Preconditioned Conjugate-Gradient Solver

Aleksandar Donev

June 2001

1 Module Conjugate_Gradient

This documentation is out of date and not finished!

This module implements the conjugate gradient for the solution of a linear system $Ax = b$, where $A$ is symmetric and positive semidefinite or definite, via the routine PreconditionedConjugateGradient. In this particular version I do not explicitly use calls to the BLAS level-1 routines, simply because most compilers today can do the needed optimizations themselves, and I also use some of the Fortran 90 intrinsics such as DOT_PRODUCT. Make sure though your compiler implements these well!

There are some serious issues surrounding the design of user-interfaced routines such as PCG in Fortran, and these I will document somewhere else. The solution implemented here is a compromise between efficiency and elegance, though as F2K compilers become proficient the approach can be switched entirely toward elegance. Details are given below. One of these F2K fixes is the introduction of IMPORT, which is not present in F95 and therefore I resort to making a separate module, named CG_Data_Types to hold the definitions of the derived data-types. However, I make all of these public through Conjugate_Gradient, so the user need not ever use CG_Data_Types.

This module uses allocatable components to achieve elegance. This is an TR-15581 extension to Fortran 95 and is not yet implemented in all compilers. Even though Lahey LF95 6.0 implements it, their implementation is basically a “syntactic-sugar” fix that masks internal array pointers as allocatable arrays. The bad thing is that the optimizer does not know this but rather treats the arrays as array pointer and thus assumes aliasing, which makes for poor optimization and costly buffer use. To avoid these issues I do all intensive operations inside routines which accept assumed-shape arrays as arguments (these are implemented well in almost all compilers by now), and then call these with the allocatable or pointer arrays as arguments. This bypasses the aliasing problem relatively effectively. For these reasons, I give here a bunch of trivial vector-operations routines such as vector addition, multiplication, axpy etc. These could also be wrappers around certain BLAS-1 routines, though I see no reason for that with today’s compilers.

"WEAVE.f90" 1.0.0.1

MODULE CG_Data_Types
USE Precision
USE Error_Handling
USE System_Monitors
IMPLICIT NONE
PUBLIC :: CG_Errors, CG_Timers, CG_Solver, SPD_Matrix, CG_System
PRIVATE

INTEGER, PARAMETER, PUBLIC :: CG_free_solver = 0, CG_inactive_solver = -1,
                                CG_active_solver = 1
INTEGER, PARAMETER, PUBLIC :: CG_unconverged = -1, CG_converged = -2, CG_working = 0

TYPE CG_Errors
  REAL(KIND=r_wp) :: relative_error = 10.0*EPSILON(1.0,r_wp), residual_norm = -1.0,r_wp,
                    largest_residual = -1.0,r_wp
ENDTYPE

TYPE CG_Timers
  INTEGER :: preconditioning_timer = -1, dot_timer = -1, vector_timer = -1,
             multiplication_timer = -1       // For timing the CG components
ENDTYPE

TYPE CG_Solver     // Need not be EXTENSIBLE
  INTEGER :: niterations = 0, max_iterations = 0, min_iterations = -1
  INTEGER :: status = CG_free_solver
  INTEGER :: convergence = CG_unconverged
  _TYPE(CG_Errors) :: errors
  _TYPE(CG_Timers) :: timers
  INTEGER :: log_unit = 0
  INTEGER(KIND=i_wp) :: lb = 1, ub = 0       // Lower and upper bounds for arrays
  REAL(KIND=r_wp), DIMENSION (:), DYNAMIC :: NULLIFIED(cg_Vx), NULLIFIED(cg_Vy),
                                           NULLIFIED(cg_Residuals)
  REAL(KIND=r_wp), DIMENSION (:), POINTER :: cg_x = NULL  // x-the solution
ENDTYPE

TYPE SPD_Matrix     // EXTENSIBLE in F2x
  INTEGER :: PIN = 1       // Integer handle
  // PROCEDURE (Multiply), POINTER :: MatrixVectorMultiplication = NULL in F2x
  // PROCEDURE (Precondition), POINTER :: Preconditioner = NULL in F2x
ENDTYPE

TYPE CG_System      // EXTENSIBLE in F2x
  INTEGER :: PIN = 1       // Integer handle-not in F2x
  _TYPE(SPD_Matrix), POINTER :: matrix = NULL       // CLASS (SPD_Matrix) in F2x
  _TYPE(CG_Solver), POINTER :: solver = NULL       // May be CLASS (CG_Solver) in F2x
  REAL(KIND=r_wp), DIMENSION (:), POINTER :: cg_b = NULL  // The rhs vector
ENDTYPE

