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 : yvan.notayt@agmg.eu, yvan.notay@ulb.be

version: **April 2024**

Abstract

This manual gives an introduction to the use of AGMG. AGMG is available both as a software library for Fortran or C/C++ programs, and as an Octave/Matlab function; Julia is also supported. AGMG solves systems of linear equations using the aggregation-based algebraic multigrid method described in [7] with further improvements from [5] and [8].

The software is expected to be efficient for large systems arising from the discretization of scalar second order elliptic PDEs. It may, however, be tested on any problem. It is indeed purely algebraic; that is, no information has to be supplied besides the system matrix and the right hand side. Note, however that *all diagonal entries of the system matrix should be nonzero.*

The Octave/Matlab/Julia versions accept real and complex matrices, whereas the C/C++/Fortran library is available for double precision and double complex arithmetic. Several levels of parallelism are provided: multi-threading (multi-core acceleration of sequential programs), MPI-based, or hybrid mode (MPI+multi-threading). See the web site `http://agmg.eu` for instructions to obtain a copy of the software and possible upgrade.

Key words: Multigrid, AMG, Linear Systems, Iterative Methods, Preconditioning, Aggregation, Parallel Computing, Software, Fortran, C, Octave, Matlab, Julia.

[*]Yvan Notay is Research Director of the Fonds de la Recherche Scientifique – FNRS.

Contents

1 Introduction

The AGMG library for Fortran or C/C++ programs is written in Fortran 90. The main driver subroutine is `AGMG` for the sequential & multithread versions, and `AGMGPAR` or `AGMGPARG` for the MPI & hybrid versions. Each driver is available in double precision (prefix `D`) and double complex (prefix `Z`) 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. Additional interfaces `AGMG8`, `AGMGPAR8` and `AGMGPARG8` are also provided to be called from application programs that uses by default long (8 bytes/64 bits) integers.

Octave/Matlab functions are also provided; that is, an Octave oct-file (agmg.oct) and a Matlab m-file (agmg.m) implementing the function *agmg*, which calls AGMG from the Octave/Matlab environment. Library files for the use within Julia environment under GNU/Linux are further supplied. This guide is directed towards the use of the Fortran/C/C++ library, and basic guide on the Octave and Matlab functions is primarily obtained by entering *help agmg* in the Octave/Matlab environment. However, Section 2.1.3 on input arguments and Section 2.3 on output arguments and error flags provide some additional details that might interest Octave or Matlab users. These latter should also refer to Section 2.5 for the description of the verbose output, and to Section 7 for solving singular systems.

For a sample of performance in sequential and comparison with other solvers, see the paper [6] and the report [9] (`http://agmg.eu/numcompsolv.pdf`). For the performance of the parallel version, see the paper [10].

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 [7, 5, 8]. On the other hand, this guide is oriented towards the *use* of AGMG. Further instructions on how to install the package and run the examples are given in the accompanying README file.

1.2 Release information

This guide describes AGMG 4.2.x-aca (academic version) and AGMG 4.2.x-pro (professional version). Several new features are introduced from releases AGMG 3.x.y, see the "ChangeLog" section of the "Software" page of the website `http://agmg.eu`.

Backward compatibility is guaranteed with usage based on release not anterior to 3.0.0, **except** that for versions 4.1.1 and higher `iter` is not anymore an output argument: the maximal number of iterations provided on input is unchanged on output, while the number of performed iterations may be accessed in other ways, see Sections 2.3.2 and 2.3.3.

1.3 Installation and external libraries

AGMG does not need to be installed as a library. For the sake of simplicity, source or object files are provided, which need just to be compiled or linked 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 some subset of MUMPS (`http://graal.ens-lyon.fr/MUMPS/`). For convenience, we provide the needed (public domain) source files (one per arithmetic) together with AGMG (see the files header for a detailed copyright notice). Hence, MUMPS does not need to be installed as a library. To avoid mismatch with a standard implementation of the library, all MUMPS routines provided with AGMG have been renamed. (The prefixes `d` , `z` , have been exchanged for `dagmg_` , `zagmg_`.)

AGMG and provided MUMPS source files have .F90 extension. This means that, when compiling them, preprocessor has to be invoked prior the Fortran compilation. With most compiler, this is automatic when the source file has such extension, but this is not the case, e.g., with old releases of Intel compilers. In such cases, one should make sure that the preprocessor is invoked by using the appropriate option. Otherwise, one will get at least warning messages when compiling the source files, and most often fatal error messages; and if errors are not fatal, the correct execution of the software is not guaranteed.

The preprocessor invocation allows one to switch between code versions by defining macros. For both AGMG and MUMPS source files, the possible macros are "_EXTERNALBL_" (associated option: "-D_EXTERNALBL_")[1] and "_EXTERNALBL_INT8_" (associated option: "-D_EXTERNALBL_INT8_"). Both replace some internal vector and matrix-vector operations by calls to external BLAS or LAPACK routines, and require thus linking with a library that provides them. If "_EXTERNALBL_" is defined the standard BLAS/LAPACK interface is used, whereas if "_EXTERNALBL_INT8_" is defined one should link with versions of BLAS and LAPACK that require long (8 bytes/64 bits) integer as input/output arguments. This can result in improved performance *providing that* a right version of BLAS/LAPACK is selected. For instance, linking with multithread versions of BLAS/LAPACK (which in general is done by default) often results in degraded performance because the computation for which AGMG calls these libraries are not intensive enough, hence the overhead associated with the parallelism is not compensated. This is a fortiori true for the parallel versions of AGMG, since all AGMG threads often call simultaneously these libraries, hence there is no room for additional parallelism. Note that using or not external BLAS and LAPACK may be decided independently for AGMG and MUMPS source files (e.g., based on performance tests).

A specific macro should also be defined when compiling the MUMPS source file for complex arithmetic if one intends to use in a same program both the real and complex versions of AGMG, see the last item in Section 9.

[1]The "-" sign in front of the option corresponds to Linux and macos ways of passing options. On Windows, "/" often replaces "-", hence the in this specific case the correct option is "/D_EXTERNALBL_".

1.4 Error handling

Except for the MPI versions, detected fatal errors, including failure in memory allocation, cause returning to the calling program with the variable `status` set to a positive number, while the memory specifically allocated for this aborted call is cleaned. On the other hand, `status` is set to zero if everything has gone right, whereas it is set to negative if AGMG executes correctly but the required tolerance was not reached within the prescribed number of iterations. The `status` variable may be checked after return from AGMG by calling an auxiliary function, or checking the first entry in array argument `f`, see Sections 2.3.1 and 2.3.3 for details.

In case of crash of AGMG, please check carefully of all input arguments. If errors persist, feel free to contact support@agmg.eu for further assistance.

2 Sequential version

2.1 Calling AGMG

For real input matrices, the application program has to call the main driver `AGMG` as follows (see Section 9 for complex input matrices).

Fortran Syntax:

```
call dagmg(n,a,ja,ia,f,x,ijob,iprint,nrest,maxit,tol)
```

C/C++ Syntax[2]:

```
dagmg_(&n,a,ja,ia,f,x,&ijob,&iprint,&nrest,&maxit,&tol);
```

or

```
DAGMG(&n,a,ja,ia,f,x,&ijob,&iprint,&nrest,&maxit,&tol);
```

Arguments:

`n`:	INPUT	integer
`a`:	INPUT	double precision (array/pointer)
`ja`:	INPUT	integer (array/pointer)
`ia`:	INPUT	integer (array/pointer)
`f`:	IN/OUT	double precision (array/pointer)
`x`:	IN/OUT	double precision (array/pointer)
`ijob`:	INPUT	integer
`iprint`:	INPUT	integer
`nrest`:	INPUT	integer
`maxit`:	INPUT	integer
`tol`:	INPUT	double precision

`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 2.1.3.1 below. `f` is also used as work space and thus modified on output, where its leading entries are further filled with information from the solution process, see Section 2.3.3.

How to specify the input matrix in arrays `a`, `ja` and `ia` is described in Section 2.1.2, whereas the input arguments `iprint`, `nrest`, `maxit` and `tol` are described in Section 2.1.3.

[2]AGMG is written in Fortran 90, hence the name to be used when calling from C/C++ depends on conventions that are compiler and OS dependent. Using `dagmg_` (first option, i.e., adding an underscore to the Fortran name and staying with lowercase) works with GNU and Intel compilers on Linux and macos. Using `DAGMG` (second option, i.e., using capital letters without added underscore) works with Intel compilers on Windows. Pay also attention that additional libraries may be needed at link stage. For instance, to properly link when using GNU compilers on Linux: either link with *gfortran* even if the main is in C, or link with *gcc/g++*, but reference *gfortran* and *m* libraries through the option "-lgfortran -lm"; to properly link with Intel compilers on Linux: link with *ifort* using the option "-nofor-main".

Long (8 bytes/64 bits) integer as input argument.
An alternative driver is provided for the case where *all* the integer arguments are or type long (8 bytes/64 bits) in the calling program. The calling is as follows.

Fortran Syntax:

```
call dagmg8(n,a,ja,ia,f,x,ijob,iprint,nrest,maxit,tol)
```

C/C++ Syntax:

```
dagmg8_(&n,a,ja,ia,f,x,&ijob,&iprint,&nrest,&maxit,&tol);
```

or

```
DAGMG8(&n,a,ja,ia,f,x,&ijob,&iprint,&nrest,&maxit,&tol);
```

2.1.1 Changing internal parameters

Some internal integer parameters can be set to non-default values by calling the following auxiliary function.

Fortran Syntax:

```
call dagmg_intparam("key",value,instance)
```

C/C++ Syntax[3]:

```
dagmg_intparamC("key",value,instance);
```

Arguments:

`"key"`:	INPUT	string
`value`:	INPUT	integer
`instance`:	INPUT	integer

After a call with first argument set to `"key"`, the internal parameter with name `key` is set to `value` for all subsequent calls to the main AGMG driver; `instance` is ignored by default. It is only meaningful in the special *several instances mode* described in Section 6. (Outside this mode users are suggested to use the default value 0.)

The list of adjustable internal parameters is as follows.

