GALAHAD NODEND

USER DOCUMENTATION GALAHAD Optimization Library version 5.2

1 SUMMARY

This package **finds a symmetric row and column permutation $\mathbf{PAP}^T$ of a symmetric, sparse matrix A with the aim of limiting the fill-in during subsequent Cholesky-like factorization**. The package is actually a wrapper to the `METIS_NodeND` procedure from versions 4.0, 5.1 and 5.2 of the `METIS` package from the Karypis Lab; Versions 5 are freely available under an open-source licence, and included here, while Version 4 requires a more restrictive licence, and a separate download, see `https://github.com/KarypisLab`; if Version 4 is not provided, a dummy will be substituted.

ATTRIBUTES — Versions: `GALAHAD_NODEND_single`, `GALAHAD_NODEND_double`. **Calls:** `GALAHAD_KINDS`, `GALAHAD_SYMBOLS`, `GALAHAD_SMT`, `GALAHAD_SORT` and `GALAHAD_SPECFILE`, **Date:** March 2025. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003.

2 HOW TO USE THE PACKAGE

2.1 Calling sequences

The package is available with either 32-bit or 64-bit integers, and the subsidiary `SMT_type` package may use single, double and (if available) quadruple precision reals. Access to the 32-bit integer, single precision version requires the `USE` statement

```
USE GALAHAD_NODEND_single
```

with the obvious substitution `GALAHAD_NODEND_double`, `GALAHAD_NODEND_quadruple`, `GALAHAD_NODEND_single_64`, `GALAHAD_NODEND_double_64` and `GALAHAD_NODEND_quadruple_64` for the other variants.

If it is required to use more than one of the modules at the same time, the derived types `SMT_type`, `NODEND_control_type` and `NODEND_inform_type` (§2.5), and the subroutines `NODEND_order` and `NODEND_order_adjacency`, (§2.6) must be renamed on one of the `USE` statements.

There are two principal subroutines for user calls.

`NODEND_order` takes the (symmetric) pattern of **A** and finds a symmetric permutation **P** so that the fill-in during Cholesky-like factorizations of $\mathbf{PAP}^T$ is kept small.

`NODEND_order_adjacency` takes the adjacency graph of the (whole) pattern of **A**, and performs the same task as `NODEND_order`. This package is actually called by its predecessor, but is provided for use by experts, and for those whose application is naturally in adjacency form.

2.2 Matrix storage formats

The sparsity pattern of the matrix **A** may be stored in a variety of input formats.

2.2.1 Sparse co-ordinate storage format

Only the nonzero entries of the lower-triangular part of **A** are stored. For the l-th entry of the lower-triangular portion of **A**, its row index i and column index j are stored in the l-th components of the integer arrays `row` and `col`, respectively. The order is unimportant, but the total number of entries `ne` is also required.

2.2.2 Sparse row-wise storage format

Again only the nonzero entries of the lower-triangular part are stored, but this time they are ordered so that those in row i appear directly before those in row $i+1$. For the i-th row of **A**, the i-th component of an integer array `ptr` holds the position of the first entry in this row, while `ptr` $(m+1)$ holds the total number of entries plus one. The column indices j of the entries in the i-th row are stored in components $l =$ `ptr`$(i), \ldots,$`ptr` $(i+1)-1$ of the integer array `col`.

For sparse matrices, this scheme almost always requires less storage than its predecessor.

2.2.3 Dense storage format

The matrix **A** is stored as a compact dense matrix by rows, that is, the values of the entries of each row in turn are stored in order within an appropriate real one-dimensional array. Since no indexing information is needed, no integer arrays are required. Indeed, there no point in reordering a dense matrix, and this option is simply included for completeness.

2.3 Matrix-graph storage format

The sparsity pattern of **A** may also be stored as an adjacency graph. For each column of **A**, a list of indices of rows of the whole of **A** (that is, both triangles) that correspond to nonzero entries are recorded; by convention for column j, if row j occurs, it is omitted. Two integer arrays `IND` and `PTR` are used, and the row indices of column j are stored as `IND(l)`, `l` $=$ `PTR`$(j), \ldots,$`PTR` $(j+1)-1$. for $j = 1, \ldots, n$.

