GALAHAD CONVERT

USER DOCUMENTATION GALAHAD Optimization Library version 3.3

1 SUMMARY

This package takes a real matrix stored according to one of a number of commonly-occurring formats and converts it, or its transpose, to another specified format.

ATTRIBUTES — Versions: `GALAHAD_CONVERT_single`, `GALAHAD_CONVERT_double`. **Uses:** `GALAHAD_CPU_time`, `GALAHAD_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_SPECFILE`. **Date:** November 2020. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003.

2 HOW TO USE THE PACKAGE

Access to the package requires a `USE` statement such as

Single precision version

```
USE GALAHAD_CONVERT_single
```

Double precision version

```
USE GALAHAD_CONVERT_double
```

If it is required to use both modules at the same time, the derived types `SMT_type`, `CONVERT_time_type`, `CONVERT_control_type`, `CONVERT_inform_type` and `CONVERT_data_type` (Section 2.2) and the subroutines `CONVERT_between_matrix_formats` (Section 2.3) and `CONVERT_read_specfile` (Section 2.5) must be renamed on one of the `USE` statements.

2.1 Matrix storage formats

Both the input and output matrices $\mathbf{A}$ or its transpose will be stored in a variety of common formats.

2.1.1 Dense row-wise storage format

The matrix $\mathbf{A}$ 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. Component $n*(i-1)+j$ of the storage array `A%val` will hold the value $\mathbf{A}_{i,j}$ for $i = 1,\ldots,m$, $j = 1,\ldots,n$.

2.1.2 Dense column-wise storage format

The matrix $\mathbf{A}$ is stored as a compact dense matrix by columns, that is, the values of the entries of each column in turn are stored in order within an appropriate real one-dimensional array. Component $m*(j-1)+i$ of the storage array `A%val` will hold the value $\mathbf{A}_{i,j}$ for $i = 1,\ldots,m$, $j = 1,\ldots,n$.

2.1.3 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l-th entry of $\mathbf{A}$, its row index i, column index j and value $\mathbf{A}_{ij}$ are stored in the l-th components of the integer arrays `A%row`, `A%col` and real array `A%val`. The order is unimportant, but the total number of entries `A%ne` is required.

2.1.4 Sparse row-wise storage format

Again only the nonzero entries 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 $\mathbf{A}$, the i-th component of the integer array A%ptr holds the position of the first entry in this row, while A%ptr $(m+1)$ holds the total number of entries plus one. The column indices j and values $\mathbf{A}_{ij}$ of the entries in the i-th row are stored in components $l =$ A%ptr(i), ...,A%ptr $(i+1)-1$ of the integer array A%col, and real array A%val, respectively.

2.1.5 Sparse column-wise storage format

Yet again only the nonzero entries are stored, but this time they are ordered so that those in column j appear directly before those in column $j+1$. For the j-th column of $\mathbf{A}$, the j-th component of the integer array A%ptr holds the position of the first entry in this column, while A%ptr $(n+1)$ holds the total number of entries plus one. The row indices i and values $\mathbf{A}_{ij}$ of the entries in the j-th column are stored in components $l =$ A%ptr(j), ...,A%ptr $(j+1)-1$ of the integer array A%row, and real array A%val, respectively.

For sparse matrices, the row- and column-wise storage schemes almost always requires less storage than their predecessor.

2.2 The derived data types

Four derived data types are accessible from the package.

2.2.1 The derived data type for holding matrices

The derived data type SMT_TYPE is used to hold the matrix $\mathbf{A}$. The components of SMT_TYPE used here are:

m is a scalar component of type default INTEGER, that holds the number of rows in the matrix.

n is a scalar component of type default INTEGER, that holds the number of columns in the matrix.

ne is a scalar variable of type default INTEGER, that holds the number of matrix entries.

type is a rank-one allocatable array of type default CHARACTER, that is used to indicate the matrix storage scheme used. Its precise length and content depends on the type of matrix to be stored (see §2.2.1).

If the dense row-wise storage scheme (see Section 2.1.1) is used, the first thirteen components of type must contain the string DENSE_BY_ROW, while if the column-wise scheme (see Section 2.1.2) is used, the sixteen components of type must contain the string DENSE_BY_COLUMN. For the sparse co-ordinate scheme (see Section 2.1.3), the first ten components of type must contain the string COORDINATE, for the sparse row-wise storage scheme (see Section 2.1.4), the first fourteen components of type must contain the string SPARSE_BY_ROWS. and for the sparse column-wise storage scheme (see Section 2.1.5), the first seventeen components of type must contain the string SPARSE_BY_COLUMNS.

