A Set of Conjugate Gradient Routines for Real and Complex Arithmetics

Valérie Frayssé * Luc Giraud *

CERFACS Technical Report TR/PA/00/47 - July 2000

Abstract

In this report we describe the implementations of the preconditioned conjugate gradient (CG) algorithm for both real and complex, single and double precision arithmetics suitable for serial, shared memory and distributed memory computers. For the sake of simplicity, flexibility and efficiency the CG solvers have been implemented using the reverse communication mechanism for the matrix-vector product, the preconditioning and the dot product computations. Finally the implemented stopping criterion is based on a normwise backward error. After a short presentation of the CG algorithm we give a detailed description of the user interface.

Keywords : linear systems, Krylov methods, CG, reverse communication, distributed memory.

1 The Conjugate Gradient algorithm

1.1 General description

The Conjugate Gradient method was proposed in different versions in the early 50s in separate contributions by Lanczos [6] and Hestenes and Stiefel [5]. It becomes the method of choice for the solution of large sparse hermitian positive definite linear system and is the starting point of the extensive work on the Krylov methods [8].

Let $A = A^H$ (where A^H denotes the conjugate transpose of A) be a square nonsingular $n \times n$ complex hermitian positive definite matrix, and b be a complex vector of length n, defining the linear system

$$Ax = b \tag{1}$$

to be solved.

Let $x^{(0)} \in \mathbb{C}^n$ be an initial guess for this linear system, $r^{(0)} = b - Ax^{(0)}$ be its corresponding residual and M^{-1} be the preconditioner. The algorithm is classically described by

*CERFACS, 42 av. Gaspard Coriolis, 31057 Toulouse Cedex, France.
Email : fraysse,giraud@cerfacs.fr

$k = 0$
$r^{(0)} = b - Ax^{(0)}$
repeat
  $k = k + 1$
  Solve $Mz^{(k)} = r^{(k)}$
  if $k = 1$ **then**
    $p^{(1)} = z^{(0)}$
  else
    $\beta^{(k-1)} = {z^{(k-1)}}^H r(k-1)/{z^{(k-2)}}^H r(k-2)$
    $p^{(k)} = z^{(k-1)} + \beta^{(k-1)} p^{(k-1)}$
  end if
  $q^{(k)} = Ap^{(k)}$
  $\alpha^{(k)} = {z^{(k-1)}}^H r(k-1)/{p^{(k)}}^H q^{(k)}$
  $x^{(k)} = x^{(k-1)} + \alpha^{(k)} p^{(k)}$
  $r^{(k)} = r^{(k-1)} - \alpha^{(k)} q^{(k)}$
until convergence

Algorithm 1: Conjugate Gradient

The conjugate gradient algorithm constructs the minimum error solution in A-norm over the Krylov space $\mathcal{K}_k = \text{span}\left\{r^{(0)}, Ar^{(0)}, \ldots, A^{k-1}r^{(0)}\right\}$. It exists a rich literature dedicated to this method: for more details we, non-exhaustively, refer to [2, 4, 7] and the references therein.

1.2 Relationship with the symmetric Lanczos matrix

There is a close relationship between the conjugate gradient method and the Lanczos algorithm for the solution of eigenproblem [4]. The Lanczos algorithm applied to a matrix A constructs a tridiagonal matrix whose extremal eigenvalues converge to the extremal eigenvalues of A usually denoted $\lambda_{\min}(A)$ and $\lambda_{\max}(A)$. Thanks to the relationship between Lanczos and Conjugate Gradient, the entries of the tridiagonal matrix can be recovered from the computed information produced by CG at very low extra cost. More precisely the entries of the symmetric tridiagonal Lanczos matrix T can be recovered from the conjugate gradient iterations by:

$$T_{k,k} = \frac{1}{\alpha^{(k)}} + \frac{\beta^{(k-1)}}{\alpha^{(k-1)}}$$

and

$$T_{k-1,k} = -\frac{\sqrt{\beta^{(k-1)}}}{\alpha^{(k-1)}}.$$

When computed from preconditioned iterations, the extremal eigenvalues of the tridiagonal Lanczos matrix converge to the extremal eigenvalues of the preconditioned matrix $M^{-1}A$. Therefore one can get an estimation of the condition number of the preconditioned linear system defined by $\dfrac{\lambda_{min}(M^{-1}A)}{\lambda_{Max}(M^{-1}A)}$ by computing the smallest and largest eigenvalues of the tridiagonal matrix.