Parameters for versatile input matrix format (see Section 2.1.2)

`COOformat`	(default:0) , `nnzinput` (default:0)
`CSCformat`	(default:0)
`ZeroBaseIndex`	(default:0)
`SYMstorage`	(default:0)

Parameters for handling multiple right hand side (see Section 2.2)

`NRHS` (default:1) , `MajOrd` (default:0)

[3]The name is here `dagmg_intparamC` for all compilers and OS because this function is specific to the C/C++ version and supplied in a separated C source file (dagmg_intparamC.c).

Parameter for controlling multithreading (see Section 3)
`nthreads` (default:0)

Parameter for controlling the several instances mode (see Section 6)
`nthreads` (default:0)

Parameter for controlling the handling of singular systems (see Section 7)
`MaxCoaKernVect` (default:1)

Parameters for tuning by expert users: see Section 8

Note that in the above list we give the name of the internal parameter, but the argument of the auxiliary function should be a string containing the name. Further, this is case sensitive. For instance, to set the `COOformat` parameter to 1, the call should be as follows.

Fortran Syntax:

```
call dagmg_intparam("COOformat",1,0)
```

C/C++ Syntax:

```
dagmg_intparamC("COOformat",1,0);
```

Using, e.g., `"cooformat"` or `"COOFORMAT"` will have no effect besides the printing of a warning message.

In the remainder of this guide, we exclusively refer to internal parameters by their name, and one should keep in mind that this name must be surrounded by quotes when calling the auxiliary function to change the parameter value.

Long (8 bytes/64 bits) integer as input argument.
If the input integers arguments `value` and `instance` are of type long (8 bytes/64 bits), Fortran users should use the alternative function:

```
call dagmg8_intparam("key",value,instance)
```

No alternative function is provided for C/C++ users, but they can easily edit the declaration in the function `dagmg_intparamC` in the provided source file so as to match the type of integer used when calling the function (other declarations should not be modified).

2.1.2 Matrix format

The default matrix format is the "Compressed Sparse Row" (CSR) format described, e.g., in [12]. However, by setting some internal parameter(s) to non default values, most popular sparse matrix formats can be used without change in the calling program.

Before describing the formats, note that they all specify the input matrix as a list of nonzero elements; they only differ by the way the list is organized. Then, when two elements in the list correspond to the same pair of row and column indices, AGMG works automatically in *assembly mode*. That is, the final entry for any pair of row and column is the sum of all corresponding entries appearing in the list. In practice, it means that if forming the system matrix requires an assembly process, much of this can be done inside AGMG.

Default: *CSR (Compressed Sparse Row) format*
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` and `ja` must correspond to index $k =$ `ia`(`n`+1)-1.
Remark 1: (for C users) the above description matches Fortran convention; in C/C++ programs, `ia`(i) is accessed as `ia`[i-1] and `ia`(`n`+1) as `ia`[`n`]; see further below for using AGMG with 0-based indexing according to C/C++ conventions.
Remark 2: the format is non-unique; entries in each row may appear in any order.

If `COOformat` > 0: *COO (Coordinate) format*
With this format, nonzero matrix entries (numerical values) are stored in `a`, whereas `ja` and `ia` carry the corresponding column and row indices, respectively. The number of nonzero entries to process has to be specified before calling the main driver via the parameter `nnzinput` (the threes arrays `a`, `ja` and `ia` must have length at least `nnzinput`).
Remark 1: AGMG returns to the main program with a fatal error message if `COOformat` > 0 while `nnzinput` has not been set to a positive integer.
Remark 2: the format is non-unique; entries may appear in any order.

If `CSCformat` > 0 and `COOformat` ≤ 0: *CSC (Compressed Sparse Column) format*
Here, entries are stored column-wise in `a`, whereas `ja` carries the corresponding row indices; entries in `ia` indicate then where every column starts. That is, the format works like CSR with row and column indices exchanging their role. As with the CSR format, `ia` must have length (at least) `n`+1, and `ia`(`n`+1) must be defined in such a way that the last valid entry in arrays `a` and `ja` corresponds to index $k =$ `ia`(`n`+1)-1.
Remark 1: if `CSCformat` and `COOformat` are simultaneously positive (which is not recommended), COO format is used (hence CSCformat is *not* activated), but the input matrix is transposed; i.e., the role of the arrays `ia` and `ja` are interchanged: `ia` contains the column indices and `ja` the row indices; the rationale behind this is that the CSC representation of a matrix corresponds to the CSR representation of its transpose: from a programming perspective, shifting from CSR to CSC amounts to matrix transposition.
Remark 2: the format is non-unique; entries in each column may appear in any order.

If `ZeroBaseIndex` > 0: *Use 0-based indexing* (for whatever format)
By default, AGMG uses 1-based indexing; that is, indexing of arrays start at 1 and row & column indices range from 1 to `n`. This corresponds to Fortran conventions. By setting `ZeroBaseIndex` to positive, 0-based indexing is used instead, according to C/C++ conventions. That is, indices stored in `ja` range from 0 to `n`-1 (all formats), as well as the row indices stored in `ia` when using the COO format (`COOformat` > 0). Further, when using the CSR or CSC format (`COOformat` ≤ 0), `ia` is a 0-based pointer; that is, the first entry in `ia` has to set to be set to 0 (instead of 1 if `ZeroBaseIndex` ≤ 0) to tell AGMG that the first valid matrix entry in `a` and `ja` is the first one in these arrays.

If `SYMstorage` > 0: *The input matrix uses symmetric storage*
Whatever the matrix format (CSR, CSC or COO) and the type of indexing, symmetric matrices may be input using symmetric storage. This is specified by setting `SYMstorage` to positive. Then, any offdiagonal entry a_{ij} in the input list associated with `a`, `ja` and `ia` is replicated in position (j,i), ensuring that $a_{ji} = a_{ij}$ while avoiding to store both.
According to the remark above about the assembly mode, it means that if, for some i, j, $i \neq j$, $\overline{a}_{ij}$ and $\overline{a}_{ji}$ are both present in the input list, the final entry for the indices i, j will be $a_{ji} = a_{ij} = \overline{a}_{ij} + \overline{a}_{ji}$.

Example

By way of illustration, consider the matrix

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

Examples of valid representations are given below for each format.

CSR format:

$$\begin{array}{rcl} \texttt{a} & = & \left[\; 10.0 \;\; -2.0 \;\middle|\; -1.0 \;\; 11.0 \;\; -4.0 \;\; -3.0 \;\middle|\; 12.0 \;\middle|\; 13.0 \;\; -5.0 \;\right] \\ \texttt{ja} & = & \left[\; 1 \;\; 2 \;\middle|\; 1 \;\; 2 \;\; 3 \;\; 4 \;\middle|\; 3 \;\middle|\; 4 \;\; 1 \;\right] \\ \texttt{ia} & = & \left[\; 1 \;\; 3 \;\; 7 \;\; 8 \;\; 10 \;\right] \end{array}$$

(Observe that columns indices in the last row are unsorted)

CSR format with `ZeroBaseIndex` > 0:

$$\begin{array}{rcl} \texttt{a} & = & \left[\; 10.0 \;\; -2.0 \;\middle|\; -1.0 \;\; 11.0 \;\; -4.0 \;\; -3.0 \;\middle|\; 12.0 \;\middle|\; 13.0 \;\; -5.0 \;\right] \\ \texttt{ja} & = & \left[\; 0 \;\; 1 \;\middle|\; 0 \;\; 1 \;\; 2 \;\; 3 \;\middle|\; 2 \;\middle|\; 3 \;\; 0 \;\right] \\ \texttt{ia} & = & \left[\; 0 \;\; 2 \;\; 6 \;\; 7 \;\; 9 \;\right] \end{array}$$

CSC format:

$$\begin{array}{rcl} \texttt{a} & = & \left[\; 10.0 \;\; -1.0 \;\; -5.0 \;\middle|\; -6.0 \;\; 11.0 \;\; 4.0 \;\middle|\; -4.0 \;\; 12.0 \;\middle|\; 13.0 \;\; -3.0 \;\right] \\ \texttt{ja} & = & \left[\; 1 \;\; 2 \;\; 4 \;\middle|\; 1 \;\; 2 \;\; 1 \;\middle|\; 2 \;\; 3 \;\middle|\; 4 \;\; 2 \;\right] \\ \texttt{ia} & = & \left[\; 1 \;\; 4 \;\; 7 \;\; 9 \;\; 11 \;\right] \end{array}$$

(Observe that the entry in position $(1,2)$ illustrates the assembly mode.)

COO format:

$$\begin{aligned}\texttt{a} &= \begin{bmatrix} 4.0 & -1.0 & -5.0 & -6.0 & 11.0 & 10.0 & -4.0 & 13.0 & 12.0 & -3.0 \end{bmatrix}\\ \texttt{ja} &= \begin{bmatrix} 2 & 1 & 1 & 2 & 2 & 1 & 3 & 4 & 3 & 4 \end{bmatrix}\\ \texttt{ia} &= \begin{bmatrix} 1 & 2 & 4 & 1 & 2 & 1 & 2 & 4 & 3 & 2 \end{bmatrix}\end{aligned}$$

(Observe that the entry in position $(1, 2)$ illustrates the assembly mode.)

Finally, for any combination of input arrays and parameters illustrated above, if `SYMstorage` is simultaneously set to positive, the input system matrix will be:

$$A_S = \begin{pmatrix} 10 & -3 & & -5 \\ -3 & 11 & -4 & -3 \\ & -4 & 12 & \\ -5 & -3 & & 13 \end{pmatrix} .$$

(Observe that $A_S = A + A^T - \text{diag(A)}$)

2.1.3 Other input arguments: `ijob` , `iprint` , `nrest` , `maxit` and `tol`

2.1.3.1 `ijob`: INPUT, integer

`ijob` tells AGMG what has to be done. If `ijob`=0 , AGMG will solve the specified linear system in the usual way; this implies a setup phase followed by an iterative solution phase [7]. Other values tell AGMG that an initial approximation is given in `x`, and/or to separate setup and solve phase, allowing one to reuse the setup for several solves. It is also possible to call AGMG for just one application of the multigrid preconditioner (to be exploited in a more complex fashion by the calling program).

The valid values of `ijob` are listed in the Table 1.