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 of derived type SMT_TYPE that we wish to store using the co-ordinate scheme, we may simply

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

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

val is a rank-one allocatable array of type default REAL (double precision in GALAHAD_CONVERT_double) and dimension at least ne, that holds the values of the entries. Any duplicated entries that appear in the sparse co-ordinate or row-wise schemes will be summed.

row is a rank-one allocatable array of type default INTEGER, and dimension at least ne, that may hold the row indices of the entries. (see §2.1.3 and §2.1.5).

col is a rank-one allocatable array of type default INTEGER, and dimension at least ne, that may the column indices of the entries (see §2.1.3–2.1.4).

ptr is a rank-one allocatable array of type default INTEGER, and dimension at least m + 1, that may hold the pointers to the first entry in each row (see §2.1.4). or dimension at least n + 1, that may hold the pointers to the first entry in each column (see §2.1.5).

2.2.2 The derived data type for holding control parameters

The derived data type CONVERT_control_type is used to hold controlling data. Components may be changed by calling CONVERT_read_specfile (see Section 2.5.1). The components of CONVERT_control_type are:

error is a scalar variable of type default INTEGER, that holds the stream number for error messages. Printing of error messages in CONVERT_between_matrix_formats is suppressed if error ≤ 0. The default is error = 6.

out is a scalar variable of type default INTEGER, that holds the stream number for informational messages. Printing of informational messages in CONVERT_between_matrix_formats is suppressed if out < 0. The default is out = 6.

print_level is a scalar variable of type default INTEGER, that is used to control the amount of informational output which is required. No informational output will occur if print_level ≤ 0. If print_level $= 1$, a single line of output will be produced for each iteration of the process. If print_level ≥ 2, this output will be increased to provide significant detail of each iteration. The default is print_level = 0.

transpose is a scalar variable of type default LOGICAL, that must be set .TRUE. if the transpose of the matrix is to be stored on output, and .FALSE. otherwise. The default is transpose = .FALSE..

sum_duplicates is a scalar variable of type default LOGICAL, that must be set .TRUE. if any repeated entries of the matrix input in co-ordinate format are to be summed on output, and .FALSE. otherwise. The default is sum_duplicates = .FALSE..

order is a scalar variable of type default LOGICAL, that must be set .TRUE. if the rows or columns of the output matrix in sparse row- or column-format are to be arranged in order of increasing indices, and .FALSE. otherwise. The default is order = .FALSE..

space_critical is a scalar variable of type default LOGICAL, that must be set .TRUE. if space is critical when allocating arrays and .FALSE. otherwise. The package may run faster if space_critical is .FALSE. but at the possible expense of a larger storage requirement. The default is space_critical = .FALSE..

deallocate_error_fatal is a scalar variable of type default LOGICAL, that must be set .TRUE. if the user wishes to terminate execution if a deallocation fails, and .FALSE. if an attempt to continue will be made. The default is deallocate_error_fatal = .FALSE..

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, they may use the default prefix = "".

2.2.3 The derived data type for holding timing information

The derived data type CONVERT_time_type is used to hold elapsed CPU and clock times for the calculation. The components of CONVERT_time_type are:

total is a scalar variable of type default REAL, that gives the total CPU time spent in the package.

clock_total is a scalar variable of type default REAL, that gives the total clock time spent in the package.

2.2.4 The derived data type for holding informational parameters

The derived data type CONVERT_inform_type is used to hold parameters that give information about the progress and needs of the algorithm. The components of CONVERT_inform_type are:

status is a scalar variable of type default INTEGER, that gives the exit status of the algorithm. See Section 2.4 for details.

alloc_status is a scalar variable of type default INTEGER, that gives the status of the last attempted array allocation or deallocation. This will be 0 if status = 0.

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 status = 0.

duplicates is a scalar variable of type default INTEGER, that gives the number of repeated matrix entries encountered when converting the input matrix.