2.4 Real and integer kinds

We use the terms integer and real to refer to the fortran keywords `REAL(rp_)` and `INTEGER(ip_)`, where `rp_` and `ip_` are the relevant kind values for the real and integer types employed by the particular module in use. The former are equivalent to default `REAL` for the single precision versions, `DOUBLE PRECISION` for the double precision cases and quadruple-precision if 128-bit reals are available, and correspond to `rp_ = real32`, `rp_ = real64` and `rp_ = real128` respectively as defined by the fortran `iso_fortran_env` module. The latter are default (32-bit) and long (64-bit) integers, and correspond to `ip_ = int32` and `ip_ = int64`, respectively, again from the `iso_fortran_env` module.

2.5 The derived data types

Three derived data types are used by the package.

2.5.1 The derived data type for holding the matrix

The derived data type `SMT_type` is used to hold the matrix **A**. The components of `SMT_type` used are:

`n` is a scalar variable of type `INTEGER(ip_)`, that holds the order n of the matrix **A**. **Restriction:** `n` $\geq$ `1`.

`type` is an allocatable array of rank one and type default `CHARACTER`, that indicates the storage scheme used. If the sparse co-ordinate scheme (see §2.2.1) is used the first ten components of `type` must contain the string `COORDINATE`. For the sparse row-wise storage scheme (see §2.2.2), the first fourteen components of `type` must contain the string `SPARSE_BY_ROWS`, and for dense storage scheme (see §2.2.3) the first five components of `type` must contain the string `DENSE`.

For convenience, the procedure `SMT_put` may be used to allocate sufficient space and insert the required keyword into `type`. For example, if **A** is to be stored in the structure `A` of derived type `SMT_type` and we wish to use the co-ordinate scheme, we may simply

```
CALL SMT_put( A%type, 'COORDINATE', istat )
```

See the documentation for the GALAHAD package SMT for further details on the use of SMT_put.

ne is a scalar variable of type INTEGER(ip_), that holds the number of entries in the **lower triangular** part of **A** in the sparse co-ordinate storage scheme (see §2.2.1). It need not be set for any of the other schemes.

row is a rank-one allocatable array of type INTEGER(ip_), that holds the row indices of the **lower triangular** part of **A** in the sparse co-ordinate storage scheme (see §2.2.1). It need not be allocated for any of the other schemes. Any entry whose row index lies out of the range [1,n] will be ignored.

col is a rank-one allocatable array variable of type INTEGER(ip_), that holds the column indices of the **lower triangular** part of **A** in either the sparse co-ordinate (see §2.2.1), or the sparse row-wise (see §2.2.2) storage scheme. It need not be allocated when the dense storage scheme is used. Any entry whose column index lies out of the range [1,n] will be ignored, while the row and column indices of any entry from the **strict upper triangle** will implicitly be swapped.

ptr is a rank-one allocatable array of size n+1 and type INTEGER(ip_), that holds the starting position of each row of the **lower triangular** part of **A**, as well as the total number of entries plus one, in the sparse row-wise storage scheme (see §2.2.2). It need not be allocated for the other schemes.

The derived type also has a val component that may hold real values that are not used here, but can be by other applications that share a SMT_type variable.

2.5.2 The derived data type for holding control parameters

The derived data type NODEND_control_type is used to hold controlling data. Values specifically for the desired solver may be changed at run time by calling NODEND_read_specfile (see §2.8.1). The components of NODEND_control_type are:

version is a scalar variable of type default CHARACTER and length 30, that specifies the desired version of METIS. Possible values are '4.0', '5.1' and '5.2'. The default is version = '5.2'.

error is a scalar variable of type INTEGER(ip_), that holds the unit number for error messages. Printing of error messages is suppressed if error $<$ 0. The default is error = 6.

out is a scalar variable of type INTEGER(ip_), that holds the unit number for informational messages. Printing of informational messages is suppressed if out $<$ 0. The default is out = 6.