Remarks

- If `ijob` is set to an non valid value AGMG returns to the main program with a fatal error message.
- `ijob`=2,3,12,102,112,202,212 require that one has previously called AGMG with `ijob`=1 or `ijob`=101; otherwise, AGMG returns to the main program with a fatal error message.
- Standard usage of AGMG is when the system matrix coincides with the matrix used for setup (thus which served to define the multigrid preconditioner stored in internal memory). Then, relevant values of `ijob` are `ijob`=0 or 10 for solving a single linear system, or `ijob`=1 followed by several calls with `ijob`=202 or 212 when several systems with the same matrix need to be solved successively.

  With `ijob`=2 or 12 (or 102/112), AGMG allows one to change the system matrix while keeping the previously built preconditioner. This option is provided for the

`ijob`	Usage
0	Performs setup + solve + memory release, no initial guess.
10	Performs setup + solve + memory release, initial guess in `x`.
1	Performs only the setup (`f` and `x` are not accessed). (Preprocessing: prepares all parameters for subsequent solves.)
202	Performs a solve for the system matrix provided during the setup, using the multigrid preconditioner built during setup; no initial guess. (`a`, `ja` and `ia` are not accessed.)
212	...same as `ijob`=202 but an initial guess is provided 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(s) in `f`. (`a`, `ja` and `ia` are not accessed.)
2	Performs a solve for the system matrix provided in `a`, `ja` and `ia`, using the multigrid preconditioner built during setup; no initial guess. (The system matrix may differ from the one provided for the setup.)
12	...same as `ijob`=2 but an initial guess is provided in `x`.
-1 or 99	Erases the setup and releases internal memory.
100,110 101,102,112 $\}$	same as, respectively, 0,10,1,2,12, but uses the transpose of the input matrix provided in `a`, `ja` and `ia`. (Not available for MPI and hybrid versions.)

Table 1: Possible values for `ijob`.

sake of generality and should be used with care. One should not use it if the system matrix has not changed, as it is less time and memory consuming to call AGMG with `ijob`=202 or 212. Observe that then `a`, `ja` and `ia` are not accessed and the corresponding memory may thus be released in the calling program.

- Using `ijob`=100,110,101,102,112, AGMG will work with the transpose of the input matrix. As noted in the previous section, the same effect is obtained by setting `CSCformat` to positive, since shifting from CSR format to CSC format amounts to transposition. When using simultaneously `ijob`=100,110,101,102 or 112 and `CSCformat` > 0, it further turns out that both specifications cancel each other.

2.1.3.2 `iprint`: INPUT, integer

`iprint` is 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.

Warning messages about insufficient convergence (in the prescribed maximum number of iterations) are further suppressed when supplying a negative number; then AGMG works silently. This is useful, e.g., if one intends to perform solves with a fixed number of iterations without caring about the achieved residual reduction.

2.1.3.3 `nrest`: INPUT, integer

`nrest` is the restart parameter for GCR [2, 13] (an alternative implementation of GMRES [12]), which is the default main iteration routine [7]. 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, 202,212) and also (`ijob`=0,1,100,101) some simplifications are performed during the setup based on the assumption that the input matrix is symmetric. Should be used only when the matrix is symmetric and positive definite.

2.1.3.4 `maxit`: INPUT, integer

`maxit` specifies the maximal number of iterations.

2.1.3.5 `tol`: INPUT, double precision

`tol` specifies the tolerance on the relative residual norm, used as stopping test. Iterations are pursued (within the limit prescribed by `maxit`) until the residual norm is below `tol` times the norm of the input right hand side. For a single right hand side, this means $\|\mathbf{f} - A\,\mathbf{x}\| \leq \texttt{tol} \cdot \|\mathbf{f}\|$, where $\mathbf{f}$ is the right hand side vector stored on input in array `f`, where $\mathbf{x}$ is the vector stored on output in array `x`, while A is the system matrix in arrays `a`, `ja` and `ia` (input at setup or solve time, according to the value of `ijob`).

See the next section for the precise stopping test in case of several right hand sides.

Note that, as with any linear system solver (including direct solvers), the accuracy that can be reached is limited by rounding errors. Indeed, roughly speaking, the residual norm cannot be decreased much beyond $\mathbf{u}\,\|A\|\,\|\mathbf{x}\|$, where $\mathbf{u}$ is the unit roundoff of the machine. If the input argument `tol` requests going beyond this limit, in most cases AGMG will continue iterations until the criterion is seemingly satisfied; however, accuracy has been lost when computing the residual and the residual norm reported by AGMG is then not trustable. Occasionally, AGMG may detect that this maximal accuracy has been reached and stops iterations regardless the specilfied tolerance. When this happens, the apparent residual norm may, however, be already significantly lower than the true one, hence the value reported by AGMG is then not trustable as well.

This potential discrepancy between computed and true residual norms is nevertheless harmless because it takes place only when the linear system(s) has/have been solved with maximal accuracy (as may be checked by comparing the residual norms with those obtained using known backward stable direct solvers). Users are, of course, advised to select only values of the `tol` parameters that are not excessively small.

2.2 Solving at once for multiple right hand sides

AGMG may solve at once for several right hand sides. This feature is not only provided for convenience, but also for performance, as this may be significantly faster than performing successive solves with `ijob`=202 or 212. Indeed, while the number of arithmetic operations will be about the same or slightly larger when solving at once for several right hand sides compared with successive solves, the memory usage is significantly improved, entailing faster execution of the code.

To activate this option, the auxiliary function has to be called to set the parameter `NRHS` to the number of right hand sides in `f`. It is also mandatory to set `MajOrd` to either positive or negative. (If it is zero while $\texttt{NRHS} > 1$, AGMG returns to the calling program with an error message.) Mathematically, the right hand side argument is then a rectangular matrix which contains the collection of right hand sides, one per column, and `MajOrd` tells if the calling program uses major row or major column ordering. AGMG returns the solution in `x` using the same ordering.

$\texttt{MajOrd} > 0$ indicates that *row major ordering* is used (ordering for multidimensional arrays in C). That is, if $(\mathbf{f}_k)_j$ is the *jth* entry of the *kth* right hand side, entries in `f` appear in order $(\mathbf{f}_1)_1$, $(\mathbf{f}_2)_1$, ... , $(\mathbf{f}_{\texttt{NRHS}})_1$, $(\mathbf{f}_1)_2$, ...

$\texttt{MajOrd} < 0$ indicates that *column major ordering* is used (ordering for multidimensional arrays in Fortran). That is, if $(\mathbf{f}_k)_j$ is the *jth* entry of the *kth* right hand side, entries in `f` appear in order $(\mathbf{f}_1)_1$, $(\mathbf{f}_1)_2$, ... , $(\mathbf{f}_1)_{\texttt{n}}$, $(\mathbf{f}_2)_1$, ...

When $\texttt{NRHS} > 1$, the used stopping test is: $\sqrt{\sum_{k=1}^{\texttt{NRHS}} \|\mathbf{f}_k - A\,\mathbf{x}_k\|^2/\|\mathbf{f}_k\|^2} \leq \texttt{tol}$, where $\mathbf{f}_k$ is the *kth* right hand side and $\mathbf{x}_k$ the corresponding solution returned in array `x`. Observe that the tolerance criterion is then satisfied individually for each pair of right hand side and solution vector (at the price of possibly slightly oversolving some of the systems).

2.3 Output arguments and error flags

Two integer output arguments, `status` and `iter`, can be accessed via specific auxiliary functions. Further, if `ijob`=0,10,2,3,12,202 or 212 (i.e., when the right hand side argument `f` is meaningful), some informative output variables are provided in the first entries of `f`, including `status` and `iter` converted to real values (for user convenience, as the call to auxiliary functions is then no more needed).

2.3.1 `status`

This integer variable tells the status of AGMG, containing a possible error flag. A zero value indicates that the last call to AGMG completes successfully. The variable is accessed as follows.

Fortran Syntax:

```
call dagmg_status(status,instance)
```

C/C++ Syntax:

```
dagmg_status_(&status,&instance);
```

or

```
DAGMG_STATUS(&status,&instance);
```

Both arguments are of INTEGER type and `status` is an OUTPUT argument that the function sets to the corresponding internal variable. As for `dagmg_intparam`, `instance` is ignored by default and is only meaningful in the special *several instances mode* described in Section 6.

If long (8 bytes/64 bits) integers are used in the calling program, one should use instead:

Fortran Syntax:

```
call dagmg8_status(status,instance)
```

C/C++ Syntax:

```
dagmg8_status_(&status,&instance);
```

or

```
DAGMG8_STATUS(&status,&instance);
```

Possible values of `status` are listed below. Observe that the value is positive if and only if a fatal error occurred.

`status`	Meaning
0	Normal termination – no error
-1	Last AGMG call executed correctly but the stopping test was not satisfied within the prescribed maximal number of iterations (the prescribed tolerance has not been reached).
-2	Last AGMG call executed correctly but the stopping test was not satisfied because AGMG detected that the maximum accuracy has been reached while the prescribed tolerance is excessively small (see Section 2.1.3.5).
1	Illegal value of `ijob`.
2	`ijob`=2,3,12,102,112,202 or 202 while no setup has been done or kept in memory.
3	COO format requested but the information on the number of nonzero entries has not been provided (`COOformat` > 0 and `nnzinput` ≤ 0).
10	At least one diagonal entry of the input matrix is zero.
11	Failed allocation: not enough memory for setup.
12	Failed allocation: not enough memory for solve.

2.3.2 `iter`

This integer variable is set to the number of iterations performed during the last call to AGMG. It is set to 0 if no iteration was performed because of the value of `ijob` (no iterative solve is performed if `ijob`=-1,1,3 or 101). The variable is accessed as follows.

Fortran Syntax:

```
call dagmg_iter(iter,instance)
```

C/C++ Syntax:

```
dagmg_iter_(&iter,&instance);
```

or

```
DAGMG_ITER(&iter,&instance);
```

Both arguments are of INTEGER type and `iter` is an OUTPUT argument that the function set to the corresponding internal variable. As for `dagmg_intparam`, `instance` is ignored by default and is only meaningful in the special *several instances mode* described in Section 6.

