User's Guide to AGMG

Yvan Notay *

Service de Métrologie Nucléaire
Université Libre de Bruxelles (C.P. 165/84)
50, Av. F.D. Roosevelt, B-1050 Brussels, Belgium.
email : ynotay@ulb.ac.be

February 2008
Revised: March 2009, March 2010, July 2011
January 2012, October 2012

Abstract

This manual gives an introduction to the use of AGMG, a program which implements the algebraic multigrid method described in [Y. Notay, *An aggregation-based algebraic multigrid method*, Electronic Trans. Numer. Anal. (27: 123–146), 2010] with further improvements from [A. Napov and Y. Notay, *An algebraic multigrid method with guaranteed convergence rate*, SIAM J. Sci. Comput., vol. 34, pp. A1079-A1109, 2012] and from [Y. Notay, *Aggregation-based algebraic multigrid for convection-diffusion equations*, SIAM J. Sci. Comput., 2012 (to appear)].

This method solves algebraic systems of linear equations, and is expected to be efficient for large systems arising from the discretization of scalar second order elliptic PDEs. The method may however be tested on any problem, as long as *all diagonal entries of the system matrix are positive*. It is indeed purely algebraic; that is, no information has to be supplied besides the system matrix and the right-hand-side.

Both a sequential and a parallel FORTRAN 90 implementations are provided, as well as an interface allowing to use the software as a matlab function.

The software package can be downloaded from
`http://homepages.ulb.ac.be/~ynotay/AGMG`

Key words. FORTRAN, Matlab, multigrid, linear systems, iterative methods, AMG, preconditioning, parallel computing, software.

AMS subject classification. 65F10

*Supported by the Belgian FNRS ("Directeur de recherches")

```
! COPYRIGHT (c) 2012 Universite Libre de Bruxelles (ULB)
!
! ALL USAGE OF AGMG IS SUBJECT TO LICENSE. PLEASE REFER TO THE FILE "LICENSE".
! IF YOU OBTAINED A COPY OF THIS SOFTWARE WITHOUT THIS FILE,
! PLEASE CONTACT ynotay@ulb.ac.be
!
! In particular, if you have a free academic license:
!
! (1) You must be a member of an educational, academic or research institution.
!     The license agreement automatically terminates once you no longer fulfill
!     this requirement.
!
! (2) You are obliged to cite AGMG in any publication or report as:
!     "Yvan Notay, AGMG software and documentation;
!      see http://homepages.ulb.ac.be/~ynotay/AGMG".
!
! (3) You may not make available to others the software in any form, either
!     as source or as a precompiled object.
!
! (4) You may not use AGMG for the benefit of any third party or for any
!     commercial purposes. Note that this excludes the use within the
!     framework of a contract with an industrial partner.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! DICLAIMER:
!    AGMG is provided on an "AS IS" basis, without any explicit or implied
!    WARRANTY; see the file "LICENSE" for more details.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!   If you use AGMG for research, please observe that your work benefits
!   our past research efforts that allowed the development of AGMG.
!   Hence, even if you do not see it directly, the results obtained thanks
!   to the use of AGMG depend on the results in publications [1-3] below,
!   where the main algorithms used in AGMG are presented and justified.
!   It is then a normal duty to cite these publications (besides citing
!   AGMG itself) in any scientific work depending on the usage of AGMG,
!   as you would do with any former research result you are using.
!
! [1] Y. Notay, An aggregation-based algebraic multigrid method,
!    Electronic Transactions on Numerical Analysis, vol. 37, pp. 123-146, 2010
!
! [2] A. Napov and Y. Notay, An algebraic multigrid method with guaranteed
!    convergence rate, SIAM J. Sci. Comput., vol. 34, pp. A1079-A1109, 2012.
!
! [3] Y. Notay, Aggregation-based algebraic multigrid for convection-diffusion
!    equations, SIAM J. Sci. Comput., vol. 34, pp. A2288-A2316, 2012.
```

Contents

1 Introduction

The software package provides subroutines, written in FORTRAN 90, which implement the method described in [6], with further improvements from [5, 7]. The main driver subroutine is `AGMG` for the sequential version and `AGMGPAR` for the parallel version. Each driver is available in double precision (prefix `D`), double complex (prefix `Z`), single precision (prefix `S`) and single complex (prefix `C`) arithmetic. These subroutines are to be called from an application program in which are defined the system matrix and the right-hand-side of the linear system to be solved.

A matlab interface is also provided; that is, an m-file (agmg.m) implementing a matlab function (*agmg*) that allows to call (sequential) AGMG from the matlab environment. This guide is directed towards the use of the FORTRAN version; however, Section 2.2 on additional parameters, Section 2.4 on printed output and Sections 4–5 about some special usages apply to the matlab version as well. Basic information can also be obtained by entering *help agmg* in the matlab environment.

For a sample of performances in sequential and comparison with other solvers, see the report [8] (`http://homepages.ulb.ac.be/~ynotay/AGMG/numcompsolv.pdf`). For performances of the parallel version, see [6].

1.1 How to use this guide

This guide is self contained, but does not describe methods and algorithms used in AGMG, for which we refer to [6, 5, 7]. We strongly recommend their reading for an enlightened use of the package.

On the other hand, main driver routines are reproduced in the Appendix with the comment lines describing precisely each parameter. It is a good idea to have a look at the listing of a driver routine while reading the section of this guide describing its usage.

1.2 Release notes and backward compatibility

This guide describes AGMG 3.2.0-aca (academic version) and AGMG 3.2.0-pro (professional version).

The naming schemes and the calling sequence of each driver routine remain unchanged from releases 2.x, but there are slight differences with respect to releases 1.x. Whereas with releases 2.x routines were provided for backward compatibility, these are no longer supported; i.e. application programs based on the naming scheme and calling sequence of releases 1.x need to be adapted to the new scheme.

New features from **release 3.0** (the first three have no influence on the software usage):

- The aggregation algorithm is no longer the one described in [6]. It has been exchanged for the one in [5] in the symmetric case and for the one in [7] for general nonsymmetric matrices (Note that this is the user who tells if the problem is to be treated as a symmetric one or not via the parameter `nrest`, see Section 2.2.3 below).

  This is an "internal" change; i.e., it has no influence on the usage of the software. It makes the solver more robust and faster on average. We can however not exclude that in some cases AGMG 3.0 is significantly slower than previous versions. This may, in particular, happen for problems that are outside the standard scope of application. Feel free to contact the author if you have such an example. AGMG is an ongoing project and pointing out weaknesses of the new aggregation scheme can stimulate further progress.

- Another internal change is that the systems on the coarsest grid are now solved with the (sequential) MUMPS sparse direct solver [4]. Formerly, MUMPS was only used with the parallel version. Using it also in the sequential case allows to improve robustness; i.e. improved performances on "difficult" problems.

  The MUMPS library needs however not be installed when using the sequential version. We indeed provide an additional source file that contains needed subroutines from the MUMPS library. See Section 1.3 and the README file for details.

- The meaning of parameter `nrest` is slightly changed, as it influences also the aggregation process (see above). The rule however remains that `nrest` should be set to 1 only if the system matrix is symmetric positive definite; see Section 2.2.3 for details.

- It is now possible to solve a system with the transpose of the input matrix. This corresponds to further allowed values for parameter `ijob`; see Section 2.2.1 below for details.

- The meaning of the input matrix when using `ijob` not equal to its default is also changed; see Section 2.2.1 below for details.

New features from **release 3.1.1**:

- A few bugs that sometimes prevented the compilation have been fixed; the code (especially MUMPS relates routines) has been also cleaned to avoid superfluous warning messages with some compilers.

- The parameters that control the aggregation are now automatically adapted in case of too slow coarsening.

- Some limited coverage of singular systems is now provided, see Section 5.

New features from **release 3.1.2**:

- The printed output has been slightly changed: the work (number of floating points operations) is now reported in term of "work units per digit of accuracy", see Section 2.4 for details.