print_level is a scalar variable of type INTEGER(ip_), that is used to control the amount of informational output that is required. No informational output will occur if print_level $\leq$ 0. If print_level $\geq$ 1 details of the ordering process will be produced. The default is print_level = 0.

metis4_ptype is a scalar variable of type INTEGER(ip_), that specifies the partitioning method employed. 0 = multilevel recursive bisectioning: 1 = multilevel k-way partitioning The default is metis4_ptype = 0, and any invalid value will be replaced by this default.

metis4_ctype is a scalar variable of type INTEGER(ip_), that specifies the matching scheme to be used during coarsening: 1 = random matching, 2 = heavy-edge matching, 3 = sorted heavy-edge matching, and 4 = k-way sorted heavy-edge matching. The default is metis4_ctype = 3, and any invalid value will be replaced by this default.

metis4_itype is a scalar variable of type INTEGER(ip_), that specifies the algorithm used during initial partitioning: 1 = edge-based region growing and 2 = node-based region growing. The default is metis4_itype = 1, and any invalid value will be replaced by this default.

metis4_rtype is a scalar variable of type INTEGER(ip_), that specifies the algorithm used for refinement: 1 = two-sided node Fiduccia-Mattheyses (FM) refinement, and 2 = one-sided node FM refinement. The default is metis4_rtype = 1, and any invalid value will be replaced by this default.

metis4_dbglvl is a scalar variable of type INTEGER(ip_), that specifies the amount of progress/debugging information printed: 0 = nothing, 1 = timings, and $>$ 1 increasingly more. The default is metis4_dbglvl = 0, and any invalid value will be replaced by this default.

metis4_oflags is a scalar variable of type INTEGER(ip_), that specifies select whether or not to compress the graph, and to order connected components separately: 0 = do neither, 1 = try to compress the graph, 2 = order each connected component separately, and 3 = do both. The default is metis4_oflags = 1, and any invalid value will be replaced by this default.

metis4_pfactor is a scalar variable of type INTEGER(ip_), that specifies the minimum degree of the vertices that will be ordered last. More specifically, any vertices with a degree greater than 0.1 metis4_pfactor times the average degree are removed from the graph, an ordering of the rest of the vertices is computed, and an overall ordering is computed by ordering the removed vertices at the end of the overall ordering. Any value smaller than 1 means that no vertices will be ordered last. The default is metis4_pfactor =-1 .

metis4_nseps is a scalar variable of type INTEGER(ip_), that specifies the number of different separators that the algorithm will compute at each level of nested dissection. The default is metis4_nseps = 1, and any smaller value will be replaced by this default.

metis5_ptype is a scalar variable of type INTEGER(ip_), that specifies the partitioning method. The value 0 gives multilevel recursive bisectioning, while 1 corresponds to multilevel k-way partitioning. The default is metis5_ptype = 0, and any invalid value will be replaced by this default.

metis5_objtype is a scalar variable of type INTEGER(ip_), that specifies the type of the objective. Currently the only and default value metis5_objtype = 2, specifies node-based nested dissection, and any invalid value will be replaced by this default.

metis5_ctype is a scalar variable of type INTEGER(ip_), that specifies the matching scheme to be used during coarsening: 0 = random matching, and 1 = sorted heavy-edge matching. The default is metis5_ctype = 1, and any invalid value will be replaced by this default.

metis5_iptype is a scalar variable of type INTEGER(ip_), that specifies the algorithm used during initial partitioning: 2 = derive separators from edge cuts, and 3 = grow bisections using a greedy node-based strategy. The default is metis5_iptype = 2, and any invalid value will be replaced by this default.

metis5_rtype is a scalar variable of type INTEGER(ip_), that specifies the algorithm used for refinement: 2 = Two-sided node FM refinement, and 3 = One-sided node FM refinement. The default is metis5_rtype = 2, and any invalid value will be replaced by this default.

metis5_dbglvl is a scalar variable of type INTEGER(ip_), that specifies the amount of progress/debugging information printed: 0 = nothing, 1 = diagnostics, 2 = plus timings, and $>$ 2 plus more. The default is metis5_dbglvl = 0, and any invalid value will be replaced by this default.