If long (8 bytes/64 bits) integers are used in the calling program, one should use instead:

Fortran Syntax:

```
call dagmg8_iter(iter,instance)
```

C/C++ Syntax:

```
dagmg8_iter_(&iter,&instance);
```

or

```
DAGMG8_ITER(&iter,&instance);
```

2.3.3 Real output arguments

If `ijob`=0,2,3,10,12,102,112,202 or 212, that is, if `f` is a meaningful argument, its first entry on output is set to `status` (converted to real value), which is then accessible without call an auxiliary function. Further, if `ijob`=0,2,10,12,102,112,202 or 212 (that is, if an iterative solve has to be performed) and if `satus` ≤ 0 (that is, if no fatal error occurred), the next entries in `f` are filled with additional information as indicated in the table below.

In this table, we use the following conventions: A is the system matrix in arrays `a`, `ja` and `ia` (input at setup or solve time, according to the value of `ijob`), and, in case of one right hand side, $\mathbf{f}$ is the right hand side vector stored on input in array `f` and $\mathbf{x}$ is the vector stored on output in array `x`, while $\mathbf{x}^{(j)}$ stands for the obtained approximation after j iterations. (Thus, $\mathbf{x}^{(j)} = \mathbf{x}$ for j =`iter` whereas $\mathbf{x}^{(0)}$ is the initial approximation (that is, the all zero vector if `ijob`=0,2 or 202, and the vector stored on input in array `x` if `ijob`=10,12 or 212.) For more than one right hand side, $\mathbf{f}_k$ is the kth right hand side, $\mathbf{x}_k$ the corresponding solution returned in array `x`, and $\mathbf{x}_k^{(j)}$ the related obtained approximation after j iterations.

Fortran	C/C++	Contents if `NRHS` $=1$	Contents if `NRHS` >1
`f`(1)	`f`[0]	Real(`status`)	Real(`status`)
`f`(2)	`f`[1]	Real(`iter`)	Real(`iter`)
`f`(3)	`f`[2]	$\Vert\mathbf{f} - A\,\mathbf{x}\Vert/\Vert\mathbf{f}\Vert$	$\sqrt{\left(\sum_{k=1}^{\mathtt{NRHS}} \Vert\mathbf{f}_k - A\,\mathbf{x}_k\Vert^2/\Vert\mathbf{f}_k\Vert^2\right)/\mathtt{NRHS}}$
For $j = 0, \ldots,$`iter`:			
`f`(4+j)	`f`[3+j]	$\Vert\mathbf{f} - A\,\mathbf{x}^{(\mathbf{j})}\Vert$	$\sqrt{\left(\sum_{k=1}^{\mathtt{NRHS}} \Vert\mathbf{f}_k - A\,\mathbf{x}_k^{(j)}\Vert^2/\Vert\mathbf{f}_k\Vert^2\right)/\mathtt{NRHS}}$

Remarks

- Entries from the fourth one in `f` give thus the convergence history. Observe that (absolute) residual norms are returned in case of one right hand side, whereas the mean of relative residual norms is returned when `NRHS` >1.
- AGMG assumes that `f` has size `n`×`NRHS`. Hence only the first `n`×`NRHS` entries of `f` can be output arguments. Therefore, in the (unlikely) event where `n`×`NRHS`<(`iter`+4), the information returned in `f` is truncated to the `n`×`NRHS` available entries.
- Inside AGMG, `f` is treated as a one dimensional array regardless the number of right hand sides, and the returned output values are set accordingly. This should be taken into account to properly access returned values if `f` is declared as a multi-dimensional array in the calling program.

2.4 Example

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

Listing 1: source code of sequential Example (Fortran 90)

```
      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,maxit,iprint,nhinv,i
        real (kind(0d0)) :: tol
!       set inverse of the mesh size (feel free to change)
        nhinv=500
!       maximal number of iterations
        maxit=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,maxit,tol)
!       display on screen the output info returned by AGMG in f()
        print '()'
        print '("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣AGMG␣status␣:",i5)', int(f(1))
        print '("Number␣of␣performed␣iterations␣:",i5)', int(f(2))
        print '("␣␣␣␣␣␣␣␣Relative␣residual␣norm␣:",1pe9.2)', f(3)
        print '("␣␣␣␣␣␣␣␣␣␣␣Convergence␣history␣:␣#iter␣␣␣Residual␣Norm")'
        print '("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣",i5,1pe16.5)' &
             ,(i,f(4+i),i=0,int(f(2)))
      end program example_seq
```

The same example is also provided in C language. Observe that here zero based indexing is used, and AGMG is informed about that via a call to the auxiliary function. This example also illustrates the use of symmetric storage.
Note the `#include <dagmg_intparamC.c>` needed for calling the auxiliary function. (Hence the directory where this source file is located should be in the Include PATH.)

Listing 2: source code of sequential Example (C)

```
      #include <stdlib.h>
      #include <stdio.h>
      #include <dagmg_intparamC.c>
      void dagmg_(int*,double*,int*,int*,double*,double*
                 ,int*,int*,int*,int*,double*);
      void uni2d(int,double*,double*,int*,int*);
      int main(void) {
/*
   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.
*/
        double *a,*f,*x;
        int n,*ja,*ia,i;
        int zero=0,one=1;

/*      set inverse of the mesh size (feel free to change) */
        int nhinv=500;
/*      maximal number of iterations */
        int maxit=50;
/*      tolerance on relative residual norm */
        double tol=1.e-6;
/*      unit number for output messages: 6 => standard output */
        int iprint=6;
/*      generate the matrix in CSR format using symmetric storage */
/*        first allocate the vectors with correct size */
          n=(nhinv-1)*(nhinv-1);
          ia=malloc((n + 1) * sizeof(int));
          ja=malloc(3*n * sizeof(int));     /* set 5*n if you remove the */
          a=malloc(3*n * sizeof(double));   /* SYMstorage option in uni2d */
          f=malloc(n * sizeof(double));
          x=malloc(n * sizeof(double));
/*        next call function to set entries;
          observe that the function defines offdiagonal entries
          only in either the upper or the lower part of the matrix,
          i.e., uses symmetric strorage for the (symmetric) system matrix */
          uni2d(nhinv-1,f,a,ja,ia);
/*      call auxiliary function to tell AGMG that 0-based indexing is used */
          dagmg_intparamC("ZeroBaseIndex",1,0);
/*      call auxiliary function to tell AGMG that symmetric strorage is used */
          dagmg_intparamC("SYMstorage",1,0);
/*      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) */
        dagmg_(&n,a,ja,ia,f,x,&zero,&iprint,&one,&maxit,&tol);
/*      display on screen the output info returned by AGMG in f[] */
        printf("\n");
        printf("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣AGMG␣status␣:%5i\n",(int)f[0]);
        printf("Number␣of␣performed␣iterations␣:%5i\n",(int)f[1]);
        printf("␣␣␣␣␣␣␣␣Relative␣residual␣norm␣:␣%.2e\n", f[2]);
        printf("␣␣␣␣␣␣␣␣␣␣␣Convergence␣history␣:␣#iter␣␣␣Residual␣Norm\n");
        for (i=0;i<=(int)(f[1]);i++){
          printf ("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%5i␣␣␣␣␣%.5e\n",i,f[i+3]);}
      }
```

2.5 Printed output

When running the above example (in either language), AGMG prints 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)
```

```
****                 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)

****       Setup time (Elapsed):     6.40E-02 seconds

****SOLUTION: flexible conjugate gradient iterations (FCG(1))
****     AMG preconditioner with Gauss-Seidel smoothing
****     ( 1 pre- and 1 post- relaxations )
****  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:    11.37 per digit of accuracy (*)
****    Solution time (Elapsed):     1.23E-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 and some others, however, can be changed by expert users, see Section 8.

The quality threshold is the threshold used to accept or not a tentative aggregate when applying the coarsening algorithms from [5, 8]; `BlockD` indicates that the algorithm from [5] is used (quality for block diagonal smoother), whereas `Jacobi` is printed instead when the algorithm from [8] 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, 8]. 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.

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 matrix 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 [7, 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 [7, Section 3]. This allows AGMG 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.

Eventually, AGMG reports on the time elapsed during this setup phase (wall clock time).

Next one enters the solution phase. AGMG informs about the used iterative method (as defined via the input argument `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. When solving for several right hand sides ($\texttt{NRHS} > 1$), AGMG reports only the mean of the relative residual norms (as defined in Section 2.3.3).

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 cumulative 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 [7, Section 3]), one will have #cycle=#call and mean=max=1.

Finally, the cost of this solution phase is reported, in term of "work units", one work unit being number of floating point operations needed for one residual evaluation (at top level: computation of $\mathbf{b} - A\,\mathbf{x}$ for some $\mathbf{b}$, $\mathbf{x}$). AGMG reports the number of needed work units 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 elapsed time (wall clock).

3 Multithread version

Professional version only

From the user viewpoint, there is no difference between calling the sequential and multithread versions.

Drivers, auxiliary functions, naming convention and input arguments are thus exactly as described in the previous section.

Which version is used (sequential or multithread) is determined by the object file the application program is linked with: those with name containing *_seqmth* provide both the sequential and the multithread version, those without provide only the purely sequential version.

The number of used threads can be specified by setting the internal parameter `nthreads` as indicated in Section 2.1.3. If `nthreads` is set to 1, AGMG switches back to the purely sequential version. If `nthreads` ≤ 0 (default), the number of threads will be automatically selected (the result depends on the system, the OpenMP implementation, and, if defined, the contents of the OMP_NUM_THREADS environment variable).

Note that the variable `nthreads` is only significant when a setup has to be done, that is, when `ijob` $= 0, 1, 10, 100, 101$ or 110. For other values of `ijob`, AGMG will use the same number of threads as for the previous setup regardless the value of `nthreads`.

If the input argument `iprint` is set to a positive number, AGMG will report in the first output lines about the number of threads actually used except if `nthreads` $= 1$ (thus confirming that one has correctly linked with the multithread version, since otherwise nothing is printed).

With the multithread version, the `status` variable may take two additional values (`status` $=$ 13 or `status` $= 14$), which correspond to failures that have been so far never met (thus a theoretical possibility), where a computer system spawns a given number of threads at setup time, but refuses later on to spawn the same number, preventing AGMG to access the related memory.

`status`	Meaning
13	Previous setup cannot be cleaned because the system failed to spawn the same number of threads.
14	Previous setup cannot be used because the system failed to spawn the same number of threads.

4 MPI version

Professional version only

We assume that the previous section about the sequential version has been carefully read. Many features are indeed common between both versions, such as most input and output arguments, and the meaning of output lines. This information will not be repeated here, where the focus is on the specific features of the MPI version.