New features from **release 3.2.0**:

- *Professional version only. A variant of the parallel version is provided that does not require the installation of the parallel MUMPS library. See Section 1.3 and the README file for details.*

- The smoother is now SOR with automatic estimation of the relaxation parameter ω, instead of Gauss-Seidel (which corresponds to SOR with $\omega = 1$). In many cases, AGMG still uses $\omega = 1$ as before, but this new feature provides additional robustness, especially in non symmetric cases.

- Default parameters for the parallel version have been modified, to improve scalability when using many processors. This may induce slight changes in the convergence, that should however not affect overall performances.

- Some compilation issues raised by the most recent versions of gfortran and ifort have been solved, as well as some compatibility issues with the latest releases of the MUMPS library.

1.3 Installation and external libraries

AGMG does not need to be installed as a library. For the sake of simplicity, source files are provided, and it is intended that they are compiled together with the application program in which AGMG routines are referenced. See the README file provided with the package for additional details and example of use.

AGMG requires LAPACK [1] and BLAS libraries. Most compilers come with an optimized implementation of these libraries, and it is strongly recommended to use it. If not available, we provide needed routines from LAPACK and BLAS together with the package. Alternatively LAPACK may be downloaded from `http://www.netlib.org/lapack/` and BLAS may be downloaded from `http://www.netlib.org/blas/`

Note that these software are free of use, but copyrighted. It is required the following. *If you modify the source for a routine we ask that you change the name of the routine and comment the changes made to the original.* This, of course, applies to the routines we redistribute together with AGMG.

In addition, the parallel version requires the installation of the parallel MUMPS library [4], which is also public domain and may be downloaded from `http://graal.ens-lyon.fr/MUMPS/`.

Professional version only. The parallel MUMPS library needs not be installed when using the variant provided in the source files whose name contains "NM" (for "No MUMPS"; see the README file for details). Indeed, this variant uses only a sequential version of MUMPS, actually the same that is used by the sequential version of AGMG. Hence what is written below for sequential AGMG applies verbatim to this variant of the parallel version. Note that the performances of both variants may differ, and which one is the best is platform dependent. Hence if you have the parallel MUMPS library available, it is worth trying both. If you don't have, however, it is useful to install it only if you seemingly face scalability issues. Feel free to contact us for advice in such cases. For many concurrent tasks, the new "MUMPS-free" variant is expected to be better in most applications.

Despite the sequential version of AGMG also uses MUMPS, it does not require MUMPS to be installed as a library. Actually, in the sequential case, AGMG requires only a subset of MUMPS, and, for users' convenience, files (one for each arithmetic) gathering needed routines are provided together with AGMG. This is possible because MUMPS is public domain; see the files header for detailed copyright notice. Note that all MUMPS routines we provide with AGMG have been renamed (the prefixes `d` , `s` , . . . , have been exchanged for `dagmg_` , `sagmg_` , . . .). Hence you may use them in an application program that is also linked with the MUMPS library: there will be no mismatch; in particular, no mismatch between the sequential version we provide and the parallel version of MUMPS installed by default. On the other hand, it means that if you have the sequential MUMPS library installed and that you would like to use it instead of the version we provide, you need to edit the source file of AGMG and restore the classical calling sequence to MUMPS.

1.4 Error handling

Except the case of a number of allowed iterations insufficient to achieve the convergence criterion, any detected error is fatal and leads to a STOP statement, with a printed information that should allow to figure out what happened. Note that AGMG does not check the validity of all input parameters. For instance, **all diagonal entries of the supplied matrix must be positive**, but no explicit test is implemented. Feel free to contact the author for further explanations if errors persist after checking that the input parameters have been correctly defined.

1.5 Common errors

Be careful that `iter` and, in the parallel case, `ifirstlistrank` are also output parameters. Hence declaring them as a constant in the calling program entails a fatal error.

Also, don't forget to reinitialize `iter` between successive calls to AGMG or AGMGPAR.

2 Sequential version

2.1 Basic usage and matrix format

The application program has to call the main driver `AGMG` as follows.

```
call dagmg(N,a,ja,ia,f,x,ijob,iprint,nrest,iter,tol)
```

`N` is the order of the linear system, whose matrix is given in arrays `a`, `ja`, `ia`. The right hand side must be supplied in array `f` on input, and the computed solution is returned in `x` on output; optionally, `x` may contain an initial guess on input, see Section 2.2.1 below.

The required matrix format is the "compressed sparse row" (CSR) format described, e.g., in [10]. With this format, nonzero matrix entries (numerical values) are stored row-wise in `a`, whereas `ja` carries the corresponding column indices; entries in `ia` indicate then where every row starts. That is, nonzero entries (numerical values) and column indices of row i are located in `a`(k), `ja`(k) for k =`ia`(i),...,`ia`(i+1)-1. `ia` must have length (at least) `N`+1, and `ia`(`N`+1) must be defined in such a way that the above rule also works for i =`N`; that is, the last valid entry in arrays `a`, `ja` must correspond to index k =`ia`(`N`+1)-1.

By way of illustration, consider the matrix

$$A = \begin{pmatrix} 10 & & & -1 & \\ -2 & 11 & & -3 & \\ & -4 & 12 & & -5 \\ & & & 13 & \\ & -8 & -9 & & 14 \end{pmatrix} .$$

The compressed sparse row format of A is given (non uniquely) by the following three vectors.

$$\begin{aligned} \texttt{a} &= \left[\ 10.0 \quad -1.0 \,\middle|\, -2.0 \quad 11.0 \quad -3.0 \,\middle|\, -4.0 \quad 12.0 \quad -5.0 \,\middle|\, 13.0 \,\middle|\, 14.0 \quad -9.0 \quad -8.0 \ \right] \\ \texttt{ja} &= \left[\ 1 \quad 4 \,\middle|\, 1 \quad 2 \quad 4 \,\middle|\, 2 \quad 3 \quad 5 \,\middle|\, 4 \,\middle|\, 5 \quad 3 \quad 2 \ \right] \\ \texttt{ia} &= \left[\ 1 \quad 3 \quad 6 \quad 9 \quad 10 \quad 13 \ \right] \end{aligned}$$

As the example illustrates (see how is stored the last row of A), entries in a same row may occur in any order. AGMG performs a partial reordering inside each row, so that, on output, `a` and `ja` are possibly different than on input; they nevertheless still represent the same matrix.

Note that the SPARSKIT package [9] (`http://www-users.cs.umn.edu/~saad`) contains subroutines to convert various sparse matrix formats to CSR format.

2.2 Additional parameter

Other parameters to be supplied are `ijob` , `iprint` , `nrest` , `iter` and `tol`.

2.2.1 `ijob`

`ijob` has default value 0. With this value, the linear system will be solved in the usual way with the zero vector as initial approximation; this implies a setup phase followed by an iterative solution phase [6]. Non-default values allow to tell `AGMG` that an initial approximation is given in `x`, and/or to call `AGMG` for setup only, for solve only, or for just one application of the multigrid preconditioner (that is, a call to Algorithm 3.2 in [6] at top level, to be exploited in a more complex fashion by the calling program).

`ijob` can also be used to tell AGMG to work with the transpose of the input matrix.

Details are as follows (see also comments in the subroutine listing in Appendix A).

`ijob`	usage or remark
0	performs setup + solve + memory release, no initial guess
10	performs setup + solve + memory release, initial guess in `x`
1	performs setup only (preprocessing: prepares all parameters for subsequent solves)
2	solves only (based on previous setup), no initial guess
12	solves only (based on previous setup), initial guess in `x`
3	the vector returned in `x` is not the solution of the linear system, but the result of the action of the multigrid preconditioner on the right hand side in `f`
-1	erases the setup and releases internal memory
100,110,101,102,112	same as, respectively, 0,10,1,2,12 but use the TRANSPOSE of the input matrix
2,3,12,102,112	require that one has previously called AGMG with `ijob`=1 or `ijob`=101
1,101	the preconditioner defined when calling AGMG is entirely kept in internal memory; hence:
3	the input matrix is not accessed
2,12,102,112	the input matrix is used only to perform matrix vector product within the main iterative solution process

It means that, upon calls with `ijob`=2,12,102,112, the input matrix may differ from the input matrix that was supplied upon the previous call with `ijob`=1 or IJOB `ijob`=101; then AGMG attempts to solve a linear system with the "new" matrix using the preconditioner set up for the "old" one. The same remark applies to `ijob`$\geq$ 100 or not: the value `ijob`=1 or 101 determines whether the preconditioner set up and stored in internal memory is based on the matrix or its transpose; the value `ijob`=2,12 or 102,112 is used

to determine whether the linear system to be solved is with the matrix or its transpose, independently of the set up. Hence one may set up a preconditioner for a matrix and use it for its transpose.