metis5_niparts is a scalar variable of type INTEGER(ip_), that specifies the number of initial partitions used by MeTiS 5.2. The default is metis5_niparts = -1, and any invalid value will be replaced by this default.

metis5_niter is a scalar variable of type INTEGER(ip_), that specifies the number of iterations used by the refinement algorithm. The default is metis5_niter = 10, and any non-positive value will be replaced by this default.

metis5_ncuts is a scalar variable of type INTEGER(ip_), that specifies the number of different partitionings that it will compute: -1 = not used. The default is metis5_ncuts = -1, and any invalid value will be replaced by this default.

metis5_seed is a scalar variable of type INTEGER(ip_), that specifies the seed for the random number generator. The default is metis5_seed = -1.

metis5_ondisk is a scalar variable of type INTEGER(ip_), that specifies whether on-disk storage is used (0 = no, 1 = yes) by MeTiS 5.2. The default is metis5_ondisk = 0, and any invalid value will be replaced by this default.

metis5_minconn is a scalar variable of type INTEGER(ip_), that specifies specify that the partitioning routines should try to minimize the maximum degree of the subdomain graph: 0 = no, 1 = yes, and -1 = not used. The default is metis5_minconn =-1 , and any invalid value will be replaced by this default.

metis5_contig is a scalar variable of type INTEGER(ip_), that specifies specify that the partitioning routines should try to produce partitions that are contiguous: 0 = no, 1 = yes, and -1 = not used. The default is metis5_contig = 1, and any invalid value will be replaced by this default.

metis5_compress is a scalar variable of type INTEGER(ip_), that specifies specify that the graph should be compressed by combining together vertices that have identical adjacency lists: 0 = no, and 1 = yes. The default is metis5_compress = 1, and any invalid value will be replaced by this default.

metis5_ccorder is a scalar variable of type INTEGER(ip_), that specifies specify if the connected components of the graph should first be identified and ordered separately: 0 = no, and 1 = yes. The default is metis5_ccorder = 0, and any invalid value will be replaced by this default.

metis5_pfactor is a scalar variable of type INTEGER(ip_), that specifies the minimum degree of the vertices that will be ordered last. More specifically, any vertices with a degree greater than 0.1 metis4_pfactor times the average degree are removed from the graph, an ordering of the rest of the vertices is computed, and an overall ordering is computed by ordering the removed vertices at the end of the overall ordering. The default is metis5_pfactor = 0, and any negative value will be replaced by this default.

metis5_nseps is a scalar variable of type INTEGER(ip_), that specifies the number of different separators that the algorithm will compute at each level of nested dissection. The default is metis5_nseps = 1, and any non-positive value will be replaced by this default.

metis5_ufactor is a scalar variable of type INTEGER(ip_), that specifies the maximum allowed load imbalance (1 + metis5_ufactor)/1000 among the partitions. The default is metis5_ufactor = 200, and any negative value will be replaced by this default.

metis5_dropedges is a scalar variable of type INTEGER(ip_), that specifies whether edges will be dropped (0 = no, 1 = yes) by MeTiS 5.2. The default is metis5_dropedges = 0, and any invalid value will be replaced by this default.

metis5_no2hop is a scalar variable of type INTEGER(ip_), that specifies specify that the coarsening will not perform any 2–hop matchings when the standard matching approach fails to sufficiently coarsen the graph: 0 = no, and 1 = yes The default is metis5_no2hop = 0, and any invalid value will be replaced by this default.

metis5_twohop is a scalar variable of type INTEGER(ip_), that is reserved for future use but ignored at present. The default is metis5_twohop = -1.

metis5_fast is a scalar variable of type INTEGER(ip_), that is reserved for future use but ignored at present. The default is metis5_fast = -1. replaced by this default.

`prefix` is a scalar variable of type default `CHARACTER` and length 30, that may be used to provide a user-selected character string to preface every line of printed output. Specifically, each line of output will be prefaced by the string `prefix(2:LEN(TRIM(prefix))-1)`, thus ignoring the first and last non-null components of the supplied string. If the user does not want to preface lines by such a string, the default `prefix = ""` should be used.