1.3 Stopping criteria

We have chosen to base our stopping criterion on the normwise backward error [3]. The backward error analysis, introduced by Givens and Wilkinson [9], is a powerful concept for analyzing the quality of an approximate solution:

1. it is independent from the details of round-off propagation: the error introduced during the computation are interpreted in terms of perturbations of the initial data, and the computed solution is considered as exact for the perturbed problem;

2. because round-off errors are seen as data perturbations, they can be compared with errors due to numerical approximations (consistency of numerical schemes) or to physical measurements (uncertainties on data coming from experiments for instance).

The backward error measures in fact the distance between the data of the initial problem and those of the perturbed problem; therefore it relies upon the choice of the data allowed to vary and the norm to measure these variations. In the context of linear systems, classical choices are the normwise and the component-wise perturbations [3]. These choices lead to explicit formulas for the backward error (often a normalized residual) which is then easily evaluated. For iterative methods, it is generally admitted that the normwise model of perturbation is appropriate [1].

Let $x^{(m)}$ be an approximation of the solution $x = A^{-1}b$. Then the quantity

$$\begin{aligned} \eta(x^{(m)}) &= \min\left\{\varepsilon > 0;\ \|\Delta A\|_2 \leq \varepsilon\alpha,\ \|\Delta b\|_2 \leq \varepsilon\beta \text{ and } (A+\Delta A)x^{(m)} = b + \Delta b\right\} \\ &= \frac{\left\|b - Ax^{(m)}\right\|_2}{\alpha\left\|x^{(m)}\right\|_2 + \beta} \end{aligned}$$

is called the *normwise backward error* associated with $x^{(m)}$. The best one can require from an algorithm is a backward error of the order of the machine precision. In practice, the approximation of the solution is acceptable when its backward error is lower than the uncertainty on the data. Therefore, there is no gain in iterating after the backward error has reached machine precision (or data accuracy). The unpreconditioned residual is iteratively computed by the algorithm via a short recurrence. Therefore, the backward error can be obtained at a low cost and we can use

$$\eta_{CG} = \frac{\left|r^{(m)}\right|}{\alpha\left\|x^{(m)}\right\|_2 + \beta}$$

as the stopping criterion of the CG iterations. However, it is well-known that, in finite precision arithmetic, the computed residual given by the short recurrence may differ significantly from the true residual. Therefore, it is not safe to use exclusively η_{CG} as the stopping criterion. Our strategy is the following: first we iterate until η_{CG} becomes lower than the tolerance, then afterwards, we iterate until η becomes itself lower than the tolerance. We hope in this way to minimize the number of residual computations (involving the computation of matrix-vector product) necessary to evaluate η, while having a reliable stopping criterion.

How to choose α and β ? Classical choices for α and β that appear in the literature are $\alpha = \|A\|_2$ and $\beta = \|b\|_2$. Any other choice that reflects the possible uncertainty on the data can also be plugged in. In our implementation, default values are used when the user's input is $\alpha = \beta = 0$. Table 1 lists the stopping criteria for different choices of α and β.

2 Implementation of CG

2.1 The user interface

For the sake of simplicity and portability, the CG implementation is based on the reverse communication mechanism

α	β	Information on the unpreconditioned system
0	0	$\frac{\left\|Ax^{(m)}-b\right\|_2}{\|b\|_2}$
0	$\neq 0$	$\frac{\left\|Ax^{(m)}-b\right\|_2}{\beta}$
$\neq 0$	0	$\frac{\left\|Ax^{(m)}-b\right\|_2}{\alpha\left\|x^{(m)}\right\|_2}$
$\neq 0$	$\neq 0$	$\frac{\left\|Ax^{(m)}-b\right\|_2}{\alpha\left\|x^{(m)}\right\|_2+\beta}$

Table 1: Stopping criterion for the CG method

- for implementing the numerical kernels that depend on the data structure of the matrix A and the preconditioners,
- for performing the dot products.