These functionalities (set up a preconditioner and use it for another matrix) are provided for the sake of generality but should be used with care; in general, set up is fast with AGMG and hence it is recommended to rerun it even if the matrix changes only slightly.

Matlab: the usage is the same except that one should not specify whether an initial approximation is present in `x`. This is told via the presence or the absence of the argument. Hence, for instance, `ijob`=0 gathers the functionality of both `ijob`=0 and `ijob`=10.

2.2.2 `iprint`

`iprint` is the unit number where information (and possible error) messages are to be printed. A nonpositive number suppresses all messages, except the error messages which will then be printed on standard output.

2.2.3 `nrest`

`nrest` is the restart parameter for GCR [2, 11] (an alternative implementation of GMRES [10]), which is the default main iteration routine [6]. A nonpositive value is converted to 10 (suggested default).

If `nrest`=1, Flexible CG is used instead of GCR (when `ijob`=0,10,2, 12,100,110,102,112) and also (`ijob`=0,1,100,101) some simplification are performed during the set up based on the assumption that the input matrix is symmetric (there is then no more difference between `ijob`=1 and `ijob`=101). This is recommended if and only if the matrix is symmetric and positive definite.

2.2.4 `iter`

`iter` is the maximal number of iterations on input, and the effective number of iterations on output. Since it is both an input *and an output* parameter, it is important not to forget to reinitialize it between successive call.

2.2.5 `tol`

`tol` is the tolerance on the relative residual norm; that is, iterations will be pursued (within the allowed limit) until the residual norm is below `tol` times the norm of the input right-hand-side.

2.3 Example

The source file of the following example is provided with the package.

Listing 1: source code of sequential Example

```
      program example_seq
!
!  Solves the discrete Laplacian on the unit square by simple call to agmg.
!  The right-hand-side is such that the exact solution is the vector of all 1.
!
        implicit none
        real (kind(0d0)),allocatable :: a(:),f(:),x(:)
        integer,allocatable :: ja(:),ia(:)
        integer :: n,iter,iprint,nhinv
        real (kind(0d0)) :: tol
!
!       set inverse of the mesh size (feel free to change)
        nhinv=500
!
!       maximal number of iterations
        iter=50
!
!       tolerance on relative residual norm
        tol=1.e-6
!
!       unit number for output messages: 6 => standard output
        iprint=6
!
!       generate the matrix in required format (CSR)
!
!         first allocate the vectors with correct size
            N=(nhinv-1)**2
            allocate (a(5*N),ja(5*N),ia(N+1),f(N),x(N))
!         next call subroutine to set entries
            call uni2d(nhinv-1,f,a,ja,ia)
!
!       call agmg
!         argument 5 (ijob)  is 0 because we want a complete solve
!         argument 7 (nrest) is 1 because we want to use flexible CG
!                            (the matrix is symmetric positive definite)
!
         call dagmg(N,a,ja,ia,f,x,0,iprint,1,iter,tol)
!
      END program example_seq
```

2.4 Printed output

The above example produces the following output.

```
****ENTERING AGMG ************************************************************

****        Number of unknowns:      249001
****                  Nonzeros :     1243009 (per row:   4.99)

****SETUP: Coarsening by multiple pairwise aggregation
****  Rmk: Setup performed assuming the matrix symmetric
****       Quality threshold (BlockD):  8.00 ;  Strong diag. dom. trs: 1.29
****         Maximal number of passes:  2  ; Target coarsening factor: 4.00
****                       Threshold for rows with large pos. offdiag.: 0.45

****                       Level:           2
****        Number of variables:       62000          (reduction ratio: 4.02)
****                     Nonzeros:      309006 (per row: 5.0; red. ratio: 4.02)

****                       Level:           3
****        Number of variables:       15375          (reduction ratio: 4.03)
****                     Nonzeros:       76381 (per row: 5.0; red. ratio: 4.05)

****                       Level:           4
****        Number of variables:        3721          (reduction ratio: 4.13)
****                     Nonzeros:       18361 (per row: 4.9; red. ratio: 4.16)

****                       Level:           5
****        Number of variables:         899          (reduction ratio: 4.14)
****                     Nonzeros:        4377 (per row: 4.9; red. ratio: 4.19)
****        Exact factorization:       0.158 work units (*)

****                  Grid complexity:     1.33
****               Operator complexity:     1.33
****  Theoretical Weighted complexity:     1.92 (K-cycle at each level)
****    Effective Weighted complexity:     1.92 (V-cycle enforced where needed)

****         memory used (peak):     2.54 real(8) words per nnz (   24.11 Mb)
****       Setup time (Elapsed):     1.13E-01 seconds

****SOLUTION: flexible conjugate gradient iterations (FCG(1))
****     AMG preconditioner with 1 Gauss-Seidel pre- and post-smoothing sweeps
****         at each level
```

```
****  Iter=    0        Resid= 0.45E+02        Relat. res.= 0.10E+01
****  Iter=    1        Resid= 0.14E+02        Relat. res.= 0.32E+00
****  Iter=    2        Resid= 0.23E+01        Relat. res.= 0.51E-01
****  Iter=    3        Resid= 0.87E+00        Relat. res.= 0.19E-01
****  Iter=    4        Resid= 0.14E+00        Relat. res.= 0.31E-02
****  Iter=    5        Resid= 0.35E-01        Relat. res.= 0.78E-03
****  Iter=    6        Resid= 0.66E-02        Relat. res.= 0.15E-03
****  Iter=    7        Resid= 0.23E-02        Relat. res.= 0.52E-04
****  Iter=    8        Resid= 0.54E-03        Relat. res.= 0.12E-04
****  Iter=    9        Resid= 0.13E-03        Relat. res.= 0.28E-05
****  Iter=   10        Resid= 0.33E-04        Relat. res.= 0.74E-06
****  - Convergence reached in   10 iterations -

****     level 2   #call=    10   #cycle=    20   mean=   2.00    max=  2
****     level 3   #call=    20   #cycle=    40   mean=   2.00    max=  2
****     level 4   #call=    40   #cycle=    80   mean=   2.00    max=  2

****       Number of work units:    12.11 per digit of accuracy (*)
****   memory used for solution:     3.32 real(8) words per nnz (   31.44 Mb)
****    Solution time (Elapsed):     2.55E-01 seconds

*** (*) 1 work unit represents the cost of 1 (fine grid) residual evaluation
****LEAVING AGMG * (MEMORY RELEASED) ****************************************
```

Note that each output line issued by the package starts with `****`.

AGMG first indicates the size of the matrix and the number of nonzero entries. One then enters the setup phase, and the name of the coarsening algorithm is recalled, together with the basic parameters used. Note that these parameters need not be defined by the user: AGMG always use default values. These, however, can be changed by expert users, see Section 2.5 below.

The quality threshold is the threshold used to accept or not a tentative aggregate when applying the coarsening algorithms from [5, 7]; `BlockD` indicates that the algorithm from [5] is used (quality for block diagonal smoother), whereas `Jacobi` is printed instead when the algorithm from [7] is used (quality for Jacobi smoother). The strong diagonal dominance threshold is the threshold used to keep outside aggregation rows and columns that are strongly diagonally dominant; by default, it is set automatically according to the quality threshold as indicated in [5, 7]. The maximal number of passes and the target coarsening factor are the two remaining parameters described in these papers. In addition, nodes having large positive offdiagonal elements in their row or column are transferred unaggregated to the coarse grid, and AGMG print the related threshold.

How the coarsening proceeds is then reported level by level. The matrix on last level is factorized exactly, and the associated cost is reported in term of number of *work units*; that is, AGMG reports the number of floating point operations needed for this factorization divided by the number of floating point operations needed for one residual evaluation (at top level: computation of $\mathbf{b} - A\,\mathbf{x}$ for some $\mathbf{b}$, $\mathbf{x}$), which represents 1 work unit.