`prefix` is a scalar variable of type default `CHARACTER` and length 30, that may be used to provide a user-selected character string to preface every line of printed output. Specifically, each line of output will be prefaced by the string `prefix(2:LEN(TRIM(prefix))-1)`, thus ignoring the first and last non-null components of the supplied string. If the user does not want to preface lines by such a string, the default `prefix = ""` should be used.

2.5.3 The derived data type for holding informational parameters

The derived data type `NODEND_inform_type` is used to hold parameters that give information about the progress and needs of the algorithm. The components of `NODEND_inform_type` are as follows—any component that is not relevant to the solver being used will have the value `-1` or `-1.0` as appropriate:

`status` is a scalar variable of type `INTEGER(ip_)`, that gives the exit status of the algorithm. See §2.7 for details.

`alloc_status` is a scalar variable of type `INTEGER(ip_)`, that gives the status of the last attempted array allocation or deallocation.

`bad_alloc` is a scalar variable of type default `CHARACTER` and length 80, that gives the name of the last internal array for which there were allocation or deallocation errors. This will be the null string if there have been no allocation or deallocation errors.

2.5.4 The derived data type for holding problem data

The derived data type `NODEND_data_type` is used to hold all the data for a particular problem, or sequences of problems with the same structure, between calls to `NODEND` procedures. All components are private.

2.6 Argument lists and calling sequences

2.6.1 The basic ordering subroutine

A nested-dissection-based ordering of the sparsity pattern of **A** may be obtained as follows:

```
CALL NODEND_order( A, PERM, control, inform )
```

`A` is scalar `INTENT(IN)` argument of type `SMT_type` that is used to specify **A**. The user must set all of the relevant components of `matrix` according to the storage scheme desired (see §2.5.1. Incorrectly-set components will result in errors flagged in `inform%status`, see §2.7.

`perm` is an

`PERM` is a rank-one `INTEGER(ip_) INTENT(OUT)` array argument of `INTENT(OUT)` and length $A\%n$. `PERM` will be set to the permutation array, so that the `PERM(i)`-th rows and columns in the permuted matrix $\mathbf{PAP}^T$ correspond to those labelled `i` in **A**.

`control` is a scalar `INTENT(OUT)` argument of type `NODEND_control_type`. Its components control the action of the analysis phase, as explained in §2.5.2.

`inform` is a scalar `INTENT(OUT)` argument of type `NODEND_inform_type` (see §2.5.3). A successful call is indicated when the component `status` has the value 0. For other return values of `status`, see §2.7.

2.6.2 The graph ordering subroutine

A nested-dissection-based ordering of the adjacency graph (see §2.3) of **A** may be obtained as follows:

```
CALL NODEND_order_adjacency( n, PTR, IND, PERM, control, inform )
```

`n` is an `INTENT(IN)` scalar of type `INTEGER` that gives the number of rows (and columns) of **A**.

`PTR` is a rank-one `INTEGER(ip_)` array argument of `INTENT(IN)` and length at least $n+1$. Its j entry, `PTR`(j), must be set to the position in `IND` of the first entry in column j of the whole of **A**, while `PTR`$(n+1)$ points to the first unoccupied position in `IND`.

`IND` is a rank-one `INTEGER(ip_)` array argument of `INTENT(IN)` and length at least `PTR`$(n+1)-1$. Components `IND`$((l))$, $l =$ `PTR`$(j), \ldots,$`PTR` $(j+1)-1$ must hold the row indices of non-diagonal entries in column j of **A**.

`PERM` is a rank-one `INTEGER(ip_) INTENT(OUT)` array argument of `INTENT(OUT)` and length `n`. `PERM` will be set to the permutation array, so that the `PERM(i)`-th rows and columns in the permuted matrix $\mathbf{PAP}^T$ correspond to those labelled `i` in **A**.

`control` is a scalar `INTENT(OUT)` argument of type `NODEND_control_type`. Its components control the action of the analysis phase, as explained in §2.5.2.

