Using the TAUCS Preconditioning Library

Aleksandar Donev

June 2000

1 The \textit{TAUCS} C library

This contains an interface to the TAUCS sparse-matrix factorization library of Sivan Toledo \textit{et al.}. In places, I had to use workarounds to make the Fortran interface work. TAUCS uses double precision throughout, so sometimes implicit conversions will occur depending on the working precision \texttt{r_up}. The fundamental type in TAUCS is \texttt{taucs_ces_matrix}, which is of size 6 integers.

These preconditioners solve completely or approximately a system of the form \( Lx = b \), where \( L = L = ACA^T \) is the conductance Laplacian, \( b \) is a vector of excess flows at the nodes, and \( x \) is a vector of excess potentials at the nodes. The module \texttt{CCS_Laplacians} contains the routines that convert the Laplacian \( L \) from my network graph format to compressed column storage format (similar to adjacency lists with diagonal entries included as well).

The module \texttt{TAUCS_Interface} contains initialization, creation, solvers, and destruction routines for a preconditioner created by TAUCS. It supports two kinds of preconditioners. The first is \textit{incomplete Cholesky factorization} with a given dropping tolerance, and the second is \textit{maximal weight basis} Vaydia preconditioners (these split the MST of the network into pieces, then glue these together with one joining arc per tree pair, and then completely factorize the resulting matrix \( M \)).

This documentation is out of date and not-finished!

"\texttt{WEAVE.f90}\" 1.0.0.1

\begin{verbatim}
MODULE TAUCS_Constants   // Some predefined constants for TAUCS
    use Precision
    implicit NONE
    public

    integer, parameter :: i_taucs = i_word, r_taucs = r_dp   // The taucs integer data-type
    integer, parameter :: TAUCS_CCSDIM = 6   // Size of opaque matrix type

END MODULE TAUCS_Constants
\end{verbatim}

\begin{verbatim}
\{ CCS_Laplacians 1.2.0.1\}   // The module for CCS format conversions
\end{verbatim}
module TAUCS_Interface
use Precision
use TAUCS_Constants
use Error_Handling
use System_Monitors
use Vector_Operations
use Graph_Algorithms
use Network_Data_Types
use CCS_Laplacians
implicit none
private
public :: TAUCS_InitializePreconditioner, TAUCS_DestroyPreconditioner,
         TAUCS_CreatePreconditioner, TAUCS_FreePreconditioner, TAUCS_ApplyPreconditioner
public :: i_taucs, r_taucs  // Export these kinds

/* These routines are all C external routines. I do not give explicit interfaces for them to avoid
some compiler confusions. The routines for permuting vectors were added by me and are
overloaded generically here into the routine taucs_permute_vector. I also added the routine
taucs_order_free to free permutation arrays allocated by TAUCS, as well as taucs_ces_assemble
to assemble my own matrices into a taucs_ces_matrix structure. I will use default integers
throughout to hold C pointers, which is of course a workaround until Fortran 2002! */

ml_external :: taucs_ces_write_jv, taucs_printf, taucs_ces_assemble, taucs_logfile,
taucs_ces_free, taucs_supernodal_factor_free, taucs_ces_solve_ltl, taucs_supernodal_solve_ltl,
taucs_ces_factor_ltl, taucs_ces_factor_ltl_mf, taucs_ces_factor_ltl_s, taucs_ces_order,
taucs_free_order, taucs_ces_permute_symmetrically, taucs_amuwh_preconditioner_create,
taucs_supernodal_factor_free_numeric, taucs_ces_factor_ltl_symbolic, taucs_ces_factor_ltl_numeric,
taucs_permute_vector_sp_to_sp, taucs_permute_vector_sp_to_dp, taucs_permute_vector_dp_to_dp,
taucs_permute_vector_dp_to_sp
integer :: taucs_ces_write_jv, taucs_ces_factor_ltl_symbolic, taucs_ces_factor_ltl_numeric,
taucs_ces_factor_ltl, taucs_ces_factor_ltl_mf, taucs_ces_factor_ltl_s, taucs_ces_solve_ltl,
taucs_supernodal_solve_ltl, taucs_amuwh_preconditioner_create, taucs_ces_permute_symmetrically

interface taucs_permute_vector  // Generically overloaded vector permutation routine

subroutine taucs_permute_vector_sp_to_sp(perm, x, perm_x, n)
  use Precision
  integer :: perm, n
  real(kind=r_sp) :: x, perm_x
end subroutine

subroutine taucs_permute_vector_sp_to_dp(perm, x, perm_x, n)
  use Precision
  integer :: perm, n
  real(kind=r_sp) :: x
  real(kind=r_dp) :: perm_x
end subroutine

subroutine taucs_permute_vector_dp_to_dp(perm, x, perm_x, n)
  use Precision
  integer :: perm, n
  real(kind=r_dp) :: x, perm_x
end subroutine

subroutine taucs_permute_vector_dp_to_sp(perm, x, perm_x, n)
  use Precision
  integer :: perm, n
end subroutine

end module TAUCS_Interface
REAL (KIND = 2) :: x
REAL (KIND = 2) :: perm_x
END SUBROUTINE
END INTERFACE tauc_permute_vector

PUBLIC :: TAUCS_CCS_Matrix, TAUCS_Network_Preconditioner // Public data types
INTEGER, PARAMETER, PUBLIC :: TAUCS_Llt = 1, TAUCS_MWB = 2
// Types of preconditioners supported here
INTEGER, PARAMETER, PUBLIC :: TAUCS_incomplete_fact = 1, TAUCS_complete_fact_ll = 2,
TAUCS_complete_fact_mf = 3
// Types of factorizations available