2.3 Argument lists and calling sequences

There is one primary procedure for user calls (see Section 2.5 for further features):

1. The subroutine CONVERT_between_matrix_formats is called to reformat the input matrix.

2.3.1 The conversion between formats subroutine

The conversion algorithm is called as follows:

```
CALL CONVERT_between_matrix_formats( A, output_format, A_out, control, inform )
```

A is a scalar INTENT(IN) argument of type SMT_type (see Section 2.2.1). It is used to hold the input matrix **A**. The user must allocate and set all the components that are relevant for the storage scheme used, see Section 2.1.

For all schemes, the components A%m and A%n must be set to the numbers of rows and columns of **A**. The remaining necessary components according to storage type are:

Dense row-wise storage format, see §2.1.1. The array A%type should be set so that A%type(1 : 13) = TRANSFER('DENSE_BY_ROWS', A%type) or A%type(1 : 5) = TRANSFER('DENSE', A%type). The array A%val(1:A%m*A%n) should be filed with the components of **A**, stored row by row, i.e., A%val(A%n*(i-1)+j) = $\mathbf{A}_{i,j}$.

Dense column-wise storage format, see §2.1.2. The array A%type should be set so that A%type(1 : 16) = TRANSFER('DENSE_BY_COLUMNS', A%type). The array A%val(1:A%m*A%n) should be filed with the components of **A**, stored column by column, i.e., A%val(A%m*(j-1)+i) = $\mathbf{A}_{i,j}$.

Sparse co-ordinate storage format, see §2.1.3. The array A%type should be set so that A%type(1 : 10) = TRANSFER('COORDINATE', A%type). The integer A%ne should specify the number of entries in **A**, and

the arrays A%row(1:A%ne), A%col(1:A%ne) and A%val(1:A%ne) should be filled by the row indices, column indices and values of the entries, respectively.

Sparse row-wise storage format, see §2.1.4. The array A%type should be set so that A%type(1 : 14) = TRANSFER('SPARSE_BY_ROWS', A%type). Component A%ptr(i) should give the starting address for the entries in row i of **A** for i = 1,...A%m, with A%ptr(A%m+1) giving the total number of entries in **A** plus one; the column indices and values in row i can appear in any order in components A%col(l) and A%val(l) for l = A%ptr(i), ...A%ptr(i+1)-1 respectively.

Sparse column-wise storage format, see §2.1.5. The array A%type should be set so that A%type(1 : 17) = TRANSFER('SPARSE_BY_COLUMNS', A%type). Component A%ptr(j) should give the starting address for the entries in column j of **A** for j = 1,...A%n, with A%ptr(A%n+1) giving the total number of entries in **A** plus one; the row indices and values in column j can appear in any order in components A%col(l) and A%val(l) for l = A%ptr(j), ...A%ptr(j+1)-1 respectively.

Restrictions: A%m $>$ 0, A%n $>$ 0 and (if **A** is provided in sparse co-ordinate format) A%ne $\geq$ 0. A%type $\in$ { 'DENSE', 'DENSE_BY_ROWS', 'DENSE_BY_COLUMNS', 'COORDINATE', 'SPARSE_BY_ROWS', 'SPARSE_BY_COLUMNS' }.

output_format is a scalar INTENT(IN) argument of type CHARACTER of appropriate length that is used to define the required output format for the converted **A** or its transpose. **Restrictions:** output_format $\in$ {'DENSE', 'DENSE_BY_ROWS', 'DENSE_BY_COLUMNS', 'COORDINATE', 'SPARSE_BY_ROWS' }.

A_out is a scalar INTENT(OUT) argument of type SMT_type (see Section 2.2.1) that holds the converted matrix **A** or its transpose in the format specified by output_format.

control is a scalar INTENT(IN) argument of type CONVERT_control_type (see Section 2.2.2). Default values will have been provided.

inform is a scalar INTENT(INOUT) argument of type CONVERT_inform_type (see Section 2.2.4). A successful call to CONVERT_between_matrix_formats is indicated when the component status has the value 0. For other return values of status, see Section 2.4.

2.4 Warning and error messages

A negative value of inform%status on exit from CONVERT_between_matrix_formats or CONVERT_terminate 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 A%n $>$ 0, A%m $>$ 0 or the requirement that A_type contain its relevant string 'DENSE', 'DENSE_BY_ROWS', 'DENSE_BY_COLUMNS', 'COORDINATE', 'SPARSE_BY_ROWS' or 'SPARSE_BY_COLUMNS' has been violated.