`inform` is a scalar `INTENT(OUT)` argument of type `NODEND_inform_type` (see §2.5.3). A successful call is indicated when the component `status` has the value 0. For other return values of `status`, see §2.7.

2.7 Warning and error messages

A negative value of `inform%status` on exit from the subroutines indicates that an error has occurred. No further calls should be made until the error has been corrected. Possible values are:

-1 An allocation error occurred. A message indicating the offending array is written on unit `control%error`, and the returned allocation status and a string containing the name of the offending array are held in `inform%alloc_status` and `inform%bad_alloc` respectively.

-2 A deallocation error occurred. A message indicating the offending array is written on unit `control%error` and the returned allocation status and a string containing the name of the offending array are held in `inform%alloc_status` and `inform%bad_alloc` respectively.

-3 One of the restrictions `n` > 0, `A%n` > 0 or `A%ne` < 0, for co-ordinate entry, or requirements that `A%type` contain its relevant string `'COORDINATE'`, `'SPARSE_BY_ROWS'` or `'DENSE'`, and `control%version` in one of `'4.0'`, `'5.1'` or `'5.2'` has been violated.

-26 The requested version of `METIS` is not available.

-57 `METIS` has insufficient memory to continue.

-71 An internal `METIS` error occurred.

2.8 Setting control parameters

In this section, we describe an alternative means of setting control parameters, that is components of the variable `control` of type `NODEND_control_type` (see §2.5.2), by reading an appropriate data specification file using the subroutine `NODEND_read_specfile`. This facility is useful as it allows a user to change `NODEND` control parameters without editing and recompiling programs that call `NODEND`.

A specification file, or specfile, is a data file containing a number of "specification commands". Each command occurs on a separate line, and comprises a "keyword", that is a string (in a close-to-natural language) used to identify a control parameter, and an (optional) "value", which defines the value to be assigned to the given control parameter. All keywords and values are case insensitive, keywords may be preceded by one or more blanks but values must not contain blanks, and each value must be separated from its keyword by at least one blank. Values must not contain more than 30 characters, and each line of the specification file is limited to 80 characters, including the blanks separating keyword and value.

The portion of the specification file used by `NODEND_read_specfile` must start with a "`BEGIN NODEND`" command and end with an "`END`" command. The syntax of the specfile is thus defined as follows:

```
( .. lines ignored by METIS_read_specfile .. )
  BEGIN METIS
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by METIS_read_specfile .. )
```

where keyword and value are two strings separated by (at least) one blank. The "`BEGIN NODEND`" and "`END`" delimiter command lines may contain additional (trailing) strings so long as such strings are separated by one or more blanks, so that lines such as

```
BEGIN METIS SPECIFICATION
```

and

```
END METIS SPECIFICATION
```

are acceptable. Furthermore, between the "`BEGIN NODEND`" and "`END`" delimiters, specification commands may occur in any order. Blank lines and lines whose first non-blank character is `!` or `*` are ignored. The content of a line after a `!` or `*` character is also ignored (as is the `!` or `*` character itself). This provides an easy way to "comment out" some specification commands, or to comment specific values of certain control parameters.

The value of a control parameter may be of three different types, namely integer, character or real. Integer and real values may be expressed in any relevant Fortran integer and floating-point formats (respectively).

The specification file must be open for input when `NODEND_read_specfile` is called, and the associated unit number passed to the routine in `device` (see below). Note that the corresponding file is rewound, which makes it possible to combine the specifications for more than one program/routine. For the same reason, the file is not closed by `NODEND_read_specfile`.