TYPE TAUCS_CCS_Matrix // A conductance matrix in CCS format
  _PRIVATE
  INTEGER (KIND = i_tauc) :: dimension (TAUCS_CCSDIM) :: matrix // Opaque type
  INTEGER (KIND = i_tauc) :: n_vertices = 0, n_edges = 0, n_eliminated = 0
  INTEGER (KIND = i_tauc), DIMENSION (:), DYNAMIC :: col_ptr, row_ind, incident, eliminated
  // The neighbour lists and the eliminated nodes.
  REAL (KIND = r_tauc), DIMENSION (:), DYNAMIC :: val // The edge conductances
ENDTYPE

TYPE TAUCS_Network_Preconditioner // A TAUCS preconditioner
  _TYPE (Directed_Graph), POINTER :: graph // G
  _TYPE (TAUCS_CCS_Matrix) :: ccs_matrix // The matrix AHA
  _TYPE (Fill_Reducing_Ordering), POINTER :: graph_ordering
  CHARACTER (LEN = 25) :: log_file = ""
  INTEGER (KIND = i_up), DIMENSION (:), DYNAMIC :: eliminated_nodes
  // One per component
  INTEGER :: ordered_matrix = 0, ccs_factor = 0, ccs_mwb = 0, perm = 0, invperm = 0
  // Pointers needed and returned by TAUCS as opaque integers!
  INTEGER :: precond_method = TAUCS_Llt, factorization = TAUCS_incomplete_fact
  REAL :: subtree_ratio = 0.0 // The ratio n_trees / n_nodes in the MWB preconditioner
  REAL (KIND = r_tauc) :: droptol = 0.0_tauc, subgraphs = 1.0_tauc,
  diagonalConductance = 1.0_tauc
  CHARACTER (LEN = 10) :: ordering_method = "none"
  // "none", "gmmmd", "metis" or "identity"
  LOGICAL :: reorder_graph = .FALSE., allocated_eliminated = .FALSE., allocated_xb_buffers = .FALSE.
  REAL (KIND = r_up), DIMENSION (:), POINTER :: conductances = NULL // Arc conductances
  REAL (KIND = r_up), DIMENSION (:), POINTER :: flows = NULL, potentials = NULL
  // The vectors b and x in the network linear system
  REAL (KIND = r_up), DIMENSION (:), POINTER :: excess_flows = NULL, excess_potentials = NULL
ENDTYPE

CONTAINS
  ( TAUCS_InitializePreconditioner 1.1.1)
  ( TAUCS_DestroyPreconditioner 1.1.32)
  ( TAUCS_CreatePreconditioner 1.1.21)
  ( TAUCS_FreePreconditioner 1.1.31)
  ( TAUCS_ApplyPreconditioner 1.1.4.1)

END MODULE TAUCS_INTERFACE
1.1 Module *TAUCS Interface*

This module contains the routines needed to use a TAUCS preconditioner in my SSCNO network optimization library. As usual, first I define a bunch of shortcut macros to simplify my typing later on.

There is a confusion here between SCOTCH's definition of forward and reverse permutations and TAUCS's. I think SCOTCH has a mistake.

```
"WEAVE.f90" 1.1.0.1 ≡
@m _preconditioner taucs_preconditioner
@m _graph _preconditioner % graph
@m _graph_ordering _preconditioner % graph_ordering
@m _conductances _preconditioner % conductances
@m _flows _preconditioner % flows
@m _potentials _preconditioner % potentials
@m _excess_flows _preconditioner % excess_flows
@m _excess_potentials _preconditioner % excess_potentials  // I switched these two here!
@m _nodes_renumbering _graph_ordering % nodes_renumbering
@m _nodes_reordering _graph_ordering % nodes_reordering
@m _precond_method _preconditioner % precond_method
@m _factorization _preconditioner % factorization
@m _subgraphs _preconditioner % subgraphs
@m _droptol _preconditioner % droptol
@m _ordering_method _preconditioner % ordering_method
@m _reorder_graph _preconditioner % reorder_graph
@m _eliminated_nodes _preconditioner % eliminated_nodes
@m _allocated_xb_buffers _preconditioner % allocated_xb_buffers
@m _diagonal_conductance _preconditioner % diagonal_conductance
@m _ccs_matrix _preconditioner % ccs_matrix
@m _reordered_matrix _preconditioner % reordered_matrix
@m _ccs_factor _preconditioner % ccs_factor
@m _ccs_mwb _preconditioner % ccs_mwb
@m _perm _preconditioner % perm
@m _invperm _preconditioner % invperm
@m _n_edges _ccs_matrix % n_edges
@m _n_vertices _ccs_matrix % n_vertices
@m _n_eliminated _ccs_matrix % n_eliminated
@m _matrix _ccs_matrix % matrix
@m _col_ptr _ccs_matrix % col_ptr
@m _row_ind _ccs_matrix % row_ind
@m _val _ccs_matrix % val
@m _incident _ccs_matrix % incident
@m _eliminated _ccs_matrix % eliminated
@m _log_file _preconditioner % log_file
```

1.1.1 Initialization
The routine `TAUCS_InitializePreconditioner` initializes a preconditioner by allocating all the required arrays, converting the conductance matrix from my format to CCS format, and for incomplete factorizations, performing a minimal-degree ordering of the graph. This routine needs to be called only once per graph. If the graph does not change, but only the conductances, there is no need to call this routine again. But when $G$ changes, a new conversion and ordering is needed:

\[ \langle \text{TAUCS} \_\text{InitializePreconditioner} \ 1.1.1 \rangle \equiv \]

\begin{verbatim}
SUBROUTINE TAUCS_InitializePreconditioner (taucs_preconditioner)
IMPLICIT NONE
_TYPE(TAUCS_Network_Preconditioner), INTENT(INOUT), TARGET :: taucs_preconditioner