We also assume that the reader is somewhat familiar with the MPI standard.

To work, the MPI version needs that several instances of the application program have been launched in parallel and have initialized a MPI framework, with a valid communicator. Moreover, the MPI implementation of AGMG assumes that a partitioning of the rows of the system matrix has been done in the calling program before calling AGMG. (Thus, if the matrix has been generated sequentially, some partitioning tool like METIS [3] should be called beforehand.)

Basically, the AGMG drivers requires then that each processors or MPI rank, referred to here as "*tasks*", transmits as input argument the rows that are "local" according to this partitioning, as well as the corresponding part of the right hand sides vector(s), and, if applicable, the corresponding part of the initial approximation. On output, AGMG will deliver on each task the part of the solution vector(s) that corresponds to local rows.

Note that 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 rows. The used 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 tasks.

There are two main drivers; `AGMGPARG` uses the global numbering of the unknowns and is the easiest to use; `AGMGPAR` avoids any reference to a global numbering at the price of some complications, and is reserved to expert users. (It is mainly maintained for backward compatibility reasons, as it was the only available driver with early versions of AGMG.)

4.1 Calling `AGMGPARG`

Fortran Syntax:

```
call dagmgparg(n,a,ja,ia,f,x,ijob,iprint,nrest,maxit,tol,MPI_COMM)
```

C/C++ Syntax[4]:

```
dagmgparg_(&n,a,ja,ia,f,x,&ijob,&iprint,&nrest,&maxit,&tol,&MPI_COMM);
```

or

```
DAGMGPARG(&n,a,ja,ia,f,x,&ijob,&iprint,&nrest,&maxit,&tol,&MPI_COMM);
```

[4]See footnote page 7.

Arguments:

`n`:	INPUT	integer
`a`:	INPUT	double precision (array/pointer)
`ja`:	INPUT	integer (array/pointer)
`ia`:	INPUT	integer (array/pointer)
`f`:	IN/OUT	double precision (array/pointer)
`x`:	IN/OUT	double precision (array/pointer)
`ijob`:	INPUT	integer
`iprint`:	INPUT	integer
`nrest`:	INPUT	integer
`maxit`:	INPUT	integer
`tol`:	INPUT	double precision
`MPI_COMM`:	INPUT	integer

This driver assumes that a global numbering of the unknowns has been set up in the calling program. This global numbering must satisfy the two following requirements.

1. The local rows in every task form a block of consecutive rows in the global matrix.
2. The global numbering is consistent with the MPI rank ordering: local rows in task with rank 0 form the first block of rows, local rows in task with rank 1 the next one, etc.

Then, on each task, the argument `n` is the number of local rows and the arrays `a`, `ja`, `ia` contain the submatrix corresponding to these local rows. Note that they should contain the whole rows, regardless whether column indices point to local or non local rows. (Formally, the matrix that is input on each task is thus a rectangular matrix.)

Long (8 bytes/64 bits) integer as input argument.
An alternative driver is provided for the case where *all* the integer arguments are or type long (8 bytes/64 bits) in the calling program.

Fortran Syntax:

```
call dagmgparg8(n,a,ja,ia,f,x,ijob,iprint,nrest,maxit,tol,MPI_COMM)
```

C/C++ Syntax[5]:

```
dagmgparg8_(&n,a,ja,ia,f,x,&ijob,&iprint,&nrest,&maxit,&tol,&MPI_COMM);
```

or

```
DAGMGPARG8(&n,a,ja,ia,f,x,&ijob,&iprint,&nrest,&maxit,&tol,&MPI_COMM);
```

[5]See footnote page 7.

4.1.1 Matrix format

Because the MPI version of AGMG is row oriented, neither the CSC format nor the symmetric storage option is available. The default format is CSR, but the COO format can be activated in a similar way as in the sequential case (see Section 4.3 below). It is also possible to specify that one uses 0-based indexing. Note that, as in the sequential case, AGMG works automatically in assembly mode when duplicated entries appear in the input list (see Section 2.1.2).

If `COOformat` > 0 : *COO (Coordinate) format*
This format is the easier to use: nonzero matrix entries (numerical values) are stored in `a`, whereas `ja` and `ia` carry the corresponding column and row indices, respectively. The number of nonzero entries to process has to be specified before calling the main driver via the parameter `nnzinput` (the threes arrays `a`, `ja` and `ia` must have length at least `nnzinput`). **The global numbering should be used for both row and column indices, and the user should make sure that on every task all input row indices correspond to local rows.** (Otherwise AGMG will not work properly.)
Remark 1: AGMG returns to the main program with a fatal error message if `COOformat` > 0 while `nnzinput` has not been set to a positive integer.
Remark 2: the format is non-unique; entries may appear in any order.

Default: *CSR (Compressed Sparse Row) format*
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. A slight complication occurs because, on each task, only local rows are to be input, hence it would be a waste of resources to the define the pointer array `ia` for every global row. Therefore, the assumed length of `ia` is only `n`+1 (regardless the global number of unknowns), and `ia`(1) should point to the beginning of the first *local* row in `a` and `ja`, `ia`(2) to the beginning of the next one, etc. However, regarding column indices in `ja`, the global numbering should be used. Formally, it amounts to use on each task the CSR format for the corresponding rectangular input matrix after having converted the (global) row indices to $1, \ldots,$ `n` (leaving column indices unchanged). As in the sequential case, `ia`(`n`+1) must be defined in such a way that the last valid entry in arrays `a` and `ja` correspond to index k =`ia`(`n`+1)-1.
Remark 1: (for C users) the above description matches Fortran convention; in C/C++ programs, `ia`(i) is accessed as `ia`[i-1] and `ia`(`n`+1) as `ia`[`n`]; see further below for using AGMG with 0-based indexing according to C/C++ conventions.
Remark 2: the format is non-unique; entries in each row may appear in any order.

If `ZeroBaseIndex` > 0 : *Use 0-based indexing* (for whatever format)
By default, AGMG uses 1-based indexing; that is, indexing of arrays start at 1 and row & column indices range from 1 to `n`. This corresponds to Fortran conventions. By setting `ZeroBaseIndex` to positive, 0-based indexing is used instead, according to C/C++ conventions. That is, indices stored in `ja` range from 0 to `n`-1 (all formats), as well as the row indices stored in `ia` when using the COO format (`COOformat` > 0) . Further, when using the CSR format (`COOformat` ≤ 0), `ia` is a 0-based pointer; that is, the first entry in `ia` has

to set to be set to 0 (instead of 1 if `ZeroBaseIndex` ≤ 0) to tell AGMG that the first valid matrix entry in `a` and `ja` is the first one in these arrays.

Example By way of illustration, consider the matrix

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

partitioned into 3 tasks. We assume that Task 0 receives rows 1 & 2, Task 1 rows 3 & 4, and Task 2 row 5 (which is consistent with the requirements stated above).

This yields (for instance, as both formats are non-unique):

$$\begin{array}{lcl@{\qquad}rcl}
 & & \text{COO} & & & \text{CSR} \\[1ex]
\textsc{Task 0} & & (\mathtt{n}=2) & & & \\
\mathtt{a} & = & \left[\ 10.0 \quad -1.0 \,\middle|\, -2.0 \quad 11.0 \quad -3.0\ \right] & \mathtt{a} & = & \left[\ 10.0 \quad -1.0 \,\middle|\, -2.0 \quad 11.0 \quad -3.0\ \right] \\
\mathtt{ja} & = & \left[\ 1 \quad 4 \,\middle|\, 1 \quad 2 \quad 4\ \right] & \mathtt{ja} & = & \left[\ 1 \quad 4 \,\middle|\, 1 \quad 2 \quad 4\ \right] \\
\mathtt{ia} & = & \left[\ 1 \quad 1 \,\middle|\, 2 \quad 2 \quad 2\ \right] & \mathtt{ia} & = & \left[\ 1 \quad 3 \quad 6\ \right] \\[1ex]
\textsc{Task 1} & & (\mathtt{n}=2) & & & \\
\mathtt{a} & = & \left[\ -4.0 \quad 12.0 \quad -5.0 \,\middle|\, 13.0\ \right] & \mathtt{a} & = & \left[\ -4.0 \quad 12.0 \quad -5.0 \,\middle|\, 13.0\ \right] \\
\mathtt{ja} & = & \left[\ 2 \quad 3 \quad 5 \,\middle|\, 4\ \right] & \mathtt{ja} & = & \left[\ 2 \quad 3 \quad 5 \,\middle|\, 4\ \right] \\
\mathtt{ia} & = & \left[\ 3 \quad 3 \quad 3 \,\middle|\, 4\ \right] & \mathtt{ia} & = & \left[\ 1 \quad 4 \quad 5\ \right] \\[1ex]
\textsc{Task 2} & & (\mathtt{n}=1) & & & \\
\mathtt{a} & = & \left[\ 14.0 \quad -9.0 \quad -8.0\ \right] & \mathtt{a} & = & \left[\ 14.0 \quad -9.0 \quad -8.0\ \right] \\
\mathtt{ja} & = & \left[\ 5 \quad 3 \quad 2\ \right] & \mathtt{ja} & = & \left[\ 5 \quad 3 \quad 2\ \right] \\
\mathtt{ia} & = & \left[\ 5 \quad 5 \quad 5\ \right] & \mathtt{ia} & = & \left[\ 1 \quad 4\ \right]
\end{array}$$

4.1.2 Other input arguments

The arguments `f` and `x` have the same meaning as in the sequential case, except that each task needs to receive and will return only the portion of these vectors corresponding to local rows.

Regarding the arguments `ijob`, `iprint`, `nrest`, `maxit` and `tol`, they have exactly the same meaning as in the sequential case; see Section 2.1.3. The only difference, already mentioned there, is that with the MPI version it is not permitted to transpose the input matrix. Hence `ijob`=100,101,102,110,112 are forbidden.

Finally, the new input argument `MPI_COMM` has to contain the relevant MPI communicator. (Most often, this will be MPI_COMM_WORLD.)

Important remark: whereas `ijob`, `nrest`, `maxit`, `tol` and `MPI_COMM` **should be assigned the same value on all tasks**, `iprint` *may be rank dependent*. In fact, it is advised to use this possibility to separate the output generated by the different tasks. In case of massive parallelism, it is even strongly advised against using positive `iprint` on more than just a few tasks. (In general, setting `iprint` to positive only for the Task with rank 0 provides enough information.)