This last point has been implemented to allow the use of CG in a parallel distributed memory environment, where only the user knows how he has spread his data. We have one driver per arithmetic, and we use the BLAS and LAPACK terminology that is

`DRIVE_SCG`	for real single precision arithmetic computation,
`DRIVE_DCG`	for real double precision arithmetic computation,
`DRIVE_CCG`	for complex single precision arithmetic computation,
`DRIVE_ZCG`	for complex double precision arithmetic computation.

Finally, to hide as much as possible the numerical method from the user, only a few parameters are required by the drivers, whose interfaces are similar for all arithmetics. Below we present the interface for the real double precision driver:

```
CALL DRIVE_DCG(N,NLOC,LWORK,WORK,IRC,ICNTL,CNTL,INFO,RINFO)
```

`N` is an INTEGER variable that must be set by the user to the order n of the matrix A. It is not altered by the subroutine.

`NLOC` is an INTEGER variable that must be set by the user to the size of the subset of entries of b and x that are allocated to the calling process in a distributed memory environment. For serial or shared memory computers `NLOC` should be equal to `N`. It is not altered by the subroutine.

`LWORK` is an INTEGER variable that must be set by the user to the size of the workspace `WORK`. `LWORK` must be greater than or equal to `6*NLOC+1` if the estimation of the extremal eigenvalues is not requested, `6*NLOC+1+2*Itermax` otherwise. It is not altered by the subroutine.

`WORK` is a SINGLE/DOUBLE PRECISION REAL/COMPLEX array of length `LWORK`. The first `NLOC` entries contain the initial guess x_0 in input and the computed approximation of the unpreconditioned solution in output. The nexnext `NLOC` entries contain

the right-hand side b of the unpreconditioned system. The remaining entries are used as workspace by the subroutine.

`IRC` is an INTEGER array of length 5 that need not be set by the user. This array controls the reverse communication. Details of the reverse communication management are given in Section 2.2.

`ICNTL` is an INTEGER array of length 7 that contains control parameters that must be set by the user. Details of the control parameters are given in Section 2.3.

`CNTL` is a SINGLE/DOUBLE PRECISION REAL array of length 3 that contains control parameters that must be set by the user. Details of the control parameters are given in Section 2.3.

`INFO` is an INTEGER array of length 3 which contains information on the reasons of exiting CG. Details are given in Section 2.4.

`RINFO` is a SINGLE/DOUBLE PRECISION REAL array of length 3 that contains the backward error for the linear system and the estimation of the extremal eigenvalues if requested.

2.2 The reverse communication management

The INTEGER array `IRC` permits to implement the reverse communication. None of its entries need be set by the user.
On each exit, `IRC(1)` indicates the action that must be performed by the user before invoking the driver again. Possible values of `IRC(1)` and the associated actions are as follows:

`0` Normal exit.
`1` The user must perform the matrix vector product $z \leftarrow Ax$.
`2` The user must perform the right preconditioning $z \leftarrow M_i^{-1}x$.
`3` The user must perform the product of x^H and y $z \leftarrow x^H y$.

On each exit with `IRC(1) >` 0, `IRC(2)` and `IRC(3)` indicate the index in `WORK` where x and y should be read and `IRC(4)` indicates the index in `WORK` where z should be written.

We refer to Section 4 for an example of use of the driver routine.

2.3 The control parameters

The entries of the array `ICNTL` control the execution of the `DRIVE_CG` subroutine. All entries of `ICNTL` are input parameters.

`ICNTL(1)` is the stream number for the error messages.
`ICNTL(2)` is the stream number for the warning messages. No warning message if set to 0.
`ICNTL(3)` is the stream number for the convergence history. No convergence history if set to 0.
`ICNTL(4)` indicates if a preconditioner will be used
`ICNTL(5)` controls whether the user wishes to supply an initial guess of the solution vector. If ICNTL(5)=0, the initial guess is set to zero.
`ICNTL(6)` is the maximum number of iterations allowed.
`ICNTL(7)` indicates if eigenvalues estimation is requested.