  /* Temporary and local variables: */
_TYPE(Directed_Graph), POINTER :: graph
INTEGER :: alloc_status, result
INTEGER (KIND = i_taucs) :: n_vertices, n_edges, second_node
INTEGER (KIND = i_taucs) :: n_special_nodes, n_nodes, n_special_arcs, n_arcs
LOGICAL :: valid_graph

  graph = _graph
  valid_graph = T    // Check to see if the input is valid
  IF (~ASSOCIATED(graph)) THEN
    valid_graph = F
  ELSE
    IF (~ASSOCIATED(_heads_tails)) valid_graph = F
  END IF
  IF (~ASSOCIATED(_conductances)) valid_graph = F
  IF (~valid_graph) THEN
    CALL CriticalError(message = "Incomplete or no graph passed to TAUCS
                                      library", caller = "TAUCS_InitializePreconditioner")
  RETURN
END IF

n_special_nodes = _n_special_nodes
n_nodes = _n_nodes
n_special_arcs = _n_special_arcs
n_arcs = _n_arcs
n_vertices = n_nodes + n_special_nodes + 1    // The number of vertices in CCS format
n_edges = n_vertices + n_arcs + n_special_arcs + 1    // Number of nonzeros in CCS format
  // Note: Self-loops are double counted here, but really ignored during the matrix construction

_reorder_graph = F
  IF (ASSOCIATED(_graph_ordering)) THEN
    IF (ASSOCIATED(_nodes_reordering) \& ASSOCIATED(_nodes_renumbering)) THEN
      _reorder_graph = T
    END IF
  END IF
  IF (~_reorder_graph \& ordering_method = "none") THEN
  // TAUCS will not perform any further ordering!

  /* Because of the reordering (permutation), and possibly also different precisions between r_up and r_taucs, we need to make separate buffers for the vectors x and b in the network linear system. We will then copy these back and forth from TAUCS when we apply the preconditioner. Also, arrays need to be allocated to store the Laplacian in CCS format */
\end{verbatim}
IF ((r_Taucs ≠ r_wp) | (ordering_method ≠ "none") | (r_reorder_graph)) THEN
  WRITE(message_log_unit, *) "Allocating x and b buffers in TAUCS"
  allocated_xb_buffers = T
  _AllocateNodalArray(flows, 1.0_Jaucs, ~ASSOCIATED, "TAUCS.InitializePreconditioner")
  _AllocateNodalArray(potentials, 1.0_Jaucs, ~ASSOCIATED, "TAUCS.InitializePreconditioner")
ELSE // To avoid strong-typing problems
  CALL VectorPointerCopy (source = _excess_potentials, target = _potentials)
  CALL VectorPointerCopy (source = _excess_flows, target = _potentials)
END IF

_AllocateArray(_col_ptr, 0, n_vertices, l_Jaucs, ~NON_NULL, "TAUCS.InitializePreconditioner")
// The pointer to the neighbour lists
_AllocateArray(_row_ind, 0, n_edges - 1, l_Jaucs, ~NON_NULL, "TAUCS.InitializePreconditioner")
// The adjacency list itself—with selfloops!
_AllocateArray(_incident, 0, n_edges - 1, l_Jaucs, ~NON_NULL, "TAUCS.InitializePreconditioner")
// Arc numbers for adjacency lists
_AllocateArray(_val, 0, n_edges - 1, 1.0_Jaucs, ~NON_NULL, "TAUCS.InitializePreconditioner")
// The edge conductances (entries in the Laplacian)

/* The conversion routine to CCS format is given below. It will be useful for other
general-purpose sparse-matrix libraries that can read CCS format: */
IF (~r_reorder_graph) THEN
  CALL CreateCCSLaplacian(indexing_offset = 0, arc_offset = n_special_arcs,
    node_offset = n_special_nodes, heads_tails = _heads_tails, neighbours = _row_ind,
    my_neighbours = _col_ptr, incident_arcs = _incident, _edges = _n_edges)
ELSE
  CALL CreateCCSLaplacian(indexing_offset = 0, arc_offset = n_special_arcs,
    node_offset = n_special_nodes, heads_tails = _heads_tails, neighbours = _row_ind,
    my_neighbours = _col_ptr, incident_arcs = _incident, nodes_reordering = nodes_reordering,
    nodes_renumbering = nodes_renumbering, _edges = _n_edges)
END IF

_n_vertices = n_vertices   // Save this number
_n_edges = n_edges         // Save this number

IF (_log_file ≠ "") THEN
  CALL taurc_logfile(TRIM(_log_file) || CHAR(0))   // It is very difficult to interface here
  CALL taurc_printf("The Laplacian has %d non-zero entries \n" || CHAR(0),
    _VAL(INT(_n_edges)))
END IF

/* Because the Laplacian is a singular matrix, one node per connected component needs to be
eliminated manually. Here I do this by adding a diagonal conductance to these nodes. The
user should provide the list of nodes to eliminate in _eliminated_nodes, or otherwise only the
first vertex is eliminated manually: */

IF (~NON_NULL(_eliminated_nodes)) THEN
  taurc_preconditioner % allocated_eliminated = T
  ALLOCATE(_eliminated_nodes(1:1))   // One node will always be eliminated
  _eliminated_nodes = ~n_special_nodes   // Eliminate the first node
END IF