4.2 Calling AGMGPAR

Fortran Syntax:

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

C/C++ Syntax [6]:

```
    dagmg_(&n,a,ja,ia,f,x,&ijob,&iprint,&nrest,&maxit,&tol,&MPI_COMM,
           listrank,&ifirstlistrank);
```

or

```
    DAGMGPAR(&n,a,ja,ia,f,x,&ijob,&iprint,&nrest,&maxit,&tol,&MPI_COMM,
           listrank,&ifirstlistrank);
```

New arguments (see Section 4.1 for the other ones):

`listrank`:	INPUT	integer (array/pointer)
`ifirstlistrank`:	INPUT	integer

This driver does not refer to any global ordering of the unknowns. Instead, one has to define on each task some LOCAL ORDERING of the unknowns, such that local rows (and thus local variables) have numbers $1, \dots, \texttt{n}$, while nonlocal variable are given arbitrary indices larger than `n` (see below).

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 $1 \leq i \leq \texttt{n}$) corresponds to a local connection if $j \leq \texttt{n}$ and to an external connection if $j > \texttt{n}$.

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 (with, possibly, a numerical value equal to zero) referencing (an external variable corresponding to) i.

CONSISTENCY OF LOCAL ORDERINGS. Besides the condition that they are larger than `n`, indices of nonlocal variable may be chosen arbitrarily, providing that their ordering is

[6]See footnote page 7.

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 $j\,,\,l > \texttt{n}$, 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` ($\texttt{n} < j_{\min} \leq j_{\max}$). 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 ***negative*** 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 $i \leq \texttt{n}$).

Example

By way of illustration, consider the same matrix as in the previous section 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

$$\begin{array}{rcl}
\texttt{n} & = & 2 \\
\texttt{a} & = & \left[\begin{array}{cc|ccccc} 10.0 & -1.0 & -2.0 & 11.0 & -3.0 & 0.0 & 0.0 \end{array}\right] \\
\texttt{ja} & = & \left[\begin{array}{cc|ccccc} 1 & 4 & 1 & 2 & 4 & 3 & 5 \end{array}\right] \\
\texttt{ia} & = & \left[\begin{array}{ccc} 1 & 3 & 8 \end{array}\right] \\
\texttt{listrank} & = & \left[\begin{array}{ccccc} * & * & 1 & 1 & 2 \end{array}\right]
\end{array}$$

Task 1

$$\begin{array}{rcl}
\texttt{n} & = & 2 \\
\texttt{a} & = & \left[\begin{array}{ccc|ccc} -4.0 & 12.0 & -5.0 & 13.0 & 0.0 & 0.0 \end{array}\right] \\
\texttt{ja} & = & \left[\begin{array}{ccc|ccc} 7 & 1 & 5 & 2 & 6 & 7 \end{array}\right] \\
\texttt{ia} & = & \left[\begin{array}{ccc} 1 & 4 & 7 \end{array}\right] \\
\texttt{listrank} & = & \left[\begin{array}{ccccccc} * & * & * & * & 2 & 0 & 0 \end{array}\right]
\end{array}$$

Task 2

$$\begin{array}{rcl}
\texttt{n} & = & 1 \\
\texttt{a} & = & \left[\begin{array}{ccc} 14.0 & -9.0 & -8.0 \end{array}\right] \\
\texttt{ja} & = & \left[\begin{array}{ccc} 1 & 5 & 3 \end{array}\right] \\
\texttt{ia} & = & \left[\begin{array}{cc} 1 & 4 \end{array}\right] \\
\texttt{listrank} & = & \left[\begin{array}{ccccc} * & * & 0 & -1 & 1 \end{array}\right]
\end{array}$$

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.

Common error: don't forget to initialize "holes" in `listrank` with negative numbers.

Other input arguments: See Section 4.1.2.

Alternative driver: `dagmgpar8` should be called instead if *all* the integer arguments are or type long (8 bytes/64 bits) in the calling program.

4.3 Changing internal parameters

For whatever driver, internal parameters can be changed in the same way as for the sequential version. The call is as follows.

Fortran Syntax:

```
call dagmgpar_intparam("key",value,instance)
```

C/C++ Syntax [7]:

```
dagmgpar_intparamC("key",value,instance);
```

Arguments:

`"key"`:	INPUT	string
`value`:	INPUT	integer
`instance`:	INPUT	integer

The usage and the list of adjustable parameters is as indicated at the beginning of Section 2.1 for the sequential case: only the name of the function differs. The only peculiarity, already mentioned, is that CSC format and symmetric storage options are deactivated. Hence trying to change the corresponding parameters (`CSCformat` or `SYMstorage`) will have no effect besides the printing of a warning message.

In particular the COO format mentioned above is activated by setting the parameter `COOformat` to positive and setting the parameter `nnzinput` to the number of relevant entries in arrays `a`, `ja` and `ia`.

Long (8 bytes/64 bits) integer as input argument.
If the input integers arguments `value` and `instance` are of type long (8 bytes/64 bits), Fortran users should use the alternative function:

```
call dagmgpar8_intparam("key",value,instance)
```

No alternative function is provided for C/C++ users, but they can easily edit the declaration in the function `dagmgpar_intparamC` in the provided source file so as to match the type of integer used when calling the function (other declarations should not be modified).

4.4 Solving at once for multiple right hand sides

In the MPI case as well, AGMG can solve at once for more than one right hand side, by setting (with a call to the auxiliary function as explained above), `NRHS` to the number of right hand sides *and* `MajOrd` to either positive or negative, according to the type of ordering used in arguments `f` and `x` (major row or major column ordering, respectively). We refer to Section 2.2 for details, as whatever is written there applies to the MPI version as well.

4.5 Output arguments and error flags

All what is written in Section 2.3 applies to the MPI version as well, except that the naming of the functions to access the value of `status` and `iter` is modified: one has to exchange

[7]The name is here `dagmg_intparamC` for all compilers and OS because this function is specific to the C/C++ version and supplied in a separated C source file (dagmgpar_intparamC.c).

`dagmg` for `dagmgpar` (as for the auxiliary functions to change internal parameters). Hence to access `status` the call is:

Fortran Syntax:

```
call dagmgpar_status(status,instance)
```

C/C++ Syntax:

```
dagmgpar_status_(&status,&instance);
```

or

```
DAGMGPAR_STATUS(&status,&instance);
```

Or, if long (8 bytes/64 bits) integers are used in the calling program:

Fortran Syntax:

```
call dagmgpar8_status(status,instance)
```

C/C++ Syntax:

```
dagmgpar8_status_(&status,&instance);
```

or

```
DAGMGPAR8_STATUS(&status,&instance);
```

See Section 2.3.1 for the possible values of `status` and their meanings.
(As in the sequential case, `instance` is ignored by default and is only meaningful in the special *several instances mode* described in Section 6.)

Similarly, the number of iterations performed during the last call to the MPI version of AGMG is accessed with:

Fortran Syntax:

```
call dagmgpar_iter(iter,instance)
```

C/C++ Syntax:

```
dagmgpar_iter_(&iter,&instance);
```

or

```
DAGMGPAR_ITER(&iter,&instance);
```

Or, if long (8 bytes/64 bits) integers are used in the calling program:

Fortran Syntax:

```
call dagmgpar8_iter(iter,instance)
```

C/C++ Syntax:

```
dagmgpar8_iter_(&iter,&instance);
```

or

```
DAGMGPAR8_ITER(&iter,&instance);
```

Finally the argument `f` is also used to return information to the calling program in case where `ijob`=0,2,3,10,12,202 or 212, exactly in the same way as in the sequential case, see Section 2.3.3. Note that this information is returned on each task. (Since all returned quantities are global, the same contents is returned on all MPI ranks.)

4.6 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`. However, using the driver **`dagmgparg`**, it is not needed to provide such information to AGMG. All what is needed is to have the correct global indices, which each task deduces in the program below by evaluating the total number of unknowns in all tasks with lower rank (variable `nprev`).