END MODULE CG_Data_Types

MODULE Conjugate_Gradient
  USE Precision
  USE Error_Handling
  USE System_Monitors
  USE CG_Data_Types
  USE Vector_Operations
  IMPLICIT NONE
  PUBLIC :: PreconditionedConjugateGradient
PRIVATE
1.1 Preconditioned Serial Conjugate Gradient

This routine uses the iterative PCG method to solve the system $Ax = b$, where $x$ is the vector of unknowns and the right hand side vector $b$ is the load $\text{rhs\_vector}$. It assumes a suitable initial guess $x_0$ has already been provided in $\text{unknowns}$, which is intent $\text{INOUT}$!

The main argument to $\text{PreconditionedConjugateGradient}$ is a matrix of type $\text{CG\_System}$, which in F2K would be an extensible type later to be extended with any user data-structures used to describe the coefficient matrix $A$ and the preconditioner $M$ (or $M^{-1}$). Here I use the integer matrix $\text{matrix}$ Pin instead of a generic pointer, and the system being solved can be named.

The argument $\text{matrix}$ also contains a pointer (it is a pointer so that one solver can be used in several systems (not simultaneously though)) to an instance of a solver of type $\text{CG\_Solver}$. This solver contains all the variables that are needed by CG, including all array temporaries, and all of these should be either assigned values or left with the default ones before calling PCG, and especially, all the array temporaries should be allocated. There are four such temporaries:

$\text{cg\_Vx}$ and $\text{cg\_Vy}$ $\text{cg\_Vz}$ will be the left vector, $V_y$ the right one in the matrix-vector multiplication $V_z = AV_y$. Note that only these vectors will be used in the matrix-vector multiplication, so that the user can allocate them as he pleases, for example, he may choose to put shadows on them. Only the range $(n_{\text{jb}} : n_{\text{ub}})$ will be used of these arrays, any additional space can be used as the user pleases. The matrix-vector multiplication itself should be provided in the procedure argument $\text{MatrixVectorMultiplication}$, which accepts a matrix to the system being solved. Since Fortran 90 does not provide the tools needed to create generic handles (i.e. generic pointers), I had to make the handles integers, which can then be used to identify systems as the user wishes.

$\text{cg\_Vz}$ and $\text{cg\_residual}$ the residual $r = \text{cg\_residual}$ will contain the residual errors at the end and will be used as the preconditioning vector in $V_z = M^{-1}r$. The same principles as above apply--only the range $(n_{\text{jb}} : n_{\text{ub}})$ of these arrays will be used and the preconditioning will be done via calls to the preconditioning routine $\text{Preconditioning}$.

The other components of the type $\text{CG\_Solver}$ are relatively simple to understand. The errors in the sub-type $\text{CG\_Errors}$, residual norm and largest residual give the desired upper bound on the magnitude of the $\text{cg\_residuals}$ vector $r$ when terminating CG. The first value, residual norm, gives the value of $\sqrt{r^T r}$ used for checking termination (norm-2 of $r$), while largest residual gives the infinity norm of $r$, i.e. max $r_i$. Please note that evaluating these errors requires additional work, which is why they are optional, namely, a negative value indicates the error should not be calculated in convergence tests. If preconditioning is not used, then the 2-norm is calculated inside CG, but when preconditioning is used, the $M^{-1}$-norm of $r$ is calculated, i.e. $\sqrt{r^T \hat{M}^{-1} r}$. By default, CG will compare the ratio $\epsilon = \frac{\sqrt{r^T \hat{M}^{-1} r}}{\sqrt{r^T M^{-1} r}}$, where $r_0$ is the initial residual, with the tolerance $\text{relative\_error}$ when checking termination. This is a
cheap and usually OK test, and so it is always performed. If you do not want to check this, just set the
tolerance relative_error to 1.0 and then use either residual_norm or largest_residual or both to check
convergence (at the extra cost of evaluating these).

If the component log_unit is a non-zero value, then it is assumed that it gives the unit number of a
file to which the error-versus-iteration history for the iterative solver is recorded. The relative error
\( e \) is always recorded along with the iteration number, and if requested the 2- and inf- norm are also
reported after these. Also optional are timers that time the different portions that comprise CG, given
as an derived type of 4 clock numbers \( \text{CG\_Timers} \), where the different clocks upon return will contain
the total expended time in each of these operations:

preconditioning_timer for timing the preconditioner

multiplication_timer for timing dot products

vector_timer times vector additions and similar BLAS level-1 operations

multiplication_timer will time the matrix-vector products

A negative (out-of-range) value for either one of these indicates that the related timing should not be
performed.