2.8.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL METIS_read_specfile( control, device )
```

`control` is a scalar `INTENT(INOUT)` argument of type `NODEND_control_type` (see §2.5.2). Default values should have already been set, perhaps by calling `NODEND_initialize`. On exit, individual components of `control` may have been changed according to the commands found in the specfile. Specfile commands and the component (see §2.5.2) of `control` that each affects are given in Table 2.1.

`device` is a scalar `INTENT(IN)` argument of type `INTEGER(ip_)`, that must be set to the unit number on which the specification file has been opened. If `device` is not open, `control` will not be altered and execution will continue, but an error message will be printed on unit `control%error`.

command	component of control	value type
version	%version	character
error-printout-device	%error	integer
printout-device	%out	integer
print-level	%print_level	integer
metis4-ptype	%metis4_ptype	integer
metis4-ctype	%metis4_ctype	integer
metis4-itype	%metis4_itype	integer
metis4-rtype	%metis4_rtype	integer
metis4-dbglvl	%metis4_dbglvl	integer
metis4-oflags	%metis4_oflags	integer
metis4-pfactor	%metis4_pfactor	integer
metis4-nseps	%metis4_nseps	integer
metis5-ptype	%metis5_ptype	integer
metis5-objtype	%metis5_objtype	integer
metis5-ctype	%metis5_ctype	integer
metis5-iptype	%metis5_iptype	integer
metis5-rtype	%metis5_rtype	integer
metis5-dbglvl	%metis5_dbglvl	integer
metis5-niparts	%metis5_niparts	integer
metis5-niter	%metis5_niter	integer
metis5-ncuts	%metis5_ncuts	integer
metis5-seed	%metis5_seed	integer
metis5-ondisk	%metis5_ondisk	integer
metis5-minconn	%metis5_minconn	integer
metis5-contig	%metis5_contig	integer
metis5-compress	%metis5_compress	integer
metis5-ccorder	%metis5_ccorder	integer
metis5-pfactor	%metis5_pfactor	integer
metis5-nseps	%metis5_nseps	integer
metis5-ufactor	%metis5_ufactor	integer
metis5-dropedges	%metis5_dropedges	integer
metis5-no2hop	%metis5_no2hop	integer
metis5-twohop	%metis5_twohop	integer
metis5-fast	%metis5_fast	integer
output-line-prefix	%prefix	character

Table 2.1: Specfile commands and associated components of control.

3 GENERAL INFORMATION

Workspace: Provided automatically by the module.

Other modules used directly: `GALAHAD_CLOCK`, `GALAHAD_KINDS`, `GALAHAD_SYMBOLS`, `GALAHAD_SORT_single/double`, `GALAHAD_SMT_single/double` and `GALAHAD_SPECFILE_single/double`,

Input/output: Output is under control of the arguments `control%error`, `control%out`

Restrictions: `n` $\geq$ 1, `A%n` $\geq$ 1, `A%ne` $\geq$ 0 if `A%type = 'COORDINATE'`, `A%type` one of `'COORDINATE'`, `'SPARSE_BY_ROWS'` or `'DENSE'`. `control%version` one of `'4.0'`, `'5.1'` or `'5.2'`.

Portability: ISO Fortran 95 + TR 15581 or Fortran 2003. The package is thread-safe.

4 METHOD

Variants of node-based nested-dissection ordering are used.

The package relies crucially on the ordering package `METIS` from the Karypis Lab. To obtain `METIS` 4.0, see

`https://github.com/KarypisLab`.

or

`https://github.com/CIBC-Internal/metis-4.0.3`

Versions 5.1 and 5.2 are open-source software, and included.

References:

The methods used are described in the user-documentation

G. Karypis. METIS, A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices, Version 5, Department of Computer Science & Engineering, University of Minnesota Minneapolis, MN 55455, USA (2013), see

`https://github.com/KarypisLab/METIS/blob/master/manual/manual.pdf`

and paper

G. Karypis and V. Kumar (1999). A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM Journal on Scientific Computing. **20(1)** (1999) 359–392.

5 EXAMPLE OF USE

We illustrate the use of the package on the symmetric matrix with structure

$$\begin{pmatrix} * & & * & & * \\ & * & & & \\ * & & * & & \\ & & & * & * \\ * & & & * & * \end{pmatrix},$$

where $*$ denotes a nonzero. Then, we may use the following code to find a suitable nested-dissection permutation prior to Cholesky-like factorization.