2.5 Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable control of type CONVERT_control_type (see Section 2.2.2), by reading an appropriate data specification file using the subroutine CONVERT_read_specfile. This facility is useful as it allows a user to change CONVERT control parameters without editing and recompiling programs that call CONVERT.

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", which 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 specfile is limited to 80 characters, including the blanks separating keyword and value.

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

```
( .. lines ignored by BLLS_read_specfile .. )
  BEGIN BLLS
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by BLLS_read_specfile .. )
```

where keyword and value are two strings separated by (at least) one blank. The "BEGIN CONVERT" 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 BLLS SPECIFICATION
```

and

```
END BLLS SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN CONVERT" 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 manner to "comment out" some specification commands, or to comment specific values of certain control parameters.

The value of a control parameters may be of three different types, namely integer, logical or real. Integer and real values may be expressed in any relevant Fortran integer and floating-point formats (respectively). Permitted values for logical parameters are "ON", "TRUE", ".TRUE.", "T", "YES", "Y", or "OFF", "NO", "N", "FALSE", ".FALSE." and "F". Empty values are also allowed for logical control parameters, and are interpreted as "TRUE".

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

2.5.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL BLLS_read_specfile( control, device )
```

control is a scalar INTENT(INOUT) argument of type CONVERT_control_type (see Section 2.2.2). Default values should have already been set, perhaps by calling CONVERT_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 Section 2.2.2) of control that each affects are given in Table 2.1.

command	component of control	value type
error-printout-device	%error	integer
printout-device	%out	integer
print-level	%print_level	integer
transpose	%transpose	logical
sum-duplicates	%sum_duplicates	logical
order	%order	logical
deallocate-error-fatal	%deallocate_error_fatal	logical
output-line-prefix	%prefix	character

Table 2.1: Specfile commands and associated components of control.

device is a scalar INTENT(IN) argument of type default INTEGER, that must be set to the unit number on which the specfile 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.

2.6 Information printed

If control%print_level is positive, any error information will be printed on units control%error or control%out.

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: CONVERT_between_matrix_formats calls the GALAHAD packages GALAHAD_CPU_time, GALAHAD_SYMBOLS, GALAHAD_SPACE and GALAHAD_SPECFILE.

Input/output: Output is under control of the arguments control%error, control%out and control%print_level.

Restrictions: A%m >0, A%n >0, A%ne >0, A_type $\in$ 'DENSE', 'DENSE_BY_ROWS', 'DENSE_BY_COLUMNS', 'COORDINATE', 'SPARSE_BY_ROWS', 'SPARSE_BY_COLUMNS' }

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

4 EXAMPLE OF USE

Suppose we wish to transform the matrix

$$\mathbf{A} = \begin{pmatrix} 11 & 0 & 13 & 0 & 15 \\ 0 & 22 & 0 & 24 & 0 \\ 0 & 32 & 33 & 0 & 0 \\ 0 & 0 & 0 & 44 & 45 \end{pmatrix}$$

from a variety of input formats to sparse-row format Then we may use the following code:

```
! THIS VERSION: GALAHAD 3.3 - 29/10/2020 AT 08:30 GMT.
   PROGRAM GALAHAD_CONVERT_EXAMPLE
   USE GALAHAD_CONVERT_double         ! double precision version
   IMPLICIT NONE
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
   TYPE ( SMT_type ) :: A, A_out
   TYPE ( CONVERT_control_type ) :: control
   TYPE ( CONVERT_inform_type ) :: inform
   INTEGER :: i, j, l, ne, mode, s, type
! set problem data
   A%m = 4 ; A%n = 5 ; ne = 9
   DO mode = 1, 2 ! write natrix (mode=1) or its transpose (mode=2)
     control%order = .TRUE.
     IF ( mode == 2 ) THEN
       WRITE( 6, "( /, ' construct the transpose' )" )
       control%transpose = .TRUE.
     END IF
     DO type = 1, 5 ! loop over storage types
       SELECT CASE ( type )
       CASE ( 1 ) ! dense input format
         CALL SMT_put( A%type, 'DENSE_BY_ROWS', s )
         ALLOCATE( A%val( A%m * A%n ) )
         A%val = (/ 11.0_wp, 0.0_wp, 13.0_wp, 0.0_wp, 15.0_wp,                &
                    0.0_wp, 22.0_wp, 0.0_wp, 24.0_wp, 0.0_wp,                 &
                    0.0_wp, 32.0_wp, 33.0_wp, 0.0_wp, 0.0_wp,                 &
                    0.0_wp, 0.0_wp, 0.0_wp, 44.0_wp, 45.0_wp /)
       CASE ( 2 ) ! dense by columns input format
         CALL SMT_put( A%type, 'DENSE_BY_COLUMNS', s )
         ALLOCATE( A%val( A%m * A%n ) )
         A%val = (/ 11.0_wp, 0.0_wp, 0.0_wp, 0.0_wp,                          &
                    0.0_wp, 22.0_wp, 32.0_wp, 0.0_wp,                         &
                    13.0_wp, 0.0_wp, 33.0_wp, 0.0_wp,                         &
                    0.0_wp, 24.0_wp, 0.0_wp, 44.0_wp,                         &
                    15.0_wp, 0.0_wp, 0.0_wp, 45.0_wp /)
       CASE ( 3 ) ! sparse by rows input format
         CALL SMT_put( A%type, 'SPARSE_BY_ROWS', s )
         ALLOCATE( A%ptr( A%m + 1 ), A%col( ne ), A%val( ne ) )
         A%ptr = (/ 1, 4, 6, 8, 10 /)
         A%col = (/ 1, 5, 3, 2, 4, 3, 2, 4, 5 /)
         A%val = (/ 11.0_wp, 15.0_wp, 13.0_wp, 22.0_wp, 24.0_wp,              &
                    33.0_wp, 32.0_wp, 44.0_wp, 45.0_wp /)
       CASE ( 4 ) ! sparse by columns input format
         CALL SMT_put( A%type, 'SPARSE_BY_COLUMNS', s )
         ALLOCATE( A%ptr( A%n + 1 ), A%row( ne ), A%val( ne ) )
         A%ptr = (/ 1, 2, 4, 6, 8, 10 /)
         A%row = (/ 1, 3, 2, 1, 3, 2, 4, 4, 1 /)
         A%val = (/ 11.0_wp, 32.0_wp, 22.0_wp, 13.0_wp, 33.0_wp,              &
                    24.0_wp, 44.0_wp, 45.0_wp, 15.0_wp /)
       CASE ( 5 ) ! sparse co-ordinate input format
         CALL SMT_put( A%type, 'COORDINATE', s )
         A%ne = ne
         ALLOCATE( A%row( ne ), A%col( ne ), A%val( ne ) )
         A%row = (/ 4, 1, 3, 2, 1, 3, 4, 2, 1 /)
         A%col = (/ 5, 1, 2, 2, 3, 3, 4, 4, 5 /)
         A%val = (/ 45.0_wp, 11.0_wp, 32.0_wp, 22.0_wp, 13.0_wp, 33.0_wp,     &
```

```
                44.0_wp, 24.0_wp, 15.0_wp /)
      END SELECT
      CALL CONVERT_between_matrix_formats( A, 'SPARSE_BY_ROWS', A_out,        &
                                           control, inform ) ! transform
      WRITE( 6, "( /, ' convert from ', A, ' to sparse-row format ')" )       &
        SMT_get( A%type )
      IF ( inform%status == 0 ) THEN
        DO i = 1, A_out%m
          WRITE( 6, "( ' row ', I0, ', ( column value ): ',                   &
         & ( 5( '(', I2, F5.1, ' )', : ) ) )" ) i, ( A_out%col( j ),          &
             A_out%val( j ), j = A_out%ptr( i ), A_out%ptr( i + 1 ) - 1 )
        END DO
      ELSE
        WRITE( 6, "( ' error return, status = ', I0 )" ) inform%status
      END IF
      SELECT CASE ( type ) ! deallocate space
      CASE ( 1, 2 ) ! dense + dense by columns
        DEALLOCATE( A%val )
      CASE ( 3 ) ! sparse by rows
        DEALLOCATE( A%ptr, A%col, A%val )
      CASE ( 4 ) ! sparse by columns
        DEALLOCATE( A%ptr, A%row, A%val )
      CASE ( 5 ) ! sparse co-ordinate
        DEALLOCATE( A%row, A%col, A%val )
      END SELECT
      DEALLOCATE( A_out%ptr, A_out%col, A_out%val, stat = i )
    END DO
  END DO