Other components of \( \text{CG\_Solver} \) include the maximum allowed number of iterations max_iterations
and the actual performed number of iterations n_iterations. Also given are two status variables, status
and convergence, which give the availability status of a given solver and the convergence status, as
defined with the (enumerated) integer parameter value in \( \text{CG\_Data\_Types} \). The following macros will
make coding rthe CG subroutine much easier and independent of the type \( \text{CG\_Solver} \).

\[
\begin{align*}
\text{cm} \ cg\_relative\_error & \text{ solver } \% \text{ errors } \% \text{ relative_error} \\
\text{cm} \ cg\_residual\_norm & \text{ solver } \% \text{ errors } \% \text{ residual}\_norm \\
\text{cm} \ cg\_largest\_residual & \text{ solver } \% \text{ errors } \% \text{ largest}\_residual \\
\text{cm} \ cg\_log\_unit & \text{ solver } \% \text{ log}\_unit \\
\text{cm} \ n\_max\_iterations & \text{ solver } \% \text{ max}\_iterations \\
\text{cm} \ n\_min\_iterations & \text{ solver } \% \text{ min}\_iterations \\
\text{cm} \ Vx & \text{ solver } \% \ cg\_Vx_{nb:nub} \\
\text{cm} \ Vy & \text{ solver } \% \ cg\_Vy_{nb:nub} \\
\text{cm} \ Vz & \text{ solver } \% \ cg\_Vz_{nb:nub} \\
\text{cm} \ residuals & \text{ solver } \% \ cg\_residuals_{nb:nub} \\
\text{cm} \ unknowns & \text{ solver } \% \ cg\_x_{nb:nub} \\
\text{cm} \ rhs\_vector & \text{ system } \% \ cg\_b_{nb:nub}
\end{align*}
\]

1.1.1 Procedure PreconditionedConjugateGradient

Here is an unpolished, non parallel-optimal version of PCG:

\[
\text{(PCG 1.1.1) } \equiv \\
\text{ SUBROUTINE PreconditionedConjugateGradient (system, MatrixVectorMultiplication, } \\
\text{ Preconditioner) }
\]
IMPLICIT NONE
_TYPE(CG_System), INTENT(INOUT) :: system  // class(CG_System) in F2x
OPTIONAL :: Preconditioner  // Not in F2x
INTERFACE  // Outside in module in F2x
  SUBROUTINE MatrixVectorMultiplication(system)  // A procedure declaration in F2x
    use CG_Data_Types  // import:: CG_System in F2x
    _TYPE(CG_System), INTENT(INOUT) :: system  // class(CG_System) in F2x
  END SUBROUTINE MatrixVectorMultiplication
  SUBROUTINE Preconditioner(system)  // A procedure declaration in F2x
    use CG_Data_Types  // import:: CG_System in F2x
    _TYPE(CG_System), INTENT(INOUT) :: system  // class(CG_System) in F2x
  END SUBROUTINE Preconditioner
END INTERFACE

_TYPE(CG_Solver), POINTER :: solver  // A temporary
REAL(KIND=r_wp) :: relative_error_tolerance, norm_2_tolerance, norm_inf_tolerance, eps, alpha, beta, gamma, rho_new, rho_old, residual, norm_rhs  // CG auxiliary variables
INTEGER :: iteration, p_timer, d_timer, v_timer, m_timer  // Counters and timers
INTEGER(KIND=i_wp) :: variable, n_ub, n_lb  // Counters
LOGICAL :: log_convergence, use_preconditioning, time_p, time_d, time_v, time_m, converged, check_2_norm, check_inf_norm  // Logical indicators

  solver => system % solver  // Just a shortcut
  n_lb = solver % lb
  n_ub = solver % ub
  solver % status = CG_active_solver  // Solver in use
  solver % convergence = CG_workin  // Starting work
  p_timer = (solver % timers % preconditioning_timer)
  time_p = (p_timer > 0)
  d_timer = (solver % timers % dot_timer)
  time_d = (d_timer > 0)
  v_timer = (solver % timers % vector_timer)
  time_v = (v_timer > 0)
  m_timer = (solver % timers % multiplication_timer)
  time_m = (m_timer > 0)

  use_preconditioning = PRESENT(Preconditioner)
  log_convergence = (cg_log_unit /= 0)
  eps = 10.0_r_wp * EPSILON(1.0_r_wp)  // Numerical rounding tolerance
  relative_error_tolerance = MAX(10.0_r_wp * eps, cg_relative_error)
  check_2_norm = (cg_residual_norm > eps)
  if (check_2_norm) norm_2_tolerance = MAX(10.0_r_wp * eps, cg_residual_norm)
  check_inf_norm = (cg_largest_residual > eps)
  if (check_inf_norm) norm_inf_tolerance = MAX(10.0_r_wp * eps, cg_largest_residual)