Possible values for `ICNTL(4)` are

`0` no preconditioning,

1 preconditioning,
2 error, default set in the routine INIT_CG.

Possible values for ICNTL(7) are

0 no eigenvalue estimation requested, default set in the routine INIT_CG.
1 estimation of the extremal eigenvalues of the preconditioner system is requested.

The entries of the CNTL array define the tolerance and the normalizing factors (see Section 1.3) that control the execution of the algorithm:

CNTL(1) is the convergence tolerance for the backward error (see Section 1.3 for details).
CNTL(2) is the normalizing factor α.
CNTL(3) is the normalizing factor β.

2.4 The information parameters

Once IRC(1) = 0, the entries of the array INFO explain the circumstances under which CG was exited. All entries of INFO are output parameters.

Possible values for INFO(1) are

0 normal exit. Convergence has been observed.
-1 erroneous value $n < 1$.
-2 LWORK too small.
-3 convergence not achieved after ICNTL(6) iterations.
-4 illegal preconditioning information provided.

If INFO(1) = 0, then INFO(2) contains the number of iterations performed until achievement of the convergence and INFO(3) gives the minimal size for workspace. If INFO(1) = -3, then INFO(2) contains the minimal size necessary for the workspace.

If INFO(1) = 0, then RINFO(1) contains the backward error for the linear system. Additionnally, if the estimation has been erquested (i.e. INFO(7) = 1) RINFO(2) contains an estimation of the smallest eigenvalue of the preconditioned system and RINFO(3) an estimation of the largest eigenvalue. Thus an estimation of the condition number of the preconditioned system can be computed.

2.5 Initialization of the parameters

An initialization routine is available to the user for each arithmetic:

INIT_SCG for real single precision arithmetic computation,
INIT_DCG for double precision arithmetic computation,
INIT_CCG for complex single precision arithmetic computation,
INIT_ZCG for complex double precision arithmetic computation.

These routines set the input control parameters ICNTL and CNTL defined above to default values. The generic interface is

```
CALL INIT_CG(ICNTL,CNTL)
```

The default value for

`ICNTL(1)` is 6,
`ICNTL(2)` is 6,
`ICNTL(3)` is 0: no convergence history,
`ICNTL(4)` is 2: undefined preconditioner,
`ICNTL(5)` is 0: default initial guess $x_0 = 0$,
`ICNTL(6)` is -1: the user must specify explicitly the maximum number of iterations,
`ICNTL(7)` is 0: no estimation of the extremal eigenvalues,
`CNTL(1)` is 10^{-6},
`CNTL(2)` is 0,
`CNTL(3)` is 0.

3 Availability of the software

The code is written in FORTRAN 77 and makes calls to BLAS and LAPACK routines, as indicated in Tables 2 and 3. The code is free for non-commercial use only. The source code is available from the WEB at the URL

`http://www.cerfacs.fr/algor/`

together with the software license agreement.

Simple precision		Double precision	
real	complex	real	complex
SAXPY	CAXPY	DAXPY	ZAXPY
SCOPY	CCOPY	DCOPY	ZCOPY

Table 2: BLAS routines called in CG

Simple precision		Double precision	
real	complex	real	complex
SSTEV	CSTEV	DSTEV	ZSTEV

