abs(dummy))/sum(abs(b))

        !beta value updated
        beta=beta*sum(r**2)

        ! compute dummy=L^-T.r
```

```
          dummy=r
          call dcsrsv('U',N,1,sl,isa,jsa,dummy)

          !Update search direction d=L^-T.r+beta*d
          d=dummy+beta*d

          call system_clock(t2,tim) ! final time

          print *,it,nres, (t2-t1)*1.0d0/tim
       end do

    case("4")
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!! Comparison of the two Algo. of incomplete Cholesky factorization !!!!!!!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

!!!! my own algorithm as in subroutine ichol
!!!! incomplete factorization of A and store both L and L^T in sl
    call system_clock(t1,tim)
    allocate(sl(1:nnz))
    sl=sa

    call ichol(sl,jsa,isa,N)
    call system_clock(t2,tim)
          print *,"Time used by ichol is:", (t2-t1)*1.0d0/tim
    deallocate(sl)
!!!! the algorithm provided, ICholesky.o
!!!! The lower triangular of A needs to be extracted to store in another CSR
!!!! format lsa, lisa, ljsa. This is down by subroutine dcsr2csr_low, which is
!!!! provided in the library of dzlsprim. Then the subroutine ICholeski will
!!!! perform the decomposition on lsa, and store the matrix L in sl, which can
!!!! share the coordinate vectros, lisa and ljsa with lsa.

    call system_clock(t1,tim)
    nl=(nnz+n)/2  ! number of elem in lsa
    allocate(lsa(1:nl))
    allocate(lisa(1:N+1))
    allocate(ljsa(1:nl))
    allocate(sl(1:nl))

    ! extract the lower triangular
    call dcsr2csr_low(0,N,sa,isa,jsa,lsa,lisa,ljsa)

    ! decomposition
    call ICholeski(N,lsa,lisa,ljsa,sl)

    call system_clock(t2,tim) ! final time

    print *,"Time used by ICholeski is:", (t2-t1)*1.0d0/tim

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    end select

    print *,''
    if (it>=itmax) then
       print *,'#','**Attention** did not converge to', err, 'after', itmax,'iterations'
    else
       if (it/=0) then
          print *,'#','Success -- converge below', err, 'in', it,'iterations'
       else
          print *,'#','Direct solver -- residual', nres
       endif
    end if

    print *,'#','Total time',(t2-t1)*1.0d0/tim

end program hw4
```