  /* Initialize the variables—remember that an initial guess for x should already be provided in unknowns before calling CG: */
  if (time_v) CALL StartTimer(v_timer)
    // Instead of Vx = unknowns (to avoid aliasing issues)
    CALL VectorCopy(target = Vx, source = unknowns)  // Vx = x0
    CALL StopTimer(v_timer)
  if (time_m) CALL StartTimer(m_timer)

5
CALL MatrixVectorMultiplication(system)  // \( V_y = A x_0 \)
IF (time,\_n) CALL StopTimer(m\_timer)
IF (time,\_v) CALL StartTimer(v\_timer)
CALL VectorSubtraction(from = rhs\_vector, what = V_y, difference = residuals)  // \( r_0 = b - A x_0 \), instead of residuals = rhs\_vector - V_y
IF (time,\_v) CALL StopTimer(v\_timer)

/* Start the CG iteration loop: */
iteration = 0

CG: DO

iteration = iteration + 1
IF (use\_preconditioning) THEN  // With preconditioning
  IF (time,\_p) CALL StartTimer(p\_timer)
  CALL Preconditioner(system)  // \( Y_2 = M^{-1} r \)
  IF (time,\_p) CALL StopTimer(p\_timer)
  IF (time,\_d) CALL StartTimer(d\_timer)
  rho\_new = DOT\_PRODUCT(residuals, Vz)  // \( \rho = r^T M^{-1} r \)
  IF (time,\_d) CALL StopTimer(d\_timer)
ELSE  // No preconditioning
  IF (time,\_d) CALL StartTimer(d\_timer)
  rho\_new = DOT\_PRODUCT(residuals, residuals)  // \( \rho = r^T r \)
  IF (time,\_d) CALL StopTimer(d\_timer)
END IF

IF (iteration \( \equiv 1 \)) norm\_rhs = SQRT(rho\_new)  // Initial \( M^{-1} \)-norm of the residual
IF (use\_preconditioning) THEN  // Test for convergence
  cg\_relative\_error = SQRT(rho\_new) / ABS(norm\_rhs + eps)
ELSE
  cg\_relative\_error = SQRT(rho\_new) / ABS(norm\_rhs + eps)
END IF
IF (log\_convergence) THEN  // Write convergence history to a file
  WRITE(unit = cg\_log\_unit, fmt = "(E10,E10.3)", advance = "NO") iteration, cg\_relative\_error
END IF
IF (check\_2\_norm) THEN
  IF (~use\_preconditioning) THEN
    cg\_residual\_norm = SQRT(rho\_new)
  ELSE
    cg\_residual\_norm = SQRT(DOT\_PRODUCT(residuals, residuals))
    // An additional dot product
  END IF
IF (log\_convergence) THEN  // Write convergence history to a file
  WRITE(unit = cg\_log\_unit, fmt = "(E10.3)", advance = "NO") cg\_residual\_norm
END IF
END IF
END IF
IF (check\_inf\_norm) THEN
  cg\_largest\_residual = 0.0,\_wp
  DO variable = J\_BOUND(residuals, i\_wp), J\_BOUND(residuals, i\_wp)  // n\_lb, n\_ub
    // Better use a loop here than MAXVAL due to ABS
    residual = solver % cg\_residuals\_variable
  END DO
END IF

6
IF (ABS(residual) \geq cg_{\text{largest\_residual}}) \text{ cg}_{\text{largest\_residual}} = \text{ABS}(\text{residual})
END DO

IF (log\_convergence) THEN  // Write convergence history to a file
WRITE (UNIT = cg_{log\_unit}, FMT = "(E10.3)", ADVANCE = "NO") cg_{largest\_residual}
END IF
END IF

TestConvergence: IF (iteration > n_{max\_iterations}) THEN  // Test for convergence
solver \% convergence = CG_{unconverged}
EXIT CG
ELSE IF (cg_{relative\_error} < relative\_error\_tolerance) THEN
converged = T
IF (check_{2\_norm}) THEN
IF (cg_{residual\_norm} > norm_{2\_tolerance}) converged = F
END IF
IF (check_{inf\_norm}) THEN
IF (cg_{largest\_residual} > norm_{inf\_tolerance}) converged = F
END IF
IF (converged \land (iteration > n_{min\_iterations})) THEN  
/* Now we save the current status of the solver: */
solver \% convergence = CG_{converged}
EXIT CG
END IF
END IF TestConvergence