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

Listing 3: source code of MPI Example (Fortran 90)

```
      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 (not optimal, but makes the program easier to read).
!
       implicit none
       include 'mpif.h'
       real (kind(0d0)),allocatable :: a(:),f(:),x(:)
       integer,allocatable :: ja(:),ia(:)
       integer :: n,maxit,iprint,nhinv,NPROC,IRANK,mx,my,nprev,nnz,ierr,i
       real (kind(0d0)) :: tol
       character*10 filename
!
!      set inverse of the mesh size (feel free to change)
       nhinv=1000
!
!      maximal number of iterations
       maxit=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 the local grid size and the total number of unknowns
!       in previous task with lower rank
!       (adding 1, this gives the global index of the first local unknown)
!
        mx=nhinv-1
        if (IRANK < mod(nhinv-1,NPROC)) then
           my=(nhinv-1)/NPROC+1
           n=mx*my
           nprev=mx*my*IRANK
        else
           my=(nhinv-1)/NPROC
           n=mx*my
           nprev=mx*(my*IRANK+ mod(nhinv-1,NPROC))
        end if
!
!       generate the local matrix in COO format (using global indices)
!
!         first allocate the vectors with correct size
            n=mx*my
            allocate (a(5*n),ja(5*n),ia(5*n),f(n),x(n))
            call uni2dstrip(mx,my,f,a,ja,ia,IRANK,NPROC,nprev,nnz)
!
!       call auxiliary function to tell AGMG that COO format is used
           call dagmgpar_intparam("COOformat",1,0)
!       call auxiliary function to inform AGMG about the number of entires in
!         arrays a, ja and ia
           call dagmgpar_intparam("nnzinput",nnz,0)
!       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)
!         before calling, fix the number of threads to 2 per MPI rank;
!           this will be significant only when liking with the hybrid version
!           of AGMG
           call dagmgpar_intparam("nthreads",2,0)
!         finally, observe that we call the driver for global column and row
!            indices in ja and ia (for this latter, when CCO format is used)
        call dagmgparg(n,a,ja,ia,f,x,0,iprint,1,maxit,tol,MPI_COMM_WORLD)
!
!       display on screen the output info returned by AGMG in f()
!       all ranks have the same info, but only one has to print it
```

```
        if (irank.eq.0) then
           print '()'
           print '("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣AGMG␣status␣:",i5)', int(f(1))
           print '("Number␣of␣performed␣iterations␣:",i5)', int(f(2))
           print '("␣␣␣␣␣␣␣␣Relative␣residual␣norm␣:",1pe9.2)', f(3)
           print '("␣␣␣␣␣␣␣␣␣␣␣Convergence␣history␣:␣#iter␣␣␣Residual␣Norm")'
           print '("␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣",i5,1pe16.5)' &
                ,(i,f(4+i),i=0,int(f(2)))
        endif
!
!       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)
!
        call MPI_FINALIZE(ierr)
        deallocate (a,ja,ia,f,x)
!
      end program example_par
!-----------------------------------------------------------------------
      subroutine uni2dstrip(mx,my,f,a,ja,ia,IRANK,NPROC,nprev,nnz)
!
! Fill a matrix in COO 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
! Left and right boundaries are always real boundaries
        implicit none
        real (kind(0d0)) :: f(*),a(*)
        integer :: mx,my,ia(*),ja(*),nprev,nnz
        integer :: IRANK,NPROC,k,l,i,j
        real (kind(0d0)), parameter :: zero=0.0d0,cx=-1.0d0,cy=-1.0d0, cd=4.0d0
        !
        k=nprev
        l=0
        do i=1,my
           do j=1,mx
              k=k+1
              l=l+1
              a(l)=cd
              ja(l)=k
              ia(l)=k
              f(k-nprev)=zero
              if(j < mx) then
                 l=l+1
           a(l)=cx
           ja(l)=k+1
           ia(l)=k
        else
           f(k-nprev)=f(k-nprev)-cx
        end if
        if(i < my .OR. IRANK /= NPROC-1) then
           l=l+1
           a(l)=cy
           ja(l)=k+mx
           ia(l)=k
        else
           f(k-nprev)=f(k-nprev)-cy            !real boundary
        end if
        if(j > 1) then
           l=l+1
           a(l)=cx
           ja(l)=k-1
           ia(l)=k
        else
           f(k-nprev)=f(k-nprev)-cx
        end if
        if(i >  1 .OR. IRANK /= 0) then
           l=l+1
           a(l)=cy
           ja(l)=k-mx
           ia(l)=k
        else
           f(k-nprev)=f(k-nprev)-cy            !real boundary
        end if
     end do
  end do
  nnz=l
  !
  return
end subroutine uni2dstrip
```

4.7 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): 10.00 ;  Strong diag. dom. trs: 1.22
****         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:       15704          (reduction ratio: 7.93)
  0*       Number of local rows:        3860          (reduction ratio: 8.10)
****  Global number of nonzeros:       79406 (per row: 5.1; red. ratio: 7.84)
  0*     Nonzeros in local rows:       19390 (per row: 5.0; red. ratio: 8.05)

  0*                    Level:          4
**** Global number of variables:        1963          (reduction ratio: 8.00)
  0*       Number of local rows:         465          (reduction ratio: 8.30)
****  Global number of nonzeros:       10035 (per row: 5.1; red. ratio: 7.91)
  0*     Nonzeros in local rows:        2294 (per row: 4.9; red. ratio: 8.45)

****           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*       Setup time (Elapsed):     1.78E-01 seconds
  0*SOLUTION: flexible conjugate gradient iterations (FCG(1))
****     AMG preconditioner with Gauss-Seidel smoothing
****     ( 1 pre- and 1 post- relaxations )
****  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.= 0.86E-01
****  Iter=    3        Resid= 0.20E+01       Relat. res.= 0.31E-01
****  Iter=    4        Resid= 0.54E+00       Relat. res.= 0.85E-02
****  Iter=    5        Resid= 0.30E+00       Relat. res.= 0.47E-02
****  Iter=    6        Resid= 0.84E-01       Relat. res.= 0.13E-02
****  Iter=    7        Resid= 0.39E-01       Relat. res.= 0.62E-03
****  Iter=    8        Resid= 0.11E-01       Relat. res.= 0.18E-03
****  Iter=    9        Resid= 0.40E-02       Relat. res.= 0.63E-04
****  Iter=   10        Resid= 0.20E-02       Relat. res.= 0.31E-04
****  Iter=   11        Resid= 0.77E-03       Relat. res.= 0.12E-04
****  Iter=   12        Resid= 0.28E-03       Relat. res.= 0.45E-05
****  Iter=   13        Resid= 0.11E-03       Relat. res.= 0.18E-05
****  Iter=   14        Resid= 0.41E-04       Relat. res.= 0.64E-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
****     level 4   #call=    56   #cycle=    84   mean=   1.50    max=  4

  0*       Number of work units:    11.51 per digit of accuracy (*)
  0*    Solution time (Elapsed):     3.15E-01 seconds

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

Most comments made in Section 2.5 apply here as well. The main difference is that, 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, reported number of work units for the solution phase is based on local the number of local floating point operations 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.

5 Hybrid version

Professional version only

The hybrid version adds multithreading on top of the MPI version; that is, each MPI rank will run in multithread mode.

From the user viewpoint, there is no difference between calling the MPI and hybrid versions.

Drivers, auxiliary functions, naming convention and input arguments are thus exactly as described in the previous section.

Which version is used (MPI or hybrid) is determined by object file the application program is linked with: those with name containing *_mpihyb* provide the hybrid version, those containing *_mpi* provide the pure MPI version.

The number of used threads can be specified by setting the internal parameter `nthreads` as indicated in Section 4.3. This is highly recommended, as system defaults in general ignore how many MPI tasks (or MPI ranks) are running concurrently on the compute unit. Moreover, **every task has to use the same number of threads**, otherwise AGMG cannot work properly. The safest way to enforce this is to fix this number of threads via the variable `nthreads` by calling the auxiliary function `dagmgpar_intparam` (Fortran) or `dagmgpar_intparamC` (C/C++).

In the remaining of this section we repeat the information given in Section 3 about the multithread version, as it applies to the hybrid version as well.

If `nthreads` is set to 1, AGMG switches back to the pure MPI version, whereas, if `nthreads` ≤ 0 (default), the number of threads will be automatically selected
(the result depends on the system, the OpenMP implementation, and, if defined, the contents of the OMP_NUM_THREADS environment variable).

Note that the variable `nthreads` is only significant when a setup has to be done, that is, when `ijob` $= 0, 1, 10, 100, 101$ or 110. For other values of `ijob`, AGMG will use the same number of threads as for the previous setup regardless the value of `nthreads`.

If the input argument `iprint` is set to a positive number, AGMG will report in the first output lines about the number of threads actually used except if `nthreads` $= 1$ (thus confirming that one has correctly linked with the hybrid version, since otherwise nothing is printed).

With the hybrid version, the `status` variable may take two additional values (`status` $= 13$ or `status` $= 14$), which correspond to failures that have been so far never met (thus a theoretical possibility), where a computer system spawns a given number of threads at setup time, but refuses later on to spawn the same number, preventing AGMG to access the related memory. See the end of Section 3 for the table of values and their exact meaning.

6 Several instances mode

In some contexts, it is interesting to hold simultaneously in memory several instances of the AGMG preconditioner, related to different matrices (with possibly different sizes). For example, 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. The *several instances mode* has been designed for this purpose.

One enters this mode by setting the internal parameter `MaxInstance` to a positive value equal to the maximl number of instances one would like to hold simultaneously in memory. Except for Octave and Matlab versions (see the provided help for these), this is done by calling the auxiliary function `dagmg_intparam` or `dagmg_intparamC` as explained in Section 2.1.1, with first argument set to `"MaxInstance"` (case sensitive!), second argument set to the desired value and third argument set to zero.[8]

Once in *several instances mode*, all calls to AGMG drivers should be associated with a positive instance number `instance` not larger than `MaxInstance`. This is done by setting the `ijob` argument to any nonnegative vale listed in Table 1 plus 1000×`instance`. Then, `ijob`=1001 means setup for instance 1, `ijob`=3202 means solve for instance 3, `ijob`=2099 means memory release for instance 2, etc.

In *several instances mode*, `ijob`=-1 is still allowed and means erase the setup and release internal memory of *all* instances, being thus a shortcut for successive calls with `ijob`=1099, `ijob`=2099, etc.

One exits the *several instances mode* by resetting `MaxInstance` to 0. **It is recommended to always reset `MaxInstance` to 0 after the last call to AGMG** because the *several instances mode* requires a small amount of additional memory that is only released when explicitly exiting the mode. (In fact, this amount of memory is practically insignificant, but its presence will induce an undesirable memory leak detection if the software is run under the control of a memory checking application.)

Note that the several instances mode is not yet implemented for the MPI and Hybrid versions of AGMG.

Rules, restrictions and remarks

- `MaxInstance` cannot not be changed while a setup is held in memory. This, in particular, applies when one is willing to enter the *several instances mode* by setting `MaxInstance` to positive: if a previous setup is still held in memory, an error message will be issued and one will not enter the mode. Similarly, it is needed to erase the setup of all instances before exiting the mode by resetting `MaxInstance` to 0.

- Thus, in *several instances mode*, admitted values of `ijob` are -1 (erase all setups) plus those such that

[8]The third argument is always ignored when the first argument is `"MaxInstance"`; note also that function names change for complex matrices, see Section 9).

`instance` $= \lfloor$`ijob`$/1000\rfloor$ *is a valid instance number* (> 0 and at most `MaxInstance`) and

`ijb` = `ijob` mod 1000 *is one of the nonnegative value listed in Table 1.*

Then, AGMG will perform the action associated with `ijb` on the indicated instance.

Note that, in particular, in *several instances mode*, nonnegative values of `ijob` should be at least 1000.

- When compiled with the OpenMP flag, the purely sequential version of AGMG is threadsafe in *several instances mode.* It means that calls to AGMG drivers can be done safely in parallel as soon as each of these calls is associated with a distinct instance number.[9]

  Beware that only the purely sequential code is threadsafe. The multithread version of AGMG is not, even though it would run sequentially by setting the `nthreads` parameter to 1. Thus, users have to choose between parallelism inside AGMG and parallelism at upper level with independent calls to AGMG executed in parallel.