_n_eliminated = SIZE(_eliminated_nodes)   // Diagonally compensated nodes
ALLOCATE(_eliminated(1:_n_eliminated))   // These are the eliminated vertices—internal use
IF (~r_reorder_graph) THEN

6
\begin{verbatim}
&S\_eliminated = S\_eliminated + n\_special\_nodes & // The vertices that need to be eliminated
ELSE
 & S\_eliminated = S\_nodes\_renumbering(S\_eliminated\_nodes) + n\_special\_nodes
 & // Correct for new numbering
END IF

@\#if 0
 & WRITE(*, *) c\_col\_ptr
 & WRITE(*, *) c\_row\_ind
 & WRITE(*, *) c\_incident
@\#endif

CALL taucs_crs_assemble(_matrix, _VAL(n\_vertices), _col\_ptr, _row\_ind, _val)
 & // Create the matrix—I added this routine myself!

 aftermarket: IF ((precond\_method \equiv TAUCS\_LLt)) THEN
 & // Certain steps in the factorization need only be done once
Factorsymbolic: IF ((factorization \neq TAUCS\_incomplete\_fact) \\& (factorization \neq TAUCS\_complete\_fact\_Il)) THEN
 & // We want to do the symbolic phase of the factorization now:
ReorderMatrix: IF (ordering\_method \equiv "none") THEN
 & \_cgs\_factor = taucs\_cgs\_factor\_Il\_symbolic(_matrix)
ELSE ReorderMatrix
 & // Reorder the matrix (only connectivity information is used by these routines):
& CALL taucs\_cgs\_order(_matrix, \_perm, \_invperm, TRIM(ordering\_method) || CHAR(0));
& IF ((\_perm \equiv 0) \| (\_invperm \equiv 0)) CALL CriticalError(message = "Ordering failed in
& TAUCS", caller = "TAUCS\_InitializePreconditioner")
 & // Reorder the matrix using the previously computed permutation vectors
 & // There is redundancy here as the numeric values are also permuted!
& \_reordered\_matrix = taucs\_cgs\_permute\_symmetrically(_matrix, _VAL(\_perm), _VAL(\_invperm))
& IF (\_reordered\_matrix \equiv 0) CALL CriticalError(message = "Permuting failed in
& TAUCS", caller = "TAUCS\_InitializePreconditioner")
& \_cgs\_factor = taucs\_cgs\_factor\_Il\_symbolic(_VAL(\_reordered\_matrix))
& CALL taucs\_cgs\_free(_VAL(\_reordered\_matrix))
 & // We don't need the reordered matrix any more
& \_reordered\_matrix = 0 & // We will have to re-do it again later!
END IF ReorderMatrix
ELSE FactorSymbolic & // Do not do any factorization
& IF (ordering\_method \neq "none") THEN
 & // Reorder the matrix (only connectivity information is used by these routines):
& CALL taucs\_cgs\_order(_matrix, \_perm, \_invperm, TRIM(ordering\_method) || CHAR(0));
& IF ((\_perm \equiv 0) \| (\_invperm \equiv 0)) CALL CriticalError(message = "Ordering failed in
& TAUCS", caller = "TAUCS\_InitializePreconditioner")
END IF
END IF FactorSymbolic
END IF PreFactor
END SUBROUTINE TAUCS\_InitializePreconditioner
\end{verbatim}

This code is used in section 1.0.0.1.
1.1.2 Assembling the Preconditioner and Factorization

Once the conductances are known, the array of values \texttt{val} in the Laplacian needs to be assembled (the diagonal entries are sums over all incident arcs on a given node) and then factorization needs to be performed. For the MBW preconditioner, an MST is first constructed to decide which arcs to drop from the network before factorization. The routine \texttt{ComputeCCSLaplacian} computes the edge conductances, and TAUCS routines do the rest. There is still a problem here with using complete supernodal routines!

\begin{verbatim}
(TAUCS_CreatePreconditioner 1.1.21) \equiv

SUBROUTINE TAUCS_CreatePreconditioner(taucs_preconditioner)
USE Random_Numbers, only:RandomUniform    // For seed generation
IMPLICIT NONE
_TYPE(TAUCS_Network_Preconditioner), INTENT (inout), TARGET :: taucs_preconditioner
_TYPE(Directed_Graph), POINTER :: graph
INTEGER (KIND = i_int):: arc, node
INTEGER (KIND = i_int):: edge, vertex, index, seed
INTEGER :: alloc_status, result

graph \Rightarrow _graph

/* The entries in the Laplacian are negative of the arc conductances for off-diagonal elements and sums over conductances of incident arcs for diagonal entries: */
CALL ComputeCCSLaplacian(indexing_offset = 0, arc_offset = _n_special_arcs,
    arcs_conductances = _conductances, edges_conductances = _val, neighbours = _row_ind,
    my_neighbours = _col_ptr, incident_arcs = _incident)

DO index = 1, SIZE(_eliminated)    // We eliminate nodes by compensating the diagonal entries
    vertex = _eliminated(index)
    // Add a diagonal correction for this vertex:
    _val(_col_ptr(vertex)) = _val(_col_ptr(vertex)) + _diagonal_conductance
END DO
/* Now we begin the factorization process to construct the \(LL^T\) preconditioner */
Factorize: IF (_precond_method \equiv TAUCS_LLt) THEN    // Use (in)complete \(LL^T\) factorization