  END PROGRAM GALAHAD_CONVERT_EXAMPLE
```

This produces the following output:

```
convert from DENSE_BY_ROWS to sparse-row format
row 1, ( column value ): ( 1 11.0 )( 3 13.0 )( 5 15.0 )
row 2, ( column value ): ( 2 22.0 )( 4 24.0 )
row 3, ( column value ): ( 2 32.0 )( 3 33.0 )
row 4, ( column value ): ( 4 44.0 )( 5 45.0 )

convert from DENSE_BY_COLUMNS to sparse-row format
row 1, ( column value ): ( 1 11.0 )( 3 13.0 )( 5 15.0 )
row 2, ( column value ): ( 2 22.0 )( 4 24.0 )
row 3, ( column value ): ( 2 32.0 )( 3 33.0 )
row 4, ( column value ): ( 4 44.0 )( 5 45.0 )

convert from SPARSE_BY_ROWS to sparse-row format
row 1, ( column value ): ( 1 11.0 )( 5 15.0 )( 3 13.0 )
row 2, ( column value ): ( 2 22.0 )( 4 24.0 )
row 3, ( column value ): ( 3 33.0 )( 2 32.0 )
row 4, ( column value ): ( 4 44.0 )( 5 45.0 )

convert from SPARSE_BY_COLUMNS to sparse-row format
row 1, ( column value ): ( 1 11.0 )( 3 13.0 )( 5 15.0 )
row 2, ( column value ): ( 2 22.0 )( 4 24.0 )
row 3, ( column value ): ( 2 32.0 )( 3 33.0 )
```

```
row 4, ( column value ): ( 4 44.0 )( 5 45.0 )

convert from COORDINATE to sparse-row format
row 1, ( column value ): ( 1 11.0 )( 3 13.0 )( 5 15.0 )
row 2, ( column value ): ( 2 22.0 )( 4 24.0 )
row 3, ( column value ): ( 2 32.0 )( 3 33.0 )
row 4, ( column value ): ( 4 44.0 )( 5 45.0 )

construct the transpose

convert from DENSE_BY_ROWS to sparse-row format
row 1, ( column value ): ( 1 11.0 )
row 2, ( column value ): ( 2 22.0 )( 3 32.0 )
row 3, ( column value ): ( 1 13.0 )( 3 33.0 )
row 4, ( column value ): ( 2 24.0 )( 4 44.0 )
row 5, ( column value ): ( 1 15.0 )( 4 45.0 )

convert from DENSE_BY_COLUMNS to sparse-row format
row 1, ( column value ): ( 1 11.0 )
row 2, ( column value ): ( 2 22.0 )( 3 32.0 )
row 3, ( column value ): ( 1 13.0 )( 3 33.0 )
row 4, ( column value ): ( 2 24.0 )( 4 44.0 )
row 5, ( column value ): ( 1 15.0 )( 4 45.0 )

convert from SPARSE_BY_ROWS to sparse-row format
row 1, ( column value ): ( 1 11.0 )
row 2, ( column value ): ( 2 22.0 )( 3 32.0 )
row 3, ( column value ): ( 1 13.0 )( 3 33.0 )
row 4, ( column value ): ( 2 24.0 )( 4 44.0 )
row 5, ( column value ): ( 1 15.0 )( 4 45.0 )

convert from SPARSE_BY_COLUMNS to sparse-row format
row 1, ( column value ): ( 1 11.0 )
row 2, ( column value ): ( 3 32.0 )( 2 22.0 )
row 3, ( column value ): ( 1 13.0 )( 3 33.0 )
row 4, ( column value ): ( 2 24.0 )( 4 44.0 )
row 5, ( column value ): ( 4 45.0 )( 1 15.0 )

convert from COORDINATE to sparse-row format
row 1, ( column value ): ( 1 11.0 )
row 2, ( column value ): ( 2 22.0 )( 3 32.0 )
row 3, ( column value ): ( 1 13.0 )( 3 33.0 )
row 4, ( column value ): ( 2 24.0 )( 4 44.0 )
row 5, ( column value ): ( 1 15.0 )( 4 45.0 )
```