- Regarding the multithread version, for technical reasons, AGMG has to fix the maximum number of threads for any subsequent call when the driver is called for the first time in *several instances mode.* This is done based on the available information associated with the current value of `nthreads` for each instance (see next paragraph for instance dependent adjustable parameters). If one wants to use different numbers of threads for different instances, it is therefore highly recommended to set the `nthreads` parameter for each instance before any call to AGMG driver.

Internal parameters

- When entering the several instance mode, the current values of adjustable parameters are copied to the private space of each instance.

- Once in *several instances mode*, the third argument `instance` to `dagmg_intparam` and `dagmg_intparamC` can be set to a positive number not larger than `MaxInstance`; then only that instance will be concerned by the parameter update. On the other hand, if this third argument is nonpositive, the parameter update will apply to all instances. Finally, if `instance` is set to a positive number larger than `MaxInstance`, an error message will be issued and no action will be performed.

- When exiting the *several instances mode*, the current values for adjustable parameters are retrieved from those of `instance` number 1.

[9]Note that, for obvious reasons, the setting of `MaxInstance` that makes AGMG enter the *several instance mode* should either be performed before entering the parallel loop or region or followed by a BARRIER instruction.

Integer output arguments

In *several instances mode*, when calling the function `dagmg_status` or `dagmg_iter`, it is highly recommended to set the second argument to the number of the instance one wants to retrieve information from. For other values of this second argument, the function will return the last registered value of the associated internal parameter, which might be ambiguous.

7 Solving singular systems

Singular but compatible systems can be solved by AGMG without needing to tell the software that the provided linear system is singular. However, this usage should be considered with care and a few limitations apply. Usually, AGMG works smoothly when the dimension of the null space is at most 1 (see below for larger dimensions): if the matrix at coarsest level is also singular, AGMG will detect this and prevent instabilities associated with the direct inversion of a singular matrix.

However, this detection is heuristic, hence may fail in some cases. When AGMG detects a singular matrix at coarsest level, this is reported on the printed output, allowing the user to check what happened. Note that nothing reported (thus no singularity detected) does not mean that something wrong happened because it is case dependent whether or not the singularity of the system matrix is transferred at coarsest level. If a singular coarsest grid matrix is not properly detected, this may be associated with convergence instabilities, especially with the conjugate gradient method (`nrest`=1), while the computed solution may be undesirably dominated by kernel components. Feel free to contact support@agmg.eu if you think you are running in such a case.

When the dimension of the null space is larger than 1, the standard detection procedure may fail because it is tailored for a coarsest grid matrix with at most 1 singular mode. However, this is easily corrected by setting the `MaxCoaKernVect` internal parameter to the null space dimension; `MaxCoaKernVect` is the maximal number of coarse kernel vectors that will be searched for. If this parameter is larger than 1, when detecting a singularity at coarsest level, AGMG will additionally report about the numbers of detected kernel vector(s).

Eventually, any detection is prevented by setting `MaxCoaKernVect` to 0. *This is recommended if and only if AGMG reports that it treats the coarsest grid matrix as singular in case it should not*, especially if this seemingly induces convergence issues.

As alternative approach, note that AGMG can also often be efficiently applied to solve the near singular system obtained by deleting properly row(s) and column(s) in a linear system originally singular (and compatible). This approach is not recommended in general with iterative solvers because the truncated system obtained in this way tends to be ill-conditioned and hence hard to solve with iterative methods. 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 efficiently solve ill-conditioned linear systems.

8 Tuning AGMG parameters

List of adjustable internal parameters *for tuning by expert users*:
`smoothtype` (default:0) , `percent_omega` (default:70)
`npresmooth(1)` (default:1) , `npostmooth(1)` (default:1)
`npresmooth(2)` (default:1) , `npostsmooth(2)` (default:1)
`npass` (default:2; 3 for the MPI and Hybrid version)
`targetcoarsening` (default:0)
`qualitybound` (default:0) , `percent_dd` (default:0)
`percent_trspos` (default:45)
`percent_negrcsum` (default:25)
`nstep` (default:-1) , `maxcoarsesize` (default:40)
`percent_resi` (default:20) , `nlvcyc` (default:0)

List of additional adjustable internal parameters for the MPI and Hybrid versions:
`MAXITcoarse` (default:10) , `percent_tolcoarse` (default:45)

Short overview

`smoothtype` indicates which smoother is used.
If `smoothtype`=1, the smoother is based on Gauss-Seidel sweeps;
If `smoothtype`=0, the smoother is based on SOR sweeps with automatic estimation of the relaxation parameter (often reduces to Gauss-Seidel);
If `smoothtype`=-1, the smoother is based on SOR sweeps with relaxation parameter `percent_omega`/100 (e.g., `percent_omega`=70 ⇒ rel. param. $= 0.70$)
Scheme used in all three cases:
pre-smoothing: Forward sweep, then Backward sweep, then Forward, etc
post-smoothing: Backward sweep, then Forward sweep, then Backward, etc.
If `smoothtype`=2, the smoother is ILU(0)

`npresmooth(1)`: number of pre-smoothing steps at fine level (minimum: 0).
`npostmooth(1)`: number of post-smoothing steps at fine level (minimum: 1).
`npresmooth(2)`: number of pre-smoothing steps at other levels (minimum: 0).
`npostmooth(2)`: number of post-smoothing steps at other levels (minimum: 1).

`npass` is the maximal number of pairwise aggregation passes for each coarsening step, according to the algorithms in [5, 8]

`targetcoarsening` is the target coarsening factor (parameter τ in the main coarsening algorithms in [5, 8]): further pairwise aggregation passes are omitted once the number of nonzero entries has been reduced by a factor of at least `targetcoarsening`. If `targetcoarsening`≤ 0, then the default value $2^{\texttt{npass}}$ is used.

`qualitybound` is the threshold to accept or not a tentative aggregate according to the coarsening algorithms in [5, 8]. If `qualitybound`≤ 0, a default value is assigned, which depends on `npass` and on whether one uses the symmetric [5] or nonsymmetric [8] version of the algorithm.

`percent_dd`: the threshold to keep outside aggregation nodes where the matrix is strongly diagonally dominant (based on mean of row and column) is the maximum between the default value as indicated in [5, 8] and 1+`percent_dd`/100. Setting `percent_dd`=0 guarantees that one uses the default value, whereas setting `percent_dd` to a very large integer will ensure that no nodes will be kept outside aggregation.

`percent_trspos`/100 is a threshold: if a row has a positive offdiagonal entry larger than this threshold times the diagonal entry, the corresponding node is transferred unaggregated to the coarse grid.

`percent_negrcsum` indicates the percentage of nodes with negative mean row and column sum that is "tolerated". If, at some level, the actual number is beyond this threshold, to apply the coarsening algorithm, all diagonal entries are exchanged for the corresponding mean row and column sum (that is, the algorithm is applied to a modified matrix with mean row and column sum enforced to be zero).

`nstep` is the maximum number of coarsening steps. If `nstep`< 0, the coarsening is stopped when the (local) size of the coarse grid matrix is less than or equal to `maxcoarsesize`$\cdot$`n`$^{1/3}$ on each task or MPI rank.

`percent_resi`/100 is the threshold for the relative residual error in inner iterations at intermediate levels, see Algorithm 3.2 in [7]

`nlvcyc` is the number of coarse levels (from bottom) on which V-cycle formulation is enforced (Remark: K-cycle always used at first coarse level).

Tuning the MPI version

Compared with the sequential version, by default, the number of pairwise aggregation passes `npass` is 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 `qualitybound`, 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. 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. This effect can be obtained by adjusting the parameters `nstep` and/or `maxcoarsesize`, possibly according to the number of processors.

Note that the coarsest grid solver is itself an iterative one for the MPI version, and its efficiency may depends on both the prescribed maximum number of iterations `MAXITcoarse` and on the threshold used to stop these iterations, `percent_tolcoarse`/100. In particular, setting `MAXITcoarse`=1 works often pretty well, as it minimizes the work and the communication at coarsest level without having necessarily a significant impact on the overall convergence of the main iteration.

9 Complex version

The usage is exactly the same as the standard (real) version described in other sections, with the following peculiarities.

- When calling the drivers and associated functions, one has to replace in whatever name the prefix `d` by `z`; e.g., use `zagmg` instead of `dagmg` and `zagmg_intparamC` instead of `dagmg_intparamC`
- The input and output arguments are the same and have same type, except the double precision arrays `a`, `f` and `x` that have double complex type for the complex version.

  Alternatively, one can supply as arguments double precision arrays of length twice as large, using the convention that each double complex entry occupies two successive positions in the corresponding double precision array, the first one indicating the real part and the second one the imaginary part. This latter trick should be used when the calling program is in C/C++, since there is no native support for complex number in these languages.
- The output information returned in `f` is as described in Section 2.3.3, thus made of real numbers, implying that the imaginary part of affected entries in `f` is set to zero.
- Symmetric storage can still be used on input, but one should pay attention that it works only for symmetric complex matrices *and not* for Hermitian matrices.
- Regarding source and object files, one should link with those have prefix z instead of d.
- To link a program simultaneously with the real and complex versions of AGMG, the complex version of the MUMPS source file (zagmg_mumps.F90) should be compiled defining the macro "_NO_COMMON_PART_" [10] to skip the part of the sources that are also present in the real version of the MUMPS source file (dagmg_mumps.F90). The complex part should also be compiled *after* the real one to make sure that the needed modules from this latter can be referenced.

[10]Thus using the option "-D_NO_COMMON_PART_" on Linux and macos and (most often) using the option "/D_NO_COMMON_PART_" on Windows.

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.* `http://glaros.dtc.umn.edu/gkhome/views/metis`.

[4] *MUMPS software and documentation.* `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] A. Napov and Y. Notay, *Algebraic multigrid for moderate order finite elements*, SIAM J. Sci. Comput., 36 (2014), p. A1678–A1707.

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

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

[9] Y. Notay, *Numerical comparison of solvers for linear systems from the discretization of scalar PDEs*, 2012. `http://agmg.eu/numcompsolv.pdf`.

[10] Y. Notay and A. Napov, *A massively parallel solver for discrete poisson-like problems*, J. Comput. Physics, 281 (2015), pp. 237–250.

[11] 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`.

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

[13] H. A. Van der Vorst and C. Vuik, *GMRESR: a family of nested GMRES methods*, Numer. Linear Algebra Appl., 1 (1994), pp. 369–386.