```
! THIS VERSION: GALAHAD 5.2 - 2025-03-11 AT 09:00 GMT.
   PROGRAM NODEND_example
   USE GALAHAD_KINDS_double, ONLY: ip_
```

```
!    USE GALAHAD_SMT_double
     USE GALAHAD_NODEND_double
     INTEGER, PARAMETER :: out = 6
     INTEGER ( KIND = ip_ ), PARAMETER :: n = 5, ne = 8
     INTEGER ( KIND = ip_ ), DIMENSION( n ) :: PERM
     TYPE ( SMT_type ) :: A
     TYPE ( NODEND_control_type ) :: control
     TYPE ( NODEND_inform_type ) :: inform
     INTEGER :: smt_stat
     CALL SMT_put( A%type, 'COORDINATE', smt_stat )
     A%n = n ; A%ne = ne
     ALLOCATE( A%row( ne ), A%col( ne ) )
     A%row = (/ 1, 2, 3, 3, 4, 5, 5, 5 /)
     A%col = (/ 1, 2, 1, 3, 4, 1, 4, 5 /)
     control%version = '5.1'
     CALL NODEND_order( A, PERM, control, inform )
     IF ( PERM( 1 ) <= 0 ) THEN
       WRITE( out, "( ' No METIS ', A, ' available, stopping' )" )            &
         control%version
     ELSE IF ( inform%status < 0 ) THEN
       WRITE( out, "( ' Nodend ', A, ' failure, status = ', I0 )" )            &
         control%version, inform%status
     ELSE
       IF ( inform%status == 0 ) THEN
         WRITE( out, "( ' Nodend ', A, ' order call successful' )" )           &
           TRIM( control%version )
         WRITE( out, "( ' permutation =', 5I2 )" ) PERM
       ELSE
         WRITE( out, "(  ' Nodend ', A, ' order call unsuccessful,',           &
        &  ' no permutation found' )" ) TRIM( control%version )
       END IF
     END IF
     DEALLOCATE( A%row, A%col, A%type )
     END PROGRAM NODEND_example
```

This produces the following output:

```
 Nodend 5.1 order call successful
 permutation = 2 4 1 3 5
```

Alternatively, we may use the adjacency graph format to produce the same ordering.

```
! THIS VERSION: GALAHAD 5.2 - 2025-03-11 AT 09:00 GMT.
     PROGRAM NODEND_example_adjacency
     USE GALAHAD_KINDS_double, ONLY: ip_
     USE GALAHAD_NODEND_double
     INTEGER, PARAMETER :: out = 6
     INTEGER ( KIND = ip_ ), PARAMETER :: n = 5, nz = 6
     INTEGER ( KIND = ip_ ), DIMENSION( n + 1 ) :: PTR = (/ 1, 3, 3, 4, 5, 7 /)
     INTEGER ( KIND = ip_ ), DIMENSION( nz ) :: IND = (/ 3, 5, 1, 5, 1, 4 /)
     INTEGER ( KIND = ip_ ), DIMENSION( n ) :: PERM
     TYPE ( NODEND_control_type ) :: control
     TYPE ( NODEND_inform_type ) :: inform
     control%version = '5.1'
     CALL NODEND_order_adjacency( n, PTR, IND, PERM, control, inform )
     IF ( PERM( 1 ) <= 0 ) THEN
       WRITE( out, "( ' No METIS ', A, ' available, stopping' )" )            &
```

```
      control%version
    ELSE IF ( inform%status < 0 ) THEN
      WRITE( out, "( ' Nodend ', A, ' failure, status = ', I0 )" )            &
        control%version, inform%status
    ELSE
      IF ( inform%status == 0 ) THEN
        WRITE( out, "( ' Nodend ', A, ' order_adjacency call successful' )" ) &
          TRIM( control%version )
        WRITE( out, "( ' permutation =', 5I2 )" ) PERM
      ELSE
        WRITE( out, "(  ' Nodend ', A, ' order call unsuccessful,',           &
       &  ' no permutation found' )" ) TRIM( control%version )
      END IF
    END IF
    END PROGRAM NODEND_example_adjacency
```

This produces the following output:

```
 Nodend 5.1 order_adjacency call successful
 permutation = 2 4 1 3 5
```