To summarize setup, AGMG then reports on "complexities". The grid complexity is the sum over all levels of the number of variables divided by the matrix size; the operator complexity is the complexity relative to the number of nonzero entries; that is, it is the sum over all levels of the number of nonzero entries divided by the number of nonzero entries in the input matrix (see [6, eq. (4.1)]). The theoretical weighted complexity reflects the cost of the preconditioner when two inner iterations are performed are each level; see [5, page 15] (with $\gamma = 2$) for a precise definition. The effective weighted complexity corrects the theoretical weighted complexity by taking into account that V-cycle is enforced at some levels according to the strategy described in [6, Section 3]. This allows to better control the complexity in cases where the coarsening is slow. In most cases, the coarsening is fast enough and both weighted complexities will be equal, but, when they differ, only the effective weighted complexity reflects the cost of the preconditioner as defined in AGMG.

Memory usage is reported in term of real*8 (double precision) words per nonzero entries in the input matrix; the absolute value is also given in Mbyte. Note that AGMG reports the peak memory; that is, the maximal amount used at any stage of the setup. This memory is dynamically allocated within AGMG. Eventually, AGMG reports on the real time elapsed during this setup phase.

Next one enters the solution phase. AGMG informs about the used iterative method (as defined via input parameter `nrest`), the used smoother, and the number of smoothing steps (which may also be tuned by expert users). How the convergence proceeds is then reported iteration by iteration, with an indication of both the residual norm and the relative residual norm (i.e., the residual norm divided by the norm of the right hand side). Note that values reported for "Iter=0" correspond to initial values (nothing done yet). When the iterative method is GCR, AGMG also reports on how many restarts have been performed.

Upon completion, AGMG reports, for each intermediate level, statistics about inner iterations; "#call" is the number of times one entered this level; "#cycle" is the cumulated number of inner iterations; "mean" and "max" are, respectively, the average and the maximal number of inner iteration performed on each call. If V-cycle formulation is enforced at some level (see [6, Section 3]), one will have #cycle=#call and mean=max=1.

Finally, the cost of this solution phase is reported, again in term of "work units", but now per digit of accuracy; that is, how many digit of accuracy have been gained is computed as $d = \log_{10}(\|\mathbf{r}\|_0/\|\mathbf{r}\|_f)$ (where $\mathbf{r}_0$ and $\mathbf{r}_f$ are respectively the initial and final residual vectors), and the total number of work units for the solution phase is divided by d to get the mean work needed per digit of accuracy. The code also reports the memory used during this phase and the elapsed real time. Note that the report on memory usage does

not take into account the memory internally allocated within the sparse direct solver at coarsest level (i.e., MUMPS).

2.5 Fine tuning

Some parameters referenced in [6, 5, 7] are assigned default values within the code, for the sake of simplicity. However, expert users may tune these values, to improve performances and/or if they experiment convergence problems. This can be done by editing the module `dagmg_mem`, at the top of the source file.

3 Parallel version

We assume that the previous section about the sequential version has been carefully read. Many features are common between both versions, such as most input parameters and the CSR matrix format (Section 2.1) and the meaning of output lines (Section 2.4). This information will not be repeated here, where the focus is on the specific features of the parallel version.

The parallel implementation has been developed to work with MPI and we assume that the reader is somewhat familiar with this standard.

The parallel implementation is based on a partitioning of the rows of the system matrix, and assumes that this partitioning has been applied before calling AGMGPAR. Thus, if the matrix has been generated sequentially, some partitioning tool like METIS [3] should be called before going to AGMGPAR.

The partitioning of the rows induces, inside AGMG, a partitioning of the variables. Part of the local work is proportional to the number of nonzero entries in the local rows, and part of it is proportional to the number of local variables (that is, the number of local rows). The provided partitioning should therefore aim at a good load balancing of both these quantities. Besides, the algorithm scalability (with respect to the number of processors) will be best when minimizing the sum of absolute values of offdiagonal entries connecting rows assigned to different processors (or tasks).

3.1 Basic usage and matrix format

To work, the parallel version needs that several instances of the application program have been launched in parallel and have initialized a MPI framework, with a valid communicator. Let `MPI_COMM` be this communicator (by default, it is `MPI_COMM_WORLD`). All instances or *tasks* sharing this MPI communicator must call simultaneously the main driver `AGMGPAR` as follows.

```
call dagmgpar(N,a,ja,ia,f,x,ijob,iprint,nrest,iter,tol     &
              MPI_COMM,listrank,ifirstlistrank)
```

Each task should supply only a portion of the matrix rows in arrays `a`, `ja`, `ia`.
EACH ROW SHOULD BE REFERENCED IN ONE AND ONLY ONE TASK.
`N` is the number of local rows.

LOCAL ORDERING. Local rows (and thus local variables) should have numbers $1, \ldots, \texttt{N}$.
If a global ordering is in use, a conversion routine should be called before AGMGPAR, see the SPARSKIT package [9] (`http://www-users.cs.umn.edu/~saad`) for the handling matrices in CSR format (for instance, extracting block of rows).

NONLOCAL CONNECTIONS. Offdiagonal entries present in the local rows but connecting with nonlocal variables are to be referenced in the usual way; however, the corresponding column indices must be larger than `N`. The matrix supplied to AGMGPAR is thus formally a rectangular matrix with N rows, and an entry A_{ij} (with $1 \leq i \leq$ `N`) corresponds to a local connection if $j \leq$ `N` and to an external connection if $j >$ `N`.

IMPORTANT RESTRICTION. **The global matrix must be structurally symmetric with respect to nonlocal connections**. That is, if A_{ij} corresponds to an external connection, the local row corresponding to j, whatever the task to which it is assigned, should also contain an offdiagonal entry (possibly zero) referencing (an external variable corresponding to) i.

CONSISTENCY OF LOCAL ORDERINGS. Besides the condition that they are larger than `N`, indexes of nonlocal variable may be chosen arbitrarily, providing that their ordering is consistent with the local ordering on their "home" task (the task to which the corresponding row is assigned). That is, if A_{ij} and A_{kl} are both present and such that $j, l >$ `N`, and if further j and l have same home task (as specified in `listrank`, see below), then one should have $j < l$ if and only if, on their home task, the variable corresponding to j has lower index than the variable corresponding to l.

These constraints on the input matrix should not be difficult to meet in practice. Thanks to them, AGMG may set up the parallel solution process with minimal additional information. In fact, AGMG has only to know what is the rank of the "home" task of each referenced non-local variable. This information is supplied in input vector `listrank`.

Let $j_{\min}$ and $j_{\max}$ be, respectively, the smallest and the largest index of a non-local variable referenced in `ja` (`N` $< j_{\min} \leq j_{\max}$). Only entries `listrank`(j) for $j_{\min} \leq j \leq j_{\max}$ will be referenced and need to be defined. If j is effectively present in `ja`, `listrank`(j) should be equal to the rank of the "home" task of (the row corresponding to) j; otherwise, `listrank`(j) should be equal to an arbitrary *nonpositive* integer.

`listrank` is declared in AGMGPAR as `listrank(ifirstlistrank:*)`.
Setting `ifirstlistrank`= $j_{\min}$ or `ifirstlistrank`=`N`+1 allows to save on memory, since `listrank`(i) is never referenced for $i < j_{\min}$ (hence in particular for $i \leq$`N`.

`ijob`: comments in Section 2.2.1 apply, except that the parallel version cannot work directly with the transpose of the input matrix. Hence `ijob`=100,101,102,110,112 are not permitted.