ReorderMatrix: IF (_ordering_method \equiv "none") THEN

SELECT CASE (_factorization)    // Now factorize the reordered matrix
CASE (TAUCSIncompleteFact)
    _ccs_factor = taucs_ccs_factor_llt(matrix, _val(_droptol), _val(0))
CASE (TAUCSCompleteFact)
    _ccs_factor = taucs_ccs_factor_llt(matrix)
CASE DEFAULT    // Only perform the numerical phase of the factorization
    result = taucs_ccs_factor_llt_numeric(matrix, _val(_ccs_factor))
END SELECT

IF (result \neq 0) THEN
    CALL CriticalError(message = "Numerical factorization failed in TAUCS", caller = "TAUCS_CreatePreconditioner")
ENDIF
END IF
END SELECT

ELSE ReorderMatrix
\end{verbatim}
// Reorder the matrix using the previously computed permutation vectors:
_reordered_matrix = taucs_csc_permute_symmetrically(_matrix, _VAL(_perm), _VAL(_invperm))

IF (_reordered_matrix ≡ 0) CALL CriticalError(message = "Permuting failed in TAUCS", caller = "TAUCS_CreatePreconditioner")

SELECT CASE (_factorization)  // Now factorize the reordered matrix
   CASE (TAUCS_incomplete_fact)
      _ccs_factor = taucs_csc_factor_llt(_VAL(_reordered_matrix), _VAL(_droptol), _VAL(0))
   CASE (TAUCS_complete_fact_l)
      _ccs_factor = taucs_csc_factor_llt(_VAL(_reordered_matrix))
   CASE DEFAULT
      result = taucs_csc_factor_llt_numeric(_VAL(_reordered_matrix), _VAL(_ccs_factor))
      IF (result ≠ 0) THEN
         CALL CriticalError(message = "Numerical factorization failed in TAUCS", caller = "TAUCS_CreatePreconditioner")
      END IF
   END SELECT

CALL taucs_csc_free(_VAL(_reordered_matrix))  // We don't need the reordered matrix any more
_reordered_matrix = 0

END IF ReorderMatrix

ELSE  // Use Vaydia's (MWB) preconditioner
   // The number of subgraphs is a fraction of the total number of nodes:
   _subgraphs = MIN(taucs_preconditioner % _n_nodes - 1, _wp, INT(taucs_preconditioner % _n_nodes * taucs_preconditioner % subtree_ratio, _wp)))
   CALL RandUniform(seed)

   // Construct the Vaydia matrix M in CCS format:
   _ccs_mwb = taucs_amwbb_preconditioner_create(_matrix, _VAL(seed), _VAL(_subgraphs))
   /* Now we reorder and completely factor this matrix M = LLᵀ; */
   CALL taucs_csc_order(_VAL(_ccs_mwb), _perm, _invperm, TRIM(ordering_method) || CHAR(0));
   IF ((_perm ≡ 0) || (_invperm ≡ 0)) CALL CriticalError(message = "Ordering failed in TAUCS", caller = "TAUCS_CreatePreconditioner")
   _reordered_matrix = taucs_csc_permute_symmetrically(_VAL(_ccs_mwb), _VAL(_perm), _VAL(_invperm))
   IF (_reordered_matrix ≡ 0) CALL CriticalError(message = "Permuting failed in TAUCS", caller = "TAUCS_CreatePreconditioner")
   CALL taucs_csc_free(_VAL(_ccs_mwb))  // M is no longer needed
   _ccs_mwb = 0  // Explicitly nullify the pointer

SELECT CASE (_factorization)  // Now factorize the reordered matrix
   CASE (TAUCS_incomplete_fact)
      _ccs_factor = taucs_csc_factor_llt(_VAL(_reordered_matrix), _VAL(_droptol), _VAL(0))
   CASE (TAUCS_complete_fact_l)
      _ccs_factor = taucs_csc_factor_llt(_VAL(_reordered_matrix))
   CASE DEFAULT
      _ccs_factor = taucs_csc_factor_llt_mf(_VAL(_reordered_matrix))
   END SELECT

CALL taucs_csc_free(_VAL(_reordered_matrix))  // We don't need the reordered matrix now
_reordered_matrix = 0

END IF Factorize

IF (_ccs_factor ≡ 0) THEN
CALL CriticalError(message = "Factorization failed in TAUCS",
    caller = "TAUCS_CreatePreconditioner")
END IF
END SUBROUTINE TAUCS_CreatePreconditioner

This code is used in section 1.0.0.1.

1.1.3 Freeing TAUCS Memory

Since memory is managed by TAUCS explicitly, I need a separate routine here that frees the memory used up by TAUCS_CreatePreconditioner. Remember that this routine does not deallocate the space allocated by TAUCSInitializePreconditioner—the routine TAUCS_DestroyPreconditioner does that:

\( <\text{TAUCS}_{\text{FreePreconditioner}} 1.1.3.1> = \)