Table 3: LAPACK routines called in CG

4 An example of use

```
*************************************************************
**                 TEST PROGRAMME FOR THE CG CODE
*************************************************************

      program validation
*
      integer lda,  lwork
      parameter (lda = 1000)
      parameter (lwork = 8*lda+2)
*
      integer i, j, n
      integer revcom, colx, coly, colz
      integer irc(7), icntl(7), info(3)
*
      integer matvec, precondLeft,  dotProd
      parameter (matvec=1, precondLeft=2, dotProd=3)
*
      integer nout
*
      complex*16  a(lda,lda), work(lwork)
      real*8  cntl(5), rinfo(2)
*
      complex*16 ZERO, ONE, aux
      parameter (ZERO = (0.0d0,0.0d0), ONE = (1.0d0,0.0d0))
*
      complex*16 zdotc
      external zdotc
      intrinsic sqrt
*
*************************************************************
** Generate the test matrix a and set the right-hand side
** in positions (n+1) to 2n of the array work.
** The right-hand side is chosen such that the exact solution
** is the vector of all ones.
*************************************************************
*
      write(*,*) '*****************************************'
      write(*,*) 'This code is an example of use of CG'
      write(*,*) 'in single precision real arithmetic'
      write(*,*) 'Results are written in output files'
      write(*,*) 'fort.60 : log file of CG iterations '
      write(*,*) 'and sol_Testcg : output of the computation.'
      write(*,*) '*****************************************'
      write(*,*)
      write(*,*) 'Matrix size < ', lda
      read(*,*) n
      if (n.gt.lda) then
        write(*,*) 'You are asking for a too large matrix'
        goto 100
      endif
*
*  Initialiaze the matrix and the right-hand side or load them
*  from a file
  .....
*
*
*************************************************
** Initialize the control parameters to default value
*************************************************
* setup the monitoring CG variables to default values
       call init_zcg(icntl,cntl)
*
* Default initial guess
       icntl(5) = 0
*
* Define the tolerance for the stopping criterion
       cntl(1)  = 1.0 d  -9
* Define the stream for the convergence history
       icntl(3) = 60
* Define the maximum number of iterations
       icntl(6) = n+1
* Ask for the estimation of the condition number (if icntl(1) == 1)
       icntl(7) = 1

*
***************************************
```

```
** Reverse communication implementation
****************************************
*
10     call drive_zcg(n,n,lwork,work,
     &         irc,icntl,cntl,info,rinfo)

       revcom = irc(1)
       colx   = irc(2)
       coly   = irc(3)
       colz   = irc(4)
*
       if (revcom.eq.matvec) then
* perform the matrix vector product for the CG iteration
*        work(colz) <-- A * work(colx)
         call zgemv ('N', n, n, ONE, A, lda,
     &            work(colx), 1, ZERO , work(colz), 1)
           goto 10
*
        else if (revcom.eq.precondleft) then
*        work(colz) <-- M * work(colx)
           call zcopy(n,work(colx),1,work(colz),1)
           goto 10
        else if (revcom.eq.dotProd) then
*      perform the scalar product for the CG iteration
*      work(colz) <-- work(colx) work(coly)
*
         work(colz)= zdotc(n, work(colx),1, work(coly),1)
         goto 10
       endif
*
*
        if (icntl(7).eq.1) then
          write(*,*) ' Estimation of the smallest eig. ', rinfo(2)
          write(*,*) ' Estimation of the biggest  eig. ', rinfo(3)
        endif
*
*****************************
* dump the solution on a file
*****************************
     .....

*
      stop
      end
```

References

[1] M. Arioli, I. S. Duff, and D. Ruiz. Stopping criteria for iterative solvers. *SIAM J. Matrix Anal. Appl.*, 13:138–144, January 1992.

[2] O. Axelsson. *Iterative Solution Methods*. Cambridge University Press, Cambridge, 1994.

[3] F. Chaitin-Chatelin and V. Frayssé. *Lectures on Finite Precision Computations*. SIAM, Philadelphia, 1996.

[4] G. H. Golub and C. Van Loan. *Matrix computations*. Johns Hopkins University Press, 1996. Second edition.

[5] M. R. Hestenes and E. L. Stiefel. Methods of conjugate gradients for solving linear systems. *J. Res. Nat. Bur. Stand.*, 49:409–436, 1952. Section B.

[6] C. Lanczos. Solution od systems of linear equations by minimized iterations. *J. Res. Nat. Bur. Stand.*, 49:33–53, 1952.

[7] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, Boston, 1996.

[8] Y. Saad and H. A. van der Vorst. Iterative solution of linear systems in the 20-th century. Tech. Rep. UMSI-99-152, University of Minnesota, 1999.

[9] J. H. Wilkinson. *Rounding errors in algebraic processes*, volume 32. Her Majesty's stationery office, London, 1963.