By way of illustration, consider the same matrix as in Section 2.1 partitioned into 3 tasks, task 0 receiving rows 1 & 2, task 1 rows 3 & 4, and task 2 row 5. Firstly, one has to add a few entries into the structure to enforce the structural symmetry with respect to non-local connections:

$$A = \left(\begin{array}{cc|cc|c} 10 & & & -1 & \\ -2 & 11 & & -3 & \\ \hline & -4 & 12 & & -5 \\ & & & 13 & \\ \hline & -8 & -9 & & 14 \end{array}\right) \rightarrow \left(\begin{array}{cc|cc|c} 10 & & & -1 & \\ -2 & 11 & 0 & -3 & 0 \\ \hline & -4 & 12 & & -5 \\ 0 & 0 & & 13 & \\ \hline & -8 & -9 & & 14 \end{array}\right) .$$

Then A is given (non uniquely) by the following variables and vectors.

TASK 0

$$\begin{aligned} \texttt{n} &= 2 \\ \texttt{a} &= [\ 10.0 \quad -1.0 \mid -2.0 \quad 11.0 \quad -3.0 \quad 0.0 \quad 0.0\ ] \\ \texttt{ja} &= [\ 1 \quad 4 \mid 1 \quad 2 \quad 4 \quad 3 \quad 5\ ] \\ \texttt{ia} &= [\ 1 \quad 3 \quad 8\ ] \\ \texttt{listrank} &= [\ * \quad * \quad 1 \quad 1 \quad 2\ ] \end{aligned}$$

TASK 1

$$\begin{aligned} \texttt{n} &= 2 \\ \texttt{a} &= [\ -4.0 \quad 12.0 \quad -5.0 \mid 13.0 \quad 0.0 \quad 0.0\ ] \\ \texttt{ja} &= [\ 7 \quad 1 \quad 5 \mid 2 \quad 6 \quad 7\ ] \\ \texttt{ia} &= [\ 1 \quad 4 \quad 7\ ] \\ \texttt{listrank} &= [\ * \quad * \quad * \quad * \quad 2 \quad 0 \quad 0\ ] \end{aligned}$$

TASK 2

$$\begin{aligned} \texttt{n} &= 1 \\ \texttt{a} &= [\ 14.0 \quad -9.0 \quad -8.0\ ] \\ \texttt{ja} &= [\ 1 \quad 5 \quad 3\ ] \\ \texttt{ia} &= [\ 1 \quad 4\ ] \\ \texttt{listrank} &= [\ * \quad * \quad 0 \quad -1 \quad 1\ ] \end{aligned}$$

Note that these vectors, in particular the numbering of nonlocal variables, were not constructed in a logical way, but rather to illustrate the flexibility and also the requirements of the format. One sees for instance with TASK 2 that the numbering of nonlocal variables may be quite arbitrary as long as "holes" in `listrank` are filled with negative numbers.

3.2 Example

The source file of the following example is provided with the package. The matrix corresponding to the five-point approximation of the Laplacian on the unit square is partitioned according a strip partitioning of the domain, with internal boundaries parallel to the x direction. Note that this strip partitioning is not optimal and has been chosen for the sake of simplicity.

If `IRANK`> 0, nodes at the bottom line have a non-local connection with task `IRANK-1`, and if `IRANK`$<$`NPROC`-1, nodes at the top line have a non-local connection with task `IRANK+1`. Observe that these non-local variables are given indexes $>$`N` in subroutine `uni2dstrip`, and that `listrank` is set up accordingly. Note also that with this simple example the consistency of local orderings arises in a natural way.

Note also that each task will print its output in a different file. This is recommended.

Listing 2: source code of parallel Example

```
      program example_par
!
!  Solves the discrete Laplacian on the unit square by simple call to agmg.
!  The right-hand-side is such that the exact solution is the vector of all 1.
!  Uses a strip partitioning of the domain, with internal boundaries parallel
!       to the x direction.
!
        implicit none
        include 'mpif.h'
        real (kind(0d0)),allocatable :: a(:),f(:),x(:)
        integer,allocatable :: ja(:),ia(:),listrank(:)
        integer :: n,iter,iprint,nhinv,NPROC,IRANK,mx,my,ifirstlistrank,ierr
        real (kind(0d0)) :: tol
        character*10 filename
!
!       set inverse of the mesh size (feel free to change)
        nhinv=1000
!
!       maximal number of iterations
        iter=50
!
!       tolerance on relative residual norm
        tol=1.e-6
!
!  Initialize MPI
!
        call MPI_INIT(ierr)
        call MPI_COMM_SIZE(MPI_COMM_WORLD,NPROC,ierr)
        call MPI_COMM_RANK(MPI_COMM_WORLD,IRANK,ierr)
```

```
!
!       unit number for output messages (alternative: iprint=10+IRANK)
        iprint=10
        filename(1:8)='res.out_'
        write (filename(9:10),'(i2.2)') IRANK        ! processor dependent
        open(iprint,file=filename,form='formatted')
!
!     calculate local grid size
!
        mx=nhinv-1
        my=(nhinv-1)/NPROC
        if (IRANK < mod(nhinv-1,NPROC)) my=my+1
!
!       generate the matrix in required format (CSR)
!
!         first allocate the vectors with correct size
            N=mx*my
            allocate (a(5*N),ja(5*N),ia(N+1),f(N),x(n),listrank(2*mx))
!           external nodes connected with local ones on top and bottom
!           internal boundaries will receive numbers [N+1,...,N+2*mx]
            ifirstlistrank=N+1
!         next call subroutine to set entries
!           before, initialize listrank to zero so that entries
!           that do not correspond to a nonlocal variable present
!           in ja are anyway properly defined
          listrank(1:2*mx)=0
          call uni2dstrip(mx,my,f,a,ja,ia,IRANK,NPROC,listrank,ifirstlistrank)
!
!       call agmg
!         argument 5 (ijob)  is 0 because we want a complete solve
!         argument 7 (nrest) is 1 because we want to use flexible CG
!                            (the matrix is symmetric positive definite)
!
          call dagmgpar(N,a,ja,ia,f,x,0,iprint,1,iter,tol,          &
                        MPI_COMM_WORLD,listrank,ifirstlistrank)
!
!       uncomment the following to write solution on disk for checking
!
!         filename(1:8)='sol.out_'
!         write (filename(9:10),'(i2.2)') IRANK          ! processor dependent
!         open(11,file=filename,form='formatted')
!         write(11,'(e22.15)') x(1:n)
!         close(11)

        END program example_par
```

```
!-----------------------------------------------------------------------
    subroutine uni2dstrip(mx,my,f,a,ja,ia,IRANK,NPROC,listrank,ifirstlistrank)
!
! Fill a matrix in CSR format corresponding to a constant coefficient
! five-point stencil on a rectangular grid
! Bottom boundary is an internal boundary if IRANK > 0, and
!    top boundary is an internal boundary if IRANK < NPROC-1
!
      implicit none
      real (kind(0d0)) :: f(*),a(*)
      integer :: mx,my,ia(*),ja(*),ifirstlistrank,listrank(ifirstlistrank:*)
      integer :: IRANK,NPROC,k,l,i,j
      real (kind(0d0)), parameter :: zero=0.0d0,cx=-1.0d0,cy=-1.0d0, cd=4.0d0
!
      k=0
      l=0
      ia(1)=1
      do i=1,my
        do j=1,mx
          k=k+1
          l=l+1
          a(l)=cd
          ja(l)=k
          f(k)=zero
          if(j < mx) then
             l=l+1
             a(l)=cx
             ja(l)=k+1
           else
             f(k)=f(k)-cx
          end if
          if(i < my) then
             l=l+1
             a(l)=cy
             ja(l)=k+mx
           else if (IRANK == NPROC-1) then
             f(k)=f(k)-cy               !real boundary
           else
             l=l+1                      !internal boundary (top)
             a(l)=cy                    ! these external nodes are given the
             ja(l)=k+mx                 !  numbers [mx*my+1,...,mx*(my+1)]
             listrank(k+mx)=IRANK+1     !Thus listrank(mx*my+1:mx*(my+1))=IRANK+1
          end if
          if(j > 1) then
             l=l+1
```

```
            a(l)=cx
            ja(l)=k-1
          else
            f(k)=f(k)-cx
        end if
        if(i >  1) then
            l=l+1
            a(l)=cy
            ja(l)=k-mx
          else if (IRANK == 0) then
            f(k)=f(k)-cy                !real boundary
          else
            l=l+1                       !internal boundary (bottom)
            a(l)=cy                     ! these external nodes are given the
            ja(l)=k+mx*(my+1)           ! numbers [mx*(my+1)+1,...,mx*(my+2)]
            listrank(k+mx*(my+1))=IRANK-1
                                  !Thus listrank(mx*(my+1)+1:mx*(my+2))=IRANK-1
        end if
        ia(k+1)=l+1
      end do
    end do

    return
  end subroutine uni2dstrip
```