SUBROUTINE TAUCS_FreePreconditioner(tau_preconditioner)
   // Free memory from TAUCS_CreatePreconditioner
   IMPLICIT NONE
   TYPE (TAUCS_Network_Preconditioner), INTENT (INOUT), TARGET :: tau_preconditioner
   INTEGER :: alloc_status, result
   IF (_precond_method /= TAUCS_LLt) THEN
      IF ((_perm /= 0) AND (_inv_perm /= 0)) THEN  // Release the permutation arrays
         CALL taucs_free_order(_perm, _val(_inv_perm))
         _perm = 0;
         _inv_perm = 0
      END IF
   END IF
   IF (_cs_factor /= 0) THEN
      IF (_factorization /= TAUCS_incomplete_fact) THEN
         CALL taucs_csc_free(_val(_cs_factor))
         _cs_factor = 0  // TAUCS does not seem to nullify this
      ELSE IF (_factorization /= TAUCS_complete_factt) THEN
         CALL taucs Supernodal_factor_free(_val(_cs_factor))
         _cs_factor = 0  // Nullify manually
      ELSE  // Only free the numerical factor (no nullification)
         CALL taucs Supernodal_factor_free_numeric(_val(_cs_factor))
      END IF
   END IF
   IF (_cs_mwb /= 0) THEN  // Free the matrix M
      CALL taucs_csc_free(_val(_cs_mwb))
      _cs_mwb = 0  // For security
   END IF
   IF (_reordered_matrix /= 0) THEN
      CALL taucs_csc_free(_val(_reordered_matrix))
   END IF
\_\text{reordered} \_\text{matrix} = 0
\end \text{if}
\end \text{subroutine} \text{TAUCS\_FreePreconditioner}

This code is used in section 1.0.0.1.

\{\text{TAUCS\_DestroyPreconditioner} 1.1.3.2\} \equiv
\begin{align*}
\text{subroutine} \text{TAUCS\_DestroyPreconditioner}(\text{taucs\_preconditioner}) \\
\quad // \text{Free memory from TAUCS\_InitializePreconditioner} \\
\quad \text{implicit none} \\
\quad \_\_\text{type} (\text{TAUCS\_Network\_Preconditioner}), \text{intent (inout)}, \text{target :: taucs\_preconditioner} \\
\quad \_\_\text{integer :: alloc\_status, result} \\
\quad \text{if} ((\_\_\text{perm} \neq 0) \land (\_\_\text{invperm} \neq 0)) \quad // \text{Release the permutation arrays definitely} \\
\quad \quad \text{call taucs\_free\_order(\_\_\text{VAL(perm)}, \_\_\text{VAL(invperm)})} \\
\quad \quad \_\_\text{perm} = 0; \\
\quad \quad \_\_\text{invperm} = 0 \\
\quad \text{end if} \\
\quad \text{if} (\_\_\text{css\_factor} \neq 0) \quad // \text{Free the factor completely this time} \\
\quad \quad \text{if} (\_\_\text{factorization} \equiv \text{TAUCS\_incomplete\_fact}) \\
\quad \quad \text{call taucs\_css\_free(\_\_\text{VAL(css\_factor)})} \\
\quad \quad \text{else} \\
\quad \quad \quad \text{call taucs\_supernodal\_factor\_free(\_\_\text{VAL(css\_factor)})} \\
\quad \quad \text{end if} \\
\quad \_\_\text{css\_factor} = 0 \quad // \text{TAUCS does not seem to nullify this} \\
\quad \text{end if} \\
\quad \text{dealocate(\_\_\text{eliminated}, stat = alloc\_status)} \\
\quad \text{if (taucs\_preconditioner \% allocated\_eliminated)} \\
\quad \quad \text{dealocate(\_\_\text{eliminated\_nodes}, stat = alloc\_status)} \\
\quad \text{if (allocated\_xb\_buffers) then} \\
\quad \quad \_\_\text{DealocateArray(\_\_\text{flows}, 1.0, \_\_\text{taucs, associated})} \\
\quad \quad \_\_\text{DealocateArray(\_\_\text{potentials}, 1.0, \_\_\text{taucs, associated})} \\
\quad \quad \text{else} \\
\quad \quad \quad \text{nullify(\_\_\text{flows})} \\
\quad \quad \quad \text{nullify(\_\_\text{potentials})} \\
\quad \quad \text{end if} \\
\quad \_\_\text{DealocateArray(\_\_\text{col\_ptr}, 1.0, \_\_\text{taucs, non\_null})} \\
\quad \_\_\text{DealocateArray(\_\_\text{row\_ind}, 1.0, \_\_\text{taucs, non\_null})} \\
\quad \_\_\text{DealocateArray(\_\_\text{incident}, 1.0, \_\_\text{taucs, non\_null})} \\
\quad \_\_\text{DealocateArray(\_\_\text{val}, 1.0, \_\_\text{taucs, non\_null})} \\
\quad \text{end subroutine TAUCS\_DestroyPreconditioner}
\end{align*}

This code is used in section 1.0.0.1.

1.1.4 Preconditioning Step (Trigangular Solve)
The routine \texttt{TAUCS\_ApplyPreconditioner} accepts a previously constructed preconditioner, a right-hand side vector \( b = \text{excess\_flows} \) and does a fast triangular solve to find \( x = \text{excess\_potentials} \). The input vector \( b \) is permuted (reordered) at the start, while the solution \( x \) is permuted (using the inverse permutation this time) at the end of the routine. To avoid Fortran 90 copy-in/copy-out for the pointer arrays, I just pass the first element and trick the compiler that way:

\begin{verbatim}
<TAUCS\_ApplyPreconditioner 1.1.4.1> \equiv

\textbf{SUBROUTINE} \texttt{TAUCS\_ApplyPreconditioner (taucs\_preconditioner)}
\textbf{IMPLICIT} NONE
\textbf{TYPE} (\texttt{TAUCS\_Network\_Preconditioner}), \textbf{INTENT (INOUT)}, \textbf{TARGET} :: taucs\_preconditioner
\textbf{INTEGER} (\texttt{KIND = i\_up}) :: \texttt{arc}, \texttt{node}
\textbf{INTEGER} (\texttt{KIND = i\_taus}) :: \texttt{edge}, \texttt{vertex}, \texttt{offset}
\textbf{INTEGER} :: \texttt{alloc\_status}, \texttt{result}