IF (log\_convergence) THEN  // A new line
WRITE (cg_{log\_unit}, *)
END IF

IF (use\_preconditioning) THEN  // With preconditioning
IF (time\_v) CALL StartTimer(v\_timer)
IF (iteration \equiv 1) THEN  // First iteration has \beta = 0
CALL VectorCopy(target = Vx, source = Vz)  // V_x = V_z
//// V_x = V_z
ELSE
beta = rho_{new} / rho_{old}
CALL VectorAXPY(beta = beta, y = Vx, x = Vz)  // V_x = V_y + \beta V_x
//// V_x = V_z + \beta \times V_x
END IF
IF (time\_v) CALL StopTimer(v\_timer)
ELSE  // No preconditioning
IF (time\_v) CALL StartTimer(v\_timer)
IF (iteration \equiv 1) THEN  // Initialize
CALL VectorCopy(target = Vx, source = residuals)  // V_x = r
//// V_x = residuals
ELSE
beta = rho_{new} / rho_{old}
CALL VectorAXPY(beta = beta, y = Vx, x = residuals)  // V_x = V_y + \beta V_x
//// V_x = residuals + \beta \times V_x
END IF
IF (time\_v) CALL StopTimer(v\_timer)
END IF
END IF

7
rho_old = rho_new       // Save for later

IF (time_m) CALL StartTimer(m_timer)
CALL MatrixVectorMultiplication(system)    // V_y = AV_x
IF (time_m) CALL StopTimer(m_timer)

IF (time_d) CALL StartTimer(d_timer)
gamma = DOT_PRODUCT(Vx, Vy)
alpha = rho_new / gamma  // α = \frac{\rho_{new}}{\gamma} AV_x
IF (time_d) CALL StopTimer(d_timer)

IF (time_v) CALL StartTimer(v_timer)
CALL VectorAXPY(alpha = alpha, x = Vx, y = unknowns)    // x = x + αV_x
    // unknowns = unknowns + α * Vx
CALL VectorAXPY(alpha = -alpha, x = Vy, y = residuals)
    // r = r − αV_y. There may be loss of accuracy in this statement
    // use restarts if needed (residuals = residuals − α * Vy)
IF (time_v) CALL StopTimer(v_timer)

END DO CG

solver % n_iterations = iteration − 1
solver % status = CG_inactive_solver     // Done

END SUBROUTINE PreconditionedConjugateGradient

This code is used in section 1.0.0.1.
CASE_TYPE TYPE
PRIVATE PRIVATE
SIZE(array, kind,...)
荆州 (#0, 0, INT(SIZE(array), KIND=kind), INT(SIZE(array,#), KIND=kind))
荆州 (#0, 0, INT(MAXLOC(array), KIND=kind), INT(MAXLOC(array,#), KIND=kind))
荆州 (#0, 0, INT(MINLOC(array), KIND=kind), INT(MINLOC(array,#), KIND=kind))
荆州(array, kind,...)荆州 (#0, 0, INT(LBOUND(array, DIM=1), KIND=kind),
荆州(array, #), KIND=kind))
荆州(array, kind,...)荆州 (#0, 0, INT(BOUND(array, DIM=1), KIND=kind),
荆州(array, #), KIND=kind))
荆州_deata inherence(generic_name,...)
END INTERFACE generic_name
荆州 Declare i_word(...)
荆州 i_word := #.
荆州 Declare i_wp(...)
荆州 (KIND = i_wp) := #.
荆州 Declare r_wp(...)
荆州 (KIND = r_wp) := #.
荆州 Declare r_sp(...)
荆州 (KIND = r_sp) := #.
荆州 Declare r_dp(...)
荆州 (KIND = r_dp) := #.
荆州 FullExtent(rank) :#DO (DIM, 2, rank) { : }
荆州 VarSequence(variable, start, end)
荆州#start#DO (DIM, #eval(start + 1), end) { , variable@DIM }
荆州 NestedLoopStart(variable, array, rank, kind) $DO (DIM, rank, 1, -1) { DO
荆州#start#DIM = LBOUND(array, kind, DIM = DIM), LBOUND(array, kind, DIM = DIM) }
荆州 NestedLoopEnd(rank) $DO (DIM, 1, rank) { END DO }