3.3 Printed output

Running the above example with 4 tasks, task 0 produces the following output.

```
  0*ENTERING AGMG ************************************************************

**** Global number of unknowns:      998001
  0*      Number of local rows:      249750
**** Global number of nonzeros:     4986009 (per row:   5.00)
  0*    Nonzeros in local rows:     1247251 (per row:   4.99)

  0*SETUP: Coarsening by multiple pairwise aggregation
****  Rmk: Setup performed assuming the matrix symmetric
****       Quality threshold (BlockD):  8.00 ;  Strong diag. dom. trs: 1.29
****         Maximal number of passes:  3  ; Target coarsening factor: 8.00
****                      Threshold for rows with large pos. offdiag.: 0.45

  0*                      Level:           2
**** Global number of variables:      124563          (reduction ratio: 8.01)
  0*       Number of local rows:       31249          (reduction ratio: 7.99)
****  Global number of nonzeros:      622693 (per row: 5.0; red. ratio: 8.01)
  0*     Nonzeros in local rows:      155996 (per row: 5.0; red. ratio: 8.00)

  0*                      Level:           3
**** Global number of variables:       15904          (reduction ratio: 7.83)
  0*       Number of local rows:        3984          (reduction ratio: 7.84)
****  Global number of nonzeros:       80716 (per row: 5.1; red. ratio: 7.71)
  0*     Nonzeros in local rows:       20135 (per row: 5.1; red. ratio: 7.75)

  0*                      Level:           4
**** Global number of variables:        2034          (reduction ratio: 7.82)
  0*       Number of local rows:         497          (reduction ratio: 8.02)
****  Global number of nonzeros:       10516 (per row: 5.2; red. ratio: 7.68)
  0*     Nonzeros in local rows:        2513 (per row: 5.1; red. ratio: 8.01)
  0*        Exact factorization:       0.123 work units (*)

****           Global grid complexity:     1.14
****       Global Operator complexity:     1.14
  0*            Local grid complexity:     1.14
  0*        Local Operator complexity:     1.14
****  Theoretical Weighted complexity:     1.33 (K-cycle at each level)
****    Effective Weighted complexity:     1.33 (V-cycle enforced where needed)
```

```
  0*         memory used (peak):     2.82 real(8) words per nnz (   26.86 Mb)
  0*       Setup time (Elapsed):     1.27E+00 seconds

  0*SOLUTION: flexible conjugate gradient iterations (FCG(1))
****     AMG preconditioner with 1 Gauss-Seidel pre- and post-smoothing sweeps
****         at each level
****  Iter=    0        Resid= 0.63E+02        Relat. res.= 0.10E+01
****  Iter=    1        Resid= 0.20E+02        Relat. res.= 0.32E+00
****  Iter=    2        Resid= 0.54E+01        Relat. res.= 0.85E-01
****  Iter=    3        Resid= 0.20E+01        Relat. res.= 0.31E-01
****  Iter=    4        Resid= 0.53E+00        Relat. res.= 0.84E-02
****  Iter=    5        Resid= 0.30E+00        Relat. res.= 0.47E-02
****  Iter=    6        Resid= 0.81E-01        Relat. res.= 0.13E-02
****  Iter=    7        Resid= 0.37E-01        Relat. res.= 0.58E-03
****  Iter=    8        Resid= 0.10E-01        Relat. res.= 0.16E-03
****  Iter=    9        Resid= 0.36E-02        Relat. res.= 0.56E-04
****  Iter=   10        Resid= 0.15E-02        Relat. res.= 0.24E-04
****  Iter=   11        Resid= 0.53E-03        Relat. res.= 0.84E-05
****  Iter=   12        Resid= 0.21E-03        Relat. res.= 0.33E-05
****  Iter=   13        Resid= 0.84E-04        Relat. res.= 0.13E-05
****  Iter=   14        Resid= 0.29E-04        Relat. res.= 0.46E-06
****  - Convergence reached in   14 iterations -

****     level 2   #call=    14   #cycle=    28   mean=   2.00    max=  2
****     level 3   #call=    28   #cycle=    56   mean=   2.00    max=  2

  0*       Number of work units:    11.84 per digit of accuracy (*)
  0*   memory used for solution:     2.87 real(8) words per nnz (   27.35 Mb)
  0*    Solution time (Elapsed):     6.54E-01 seconds

  0 (*) 1 work unit represents the cost of 1 (fine grid) residual evaluation
  0*LEAVING AGMG * (MEMORY RELEASED) ****************************************
```

For some items, both a local and a global quantity are now given. All output lines starting with `****` correspond to global information and are printed only by the task with `RANK` 0. Other lines starts with `xxx*`, where "`xxx`" is the task `RANK`. They are printed by each task, based on local computation. For instance, the work for the exact factorization at the coarsest level is the local work as reported by MUMPS, divided by the number of local floating point operations needed for one residual evaluation (to get the local number of work units). On the other hand, the work for the solution phase is the number of local floating point operations, again relative to the local cost of one residual evaluation. If one has about the same number on each task, it means that the initial load balancing (whether good or bad) has been well preserved in AGMG.

3.4 Tuning the parallel version

As already written in Section 2.5, default parameters defined in the module `dagmgpar_mem` at the very top of the source file are easy to change. Perhaps such tuning is more useful in the parallel case than in the sequential one. Here performances not only depend on the application, but also on the computer; e.g., the parameters providing smallest average solution time may depend on the communication speed.

Compared with the sequential version, by default, the number of pairwise aggregation passes `npass` has been increased from 2 to 3. This favors more aggressive coarsening, in general at the price of some increase of the setup time. Hence, on average, this slightly increases the sequential time of roughly 10 %. But this turns out to be often beneficial in parallel, because this reduces the time spent on small grids where communications are relatively more costly and more difficult to overlap with computation. If, however, you experience convergence difficulties, it may be wise to restore the default `npass`=2. On the other hand, you may try to obtain even more aggressive coarsening by increasing also `kaptg_blocdia` and/or `kaptg_dampJac`, after having checked effects on the convergence speed (they are application dependent).

Eventually, in the parallel case, it is worth checking how the solver works on the coarsest grid. Large set up time may indicate that this latter is too large. This can be reduced by decreasing `maxcoarsesize` and `maxcoarsesizeslow`. It is also possible to tune their effect according to the number of processors in subroutine `dagmgpar_init` (just below `agmgpar_mem`). Conversely, increasing the size of the coarsest system may help reduce communications as long as the coarsest solver remains efficient, which depends not only of this size, but also on the number of processors and on the communication speed. *Professional version only: with the variant "NM" that does not use the parallel MUMPS library, the coarsest grid solver is a special iterative solver; parameters that define the target accuracy and the maximal number of iterations at this level can also be tuned.*

4 Running several instances of AGMG

In some contexts, it is interesting to have simultaneously in memory several instances of the preconditioner. For instance, when a preconditioner for a matrix in block form is defined considering separately the different blocks, and when one would like to use AGMG for several of these blocks.

Such usage of AGMG is not foreseen with the present version. However, there is a simple workaround. In the sequential case, copy the source file, say, dagmg.f90 to dagmg1.f90. Then, edit dagmg1.f90, and make a global replacement of all instances of `dagmg` (lowercase) by `dagmg1`, *but not* `DAGMG_MUMSP` (uppercase), which should remain unchanged. Repeat this as many times as needed to generate dagmg2.f90, dagmg3.f90, etc. Then, compile all these source files together with the application program and dagmg_mumps.f90. The application program can call dagmg1(), dagm2(), dagm3(), ..., defining several instances of the preconditioner that will not interact with each other.

You can proceed similarly with any arithmetic and also the parallel version. In the latter case, replace all instances of `dagmgpar` by `dagmgpar1` (no exception).