\texttt{offset = -taucs\_preconditioner \_n\_special\_nodes} // Numbering offset

\textbf{IF} (\texttt{reorder\_graph}) \textbf{THEN} // Permute the input vector
  \textbf{CALL} \texttt{Vector\_Permute (indexing\_offset = offset, source = \_excess\_flows, target = \_flows,}
  \texttt{permutation = \_nodes\_reordering)}
\textbf{ELSE IF} (\texttt{ordering\_method \neq \text{"\_none\"}}) \textbf{THEN} // The vectors still need to be permuted
  \textbf{CALL} \texttt{taus\_permute\_vector (perm = \_VAL (\_perm), x = \_excess\_flows\_offset, perm\_x = \_flows\_offset,}
  \texttt{n = \_VAL (\_n\_vertices))} // Reorder \( b \)
\textbf{ELSE} // Try to use the same arrays without copying
  \textbf{IF} (\texttt{allocated\_xb\_buffers}) \textbf{CALL} \texttt{Vector\_Pointer\_Copy (source = \_excess\_flows, target = \_flows)}
\textbf{END IF}

\textbf{IF} (\texttt{factorization \equiv \text{TAUCS\_incomplete\_fact}}) \textbf{THEN} // Use incomplete factorization
  \texttt{result = taus\_ccs\_solve\_lt(\_VAL (\_cfs\_factor), \_potentials\_offset, \_flows\_offset)}
\textbf{ELSE} // For supernodal factor
  \texttt{result = taus\_supernodal\_solve\_lt(\_VAL (\_cfs\_factor), \_potentials\_offset, \_flows\_offset)}
\textbf{END IF}

\textbf{IF} (\texttt{result \neq 0}) \textbf{THEN}
  \textbf{CALL} \texttt{Critical\_Error (message = \"Triangular solve failed in \text{TAUCS}\",}
  \texttt{caller = \"\text{TAUCS\_Apply\_Preconditioner\"}})
\textbf{END IF}

\textbf{IF} (\texttt{reorder\_graph}) \textbf{THEN} // Permute the input vector
  \textbf{CALL} \texttt{Vector\_Permute (indexing\_offset = offset, source = \_potentials, target = \_excess\_potentials,}
  \texttt{permutation = \_nodes\_renumbering)}
\textbf{ELSE IF} (\texttt{ordering\_method \neq \text{"\_none\"}}) \textbf{THEN} // The vectors still need to be permuted
  \textbf{CALL} \texttt{taus\_permute\_vector (perm = \_VAL (\_jw\_perm), x = \_excess\_potentials\_offset,}
  \texttt{x = \_potentials\_offset, n = \_VAL (\_n\_vertices))} // Reorder \( x \)
\textbf{ELSE}
  \textbf{IF} (\texttt{allocated\_xb\_buffers}) \textbf{CALL} \texttt{Vector\_Pointer\_Copy (source = \_potentials,}
  \texttt{target = \_excess\_potentials)}
\textbf{END IF}
\textbf{END IF}
\textbf{END SUBROUTINE} \texttt{TAUCS\_Apply\_Preconditioner}
\end{verbatim}

This code is used in section 1.0.0.1.

1.2 CCS Laplacian
To interface with general-purpose sparse codes I need some routines for converting the Laplacian to sparse compressed-column storage (CCS) format. These routines are similar to the ones in my Graph Al{gorithms module for making adjacency lists. The difference is that only the lower triangle of $L$ is stored, so that each arc is assigned to MIN(head, tail), and that the diagonal entries are also stored explicitly (there is some wasted storage there of course, but not too much). These are not heavily commented:

\[
\{\text{CCS}_{\text{Laplacians}} \text{ 1.2.0.1} \} \equiv
\]

**MODULE** \text{CCS}_{\text{Laplacians}}

**USE** Precision
**USE** TAUCS\_Constants
**USE** Error\_Handling
**USE** System\_Monitors
**USE** Network\_Matrix\_Operations
**USE** Graph\_Algorithms
**IMPLICIT** NONE
**PUBLIC** :: CreateCCSLaplacian, ComputeCCSLaplacian

**PRIVATE**

**CONTAINS**

\(
\{ \text{CreateCCSLaplacian \ 1.2.0.2} \}
\)

\(
\{ \text{ComputeCCSLaplacian \ 1.2.0.3} \}
\)

**END MODULE** \text{CCS}_{\text{Laplacians}}

This code is used in section 1.0.0.1.

\[
\{\text{CreateCCSLaplacian \ 1.2.0.2} \} \equiv
\]

**SUBROUTINE** CreateCCSLaplacian (indexing\_offset, arc\_offset, node\_offset, heads\_tails, neighbours, my\_neighbours, incident\_arcs, nodes\_reordering, nodes\_renumbering, n\_edges)

**IMPLICIT** NONE

**INTEGER** (KIND = i\_wp), INTENT (IN) :: node\_offset, arc\_offset, indexing\_offset

**INTEGER** (KIND = i\_wp), DIMENSION (:, -arc\_offset :), INTENT (IN) :: heads\_tails

**INTEGER** (KIND = i\_taucs), DIMENSION (indexing\_offset :), INTENT (OUT) :: neighbours, my\_neighbours, incident\_arcs