Matlab: the same result is obtained in an even simpler way. Copy the three files agmg.m, dmtlagmg.mex??? and zmtlagmg.mex??? (where ??? depends upon your OS and architecture) to, say, agmg1.m, dmtlagmg1.mex??? and zmtlagmg1.mex???. Edit the m-file agmg1.m, and replace the 3 calls to `dmtlagmg` by calls to `dmtlagmg1` , and the 3 calls to `zmtlagmg` by calls to `zmtlagmg1` . Repeat this as many times as needed. Then, functions *agmg1*, *agmg2*, ..., are defined which can be called independently of each other.

5 Solving singular systems

Singular systems can be solved directly. Further, this is done automatically, without needing to tell the software that the provided system is singular. However, this usage should be considered with care and a few limitations apply. Basically, it works smoothly when the system is compatible and when the left null space is spanned by the vector of all ones (i.e., the range of the input matrix is the set of vectors orthogonal to the constant vector and the right hand side belongs to this set). Please note that failures in other cases are generally not fatal and hence not detected if the solution is not checked appropriately; for instance, AGMG may seemingly work well but returns a solution vector which is highly dominated by null space components.

To handle such cases, note that AGMG can often be efficiently applied to solve the near singular system obtained by deleting one row and one column in a linear system originally singular (and compatible). This approach is not recommended in general because the truncated system obtained in this way tends to be ill-conditioned and hence no very easy

to solve with iterative methods in general. However, it is sensible to use this approach with AGMG because one of the features of the method in AGMG is precisely its capability to solve efficiently ill-conditioned linear systems.

A Listing of DAGMG

```
SUBROUTINE dagmg( n,a,ja,ia,f,x,ijob,iprint,nrest,iter,tol )
  INTEGER    :: n,ia(n+1),ja(*),ijob,iprint,nrest,iter
  REAL (kind(0.0d0)) :: a(*),f(n),x(n)
  REAL (kind(0.0d0)) :: tol

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
!  Arguments
!  =========
!
!  N         (input) INTEGER.
!            The dimension of the matrix.
!
!  A         (input/output) REAL (kind(0.0e0)). Numerical values of the matrix.
!  IA        (input/output) INTEGER. Pointers for every row.
!  JA        (input/output) INTEGER. Column indices.
!
!              AGMG ASSUMES THAT ALL DIAGONAL ENTRIES ARE POSITIVE
!
!            Detailed description of the matrix format
!
!              On input, IA(I), I=1,...,N, refers to the physical start
!              of row I. That is, the entries of row I are located
!              in A(K), where K=IA(I),...,IA(I+1)-1. JA(K) carries the
!              associated column indices. IA(N+1) must be defined in such
!              a way that the above rule also works for I=N (that is,
!              the last valid entry in arrays A,JA should correspond to
!              index K=IA(N+1)-1). According what is written
!              above, AGMG assumes that some of these JA(K) (for
!              IA(I)<= K < IA(I+1) ) is equal to I with corresponding
!              A(K) carrying the value of the diagonal element,
!              which must be positive.
!
!              A,IA,JA are "output" parameters because on exit the
!              entries of each row may occur in a different order (The
!              matrix is mathematically the same, but stored in
!              different way).
!
!  F         (input/output) REAL (kind(0.0e0)).
!            On input, the right hand side vector f.
!            Overwritten on output.
!            Significant only if IJOB==0, 2, 3, 10, 12, 100, 102, 110, 112
!
```

```
!  X        (input/output) REAL (kind(0.0e0)).
!           On input and if IJOB== 10, 12, 110, 112: initial guess
!              (for other values of IJOB, the default is used: the zero vector).
!           On output, the computed solution.
!
! IJOB      (input) INTEGER. Tells AGMG what has to be done.
!           0: performs setup + solve + memory release, no initial guess
!          10: performs setup + solve + memory release, initial guess in x(1:n)
!           1: performs setup only
!              (preprocessing: prepares all parameters for subsequent solves)
!           2: solves only (based on previous setup), no initial guess
!          12: solves only (based on previous setup), initial guess in x(1:n)
!           3: the vector returned in x(1:n) is not the solution of the linear
!                 system, but the result of the action of the multigrid
!                 preconditioner on the right hand side in f(1:n)
!          -1: erases the setup and releases internal memory
!
!   IJOB == 100,110,101,102,112: same as, respectively, IJOB==0,10,1,2,12
!        but, use the TRANSPOSE of the input matrix in A, JA, IA.
!
!   !!! IJOB==2,3,12,102,112 require that one has previously called AGMG
!        with IJOB==1 or IJOB==101
!
!   !!! (change with respect to versions 2.x) !!!
!        The preconditioner defined when calling AGMG
!          with IJOB==1 or IJOB==101 is entirely kept in internal memory.
!        Hence the arrays A, JA and IA are not accessed upon subsequent calls
!          with IJOB==3.
!        Upon subsequent calls with IJOB==2,12,102,112, a matrix needs to
!             be supplied in arrays A, JA, IA, but it will be used to
!             perform matrix vector product within the main iterative
!             solution process (and only for this).
!             Hence the system is solved with this matrix which
!             may differ from the matrix in A, JA, IA that was supplied
!             upon the previous call with IJOB==1 or IJOB==101;
!             then AGMG attempts to solve a linear system with the "new"
!             matrix (supplied when IJOB==2,12,102 or 112) using the
!             preconditioner set up for the "old" one (supplied when
!             IJOB==1 or 101).
!          The same remarks apply to IJOB >= 100 or not: the value IJOB==1
!             or 101 determines whether the preconditioner set up and stored
!             in internal memory is based on the matrix or its transpose;
!             the value IJOB==2,12 or 102,112 is used to determine whether
!             the linear system to be solved is with the matrix or its
!             transpose, independently of the set up.
```

```
!              Hence one may set up a preconditioner for a matrix and use it
!              for its transpose.
!         These functionalities (set up a preconditioner and use it for another
!              matrix) are provided for the sake of generality but should be
!              used with care; in general, set up is fast with AGMG and hence
!              it is recommended to rerun it even if the matrix changes only
!              slightly.
!
! IPRINT    (input) INTEGER.
!                Indicates the unit number where information is to be printed
!                (N.B.: 5 is converted to 6). If nonpositive, only error
!                messages are printed on standard output.
!
! NREST     (input) INTEGER.
!               Restart parameter for GCR (an implementation of GMRES)
!               Nonpositive values are converted to NREST=10 (default)
!
! !!   If NREST==1, Flexible CG is used instead of GCR (when IJOB==0,10,2,
!               12,100,110,102,112) and also (IJOB==0,1,100,101) performs some
!               simplification during the set up based on the assumption
!               that the matrix supplied in A, JA, IA is symmetric (there is
!               then no more difference between IJOB==1 and IJOB==101).
!
! !!!  NREST=1 Should be used if and only if the matrix is really SYMMETRIC
! !!!           (and positive definite).
!
!  ITER     (input/output) INTEGER.
!           On input, the maximum number of iterations. Should be positive.
!           On output, actual number of iterations.
!             If this number of iteration was insufficient to meet convergence
!             criterion, ITER will be returned negative and equal to the
!             opposite of the number of iterations performed.
!           Significant only if IJOB==0, 2, 10, 12, 100, 102, 110, 112
!
!  TOL      (input) REAL (kind(0.0e0)).
!           The tolerance on residual norm. Iterations are stopped whenever
!                || A*x-f || <= TOL* || f ||
!           Should be positive and less than 1.0
!           Significant only if IJOB==0, 2, 10, 12, 100, 102, 110, 112
!
!!!!! Remark !!!! Except insufficient number of iterations to achieve
!                 convergence (characterized by a negative value returned
!                 in ITER), all other detected errors are fatal and lead
!                 to a STOP statement.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

B Listing of AGMGPAR

```
  SUBROUTINE dagmgpar( n,a,ja,ia,f,x,ijob,iprint,nrest,iter,tol &
                      ,MPI_COMM_AGMG,listrank,ifirstlistrank )
    INTEGER    :: n,ia(n+1),ja(*),ijob,iprint,nrest,iter
    REAL (kind(0.0d0)) :: a(*),f(n),x(n)
    REAL (kind(0.0d0)) :: tol
    INTEGER    :: MPI_COMM_AGMG,ifirstlistrank,listrank(*)
!     Actually: listrank(ifirstlistrank:*)
!                 - listrank is only used in lower level subroutines
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!
!  Arguments
!  =========
!
!  N          (input) INTEGER.
!             The dimension of the matrix.
!
!  A          (input/output) REAL (kind(0.0e0)). Numerical values of the matrix.
!  IA         (input/output) INTEGER. Pointers for every row.
!  JA         (input/output) INTEGER. Column indices.
!
!                 AGMG ASSUMES THAT ALL DIAGONAL ENTRIES ARE POSITIVE
!
!             Detailed description of the matrix format
!
!                 On input, IA(I), I=1,...,N, refers to the physical start
!                 of row I. That is, the entries of row I are located
!                 in A(K), where K=IA(I),...,IA(I+1)-1. JA(K) carries the
!                 associated column indices. IA(N+1) must be defined in such
!                 a way that the above rule also works for I=N (that is,
!                 the last valid entry in arrays A,JA should correspond to
!                 index K=IA(N+1)-1). According what is written
!                 above, AGMG assumes that some of these JA(K) (for
!                 IA(I)<= K < IA(I+1) ) is equal to I with corresponding
!                 A(K) carrying the value of the diagonal element,
!                 which must be positive.
!
!                 A,IA,JA are "output" parameters because on exit the
!                 entries of each row may occur in a different order (The
!                 matrix is mathematically the same, but stored in
!                 different way).
!
!  F          (input/output) REAL (kind(0.0e0)).
!             On input, the right hand side vector f.
```

```
!           Overwritten on output.
!           Significant only if IJOB==0, 2, 3, 10 or 12
!
!  X        (input/output) REAL (kind(0.0e0)).
!           On input and if IJOB==10 or IJOB==12, initial guess
!              (for other values of IJOB, the default is used: the zero vector).
!           On output, the computed solution.
!
! IJOB      (input) INTEGER. Tells AGMG what has to be done.
!           0: performs setup + solve + memory release, no initial guess
!          10: performs setup + solve + memory release, initial guess in x(1:n)
!           1: performs setup only
!              (preprocessing: prepares all parameters for subsequent solves)
!           2: solves only (based on previous setup), no initial guess
!          12: solves only (based on previous setup), initial guess in x(1:n)
!           3: the vector returned in x(1:n) is not the solution of the linear
!                 system, but the result of the action of the multigrid
!                 preconditioner on the right hand side in f(1:n)
!          -1: erases the setup and releases internal memory
!
!   !!! IJOB==2,3,12 require that one has previously called AGMG with IJOB==1
!
!   !!! (change with respect to versions 2.x) !!!
!       The preconditioner defined when calling AGMG
!         with IJOB==1 is entirely kept in internal memory.
!       Hence the arrays A, JA and IA are not accessed upon subsequent calls
!         with IJOB==3.
!       Upon subsequent calls with IJOB==2, 12,  a matrix needs to
!            be supplied in arrays A, JA, IA, but it will be used to
!            perform matrix vector product within the main iterative
!            solution process (and only for this).
!            Hence the system is solved with this matrix which
!            may differ from the matrix in A, JA, IA that was supplied
!            upon the previous call with IJOB==1;
!            then AGMG attempts to solve a linear system with the "new"
!            matrix (supplied when IJOB==2,12) using the preconditioner
!            set up for the "old" one (supplied when IJOB==1).
!       These functionalities (set up a preconditioner and use it for another
!            matrix) are provided for the sake of generality but should be
!            used with care; in general, set up is fast with AGMG and hence
!            it is recommended to rerun it even if the matrix changes only
!            slightly.
!
! IPRINT    (input) INTEGER.
!              Indicates the unit number where information is to be printed
```

```
!              (N.B.: 5 is converted to 6). If nonpositive, only error
!              messages are printed on standard output.
!
! NREST     (input) INTEGER.
!              Restart parameter for GCR (an implementation of GMRES)
!              Nonpositive values are converted to NREST=10 (default)
!
! !!   If NREST==1, Flexible CG is used instead of GCR (when IJOB==0,10,2, 12)
!              and also (IJOB==0,1) performs some simplification during
!              the set up based on the assumption that the matrix
!              supplied in A, JA, IA is symmetric.
!
! !!!  NREST=1 Should be used if and only if the matrix is really SYMMETRIC
! !!!          (and positive definite).
!
!  ITER     (input/output) INTEGER.
!           On input, the maximum number of iterations. Should be positive.
!           On output, actual number of iterations.
!             If this number of iteration was insufficient to meet convergence
!             criterion, ITER will be returned negative and equal to the
!             opposite of the number of iterations performed.
!           Significant only if IJOB==0, 2, 10 or 12
!
!  TOL      (input) REAL (kind(0.0e0)).
!           The tolerance on residual norm. Iterations are stopped whenever
!                || A*x-f || <= TOL* || f ||
!           Should be positive and less than 1.0
!           Significant only if IJOB==0, 2, 10 or 12
!
! MPI_COMM_AGMG (input) INTEGER
!                MPI communicator
!
! listrank(ifirstlistrank:*) INTEGER (input/output)
!            Contains the rank of the tasks to which rows corresponding to
!            nonlocal variable referenced in (A,JA,IA) are assigned.
!            Let Jmin and Jmax be, respectively, the smallest and the largest
!            index of a nonlocal variable referenced in JA(1:IA(N+1)-1)
!            (Note that N < Jmin <= Jmax).
!            listrank(i) will be referenced if and only if  Jmin <= i <= Jmax,
!            and listrank(i) should then be equal to the rank of the "home"
!            task of i if i is effectively present in JA(1:IA(N+1)-1),
!            and equal to some arbitrary NONPOSITIVE integer otherwise.
!
!          listrank and ifirstlistrank may be modified on output, according
!          to the possible modification of the indexes of nonlocal variables;
```

```
!          that is, on output, listrank and ifirstlistrank still carry the
!          correct information about nonlocal variables, but for the
!          matrix as defined in (A,JA,IA) on output.
!
!!!!! Remark !!!! Except insufficient number of iterations to achieve
!                  convergence (characterized by a negative value returned
!                  in ITER), all other detected errors are fatal and lead
!                  to a STOP statement.
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, , and D. Sorensen, *LAPACK Users' Guide, 3rd ed.*, SIAM, 1999.

[2] S. C. Eisenstat, H. C. Elman, and M. H. Schultz, *Variational iterative methods for nonsymmetric systems of linear equations*, SIAM J. Numer. Anal., 20 (1983), pp. 345–357.

[3] G. Karypis, *METIS software and documentation.* Available online at `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[4] *MUMPS software and documentation.* Available online at `http://graal.ens-lyon.fr/MUMPS/`.

[5] A. Napov and Y. Notay, *An algebraic multigrid method with guaranteed convergence rate*, SIAM J. Sci. Comput., 34 (2012), pp. A1079–A1109.

[6] Y. Notay, *An aggregation-based algebraic multigrid method*, Electronic Trans. Numer. Anal., 37 (2010), pp. 123–146.

[7] Y. Notay, *Aggregation-based algebraic multigrid for convection-diffusion equations*, SIAM J. Sci. Comput., 34 (2012), pp. A2288–A2316.

[8] Y. Notay, *Numerical comparison of solvers for linear systems from the discretization of scalar PDEs*, 2012. `http://homepages.ulb.ac.be/~ynotay/AGMG/numcompsolv.pdf`.

[9] Y. Saad, *SPARSKIT: a basic tool kit for sparse matrix computations*, tech. rep., University of Minnesota, Minneapolis, 1994. `http://www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html`.

[10] Y. Saad, *Iterative Methods for Sparse Linear Systems*, SIAM, Philadelphia, PA, 2003. Second ed.

[11] H. A. van der Vorst and C. Vuik, *GMRESR: a family of nested GMRES methods*, Numer. Lin. Alg. Appl., 1 (1994), pp. 369–386.