**INTEGER** (KIND = i\_wp), DIMENSION (-node\_offset :), INTENT (IN), OPTIONAL :: nodes\_reordering, nodes\_renumbering

**INTEGER** (KIND = i\_wp), INTENT (OUT), OPTIONAL :: n\_edges \quad // The true number of edges

**INTEGER** (KIND = i\_wp), DIMENSION (:), ALLOCATABLE :: nodes\_degrees \quad // The nodal degrees

**INTEGER** (KIND = i\_wp) :: n\_nodes, n\_special\_nodes, n\_arcs, n\_special\_arcs, n\_vertices \quad // Counters

**INTEGER** (KIND = i\_wp) :: index, node, arc, head, tail, special\_node, special\_arc, degree, head\_neighbour, tail\_neighbour, node\_shift, arc\_shift, arc\_counter, node\_counter

**INTEGER** :: alloc\_status, status

**LOGICAL** :: graph\_OK, reorder\_nodes

n\_special\_nodes = node\_offset

n\_nodes = SIZE(my\_neighbours, i\_wp) - 2 - n\_special\_nodes

13
// The last entry in my_neighbours is a sentinel value!
n_special_arcs = -1BOUND(heads_tails, iwp, DIM = 2)
n_arcs = 1BOUND(heads_tails, iwp, DIM = 2)
n_vertices = n_nodes + n_special_nodes + 1
node_shift = n_special_nodes + indexing_offset // Shift in node numbering
arc_shift = n_special_arcs + indexing_offset // Shift in arc numbering
reorder_nodes = (PRESENT(nodes_reordering) \&\& PRESENT(nodes_renumbering))
// The nodes will be reordered when making the CCS Laplacian
AllocateNodalArray(nodes_degrees, iwp, ALLOCATED, "CreateCCSLaplacian")
/* First we need to calculate nodal degrees: */
nodes_degrees = 1 // Every node must have a self-loop value
DO arc = -n_special_arcs, n_arcs
  head = heads_tails(1, arc)
  tail = heads_tails(2, arc)
  IF (reorder_nodes) THEN
    head = nodes_renumbering head
    tail = nodes_renumbering tail
  END IF
  // Note: Each arc belongs to the smaller of the head and tail!
  IF (head < tail) THEN
    nodes_degrees head = nodes_degrees head + 1
  ELSE IF (tail < head) THEN
    nodes_degrees tail = nodes_degrees tail + 1
  END IF
END DO
/* Now we do a SUM_PREFIX on the nodal degrees to figure out the offsets in my_neighbours: */
arc_counter = indexing_offset
DO node = -n_special_nodes, n_nodes
  my_neighbours node + node_shift = arc_counter
  arc_counter = arc_counter + nodes_degrees node
END DO
my_neighbours n_vertices + indexing_offset = arc_counter // Sentinel end-of-array value
IF (PRESENT(n_edges)) n_edges = arc_counter - 1 // No self-loops included here
/* Now we can start making the adjacency lists: */
DO arc = n_arcs, -n_special_arcs, -1
  // Now we traverse the arcs in reverse order to make the adjacency lists
  head = heads_tails1, arc
  tail = heads_tails2, arc
  IF (reorder_nodes) THEN
    head = nodes_renumbering head
    tail = nodes_renumbering tail
  END IF
  IF (head < tail) THEN
    nodes_degrees head = nodes_degrees head - 1
    head_neighbour = my_neighbours head + node_shift + nodes_degrees head
    neighbours head_neighbour = tail + node_shift
    incidence arcs head_neighbour = arc
  ELSE IF (tail < head) THEN
    nodes_degrees tail = nodes_degrees tail - 1
END IF

14
\begin{verbatim}
tail_neighbour = my_neighbours_tail + node_shift + nodes_degrees_tail
neighbours_tail_neighbour = head + node_shift
incident_arcs_tail_neighbour = arc

END IF

END DO

/* Diagonal entries need to be handled separately: */
DO node = -n_special_nodes, n_nodes    // The self loop edges are artificial
    neighbours_my_neighbours(node + node_shift) = node + node_shift
    incident_arcs_my_neighbours(node + node_shift) = 0    // A dummy value
END DO

_DeallocateArray(nodes_degrees, i_wp, ALLOCATED)

END SUBROUTINE CreateCCSLaplacian

(ComputeCCSLaplacian 1.2.0.3) ≡

SUBROUTINE ComputeCCSLaplacian (indexing_offset, arc_offset, arcs_conductances,
        edges_conductances, neighbours, my_neighbours, incident_arcs)

IMPLICIT NONE
INTEGER (KIND = i_wp), INTENT (IN) :: indexing_offset, arc_offset
INTEGER (KIND = i_taus), DIMENSION (indexing_offset :) , INTENT (IN) :: neighbours, 
                                my_neighbours, incident_arcs
REAL (KIND = r_wp), DIMENSION (-arc_offset :) , INTENT (IN) :: arcs_conductances
REAL (KIND = r_taus), DIMENSION (indexing_offset :) , INTENT (OUT) :: edges_conductances

INTEGER (KIND = i_wp) :: n_vertices    // Counters
INTEGER (KIND = i_wp) :: node, arc, vertex, edge, first, second
n_vertices = size(my_neighbours, i_wp) - 1

DO vertex = indexing_offset, n_vertices + indexing_offset - 1
    edges_conductances (my_neighbours(vertex)) = 0.0_r_taus    // Initialize the diagonal sums
END DO

DO vertex = indexing_offset, n_vertices + indexing_offset - 1    // Convert the Laplacian format
    first = my_neighbours(vertex)    // This node
    DO edge = my_neighbours(vertex) + 1, my_neighbours(vertex+1) - 1
        arc = incident_arcs(edge)
        edges_conductances(edge) = -arcs_conductances(arc)    // L is an M matrix
        second = my_neighbours(neighbours(edge))    // The neighbour node
        // Now do the diagonal summation:
        edges_conductances(first) = edges_conductances(first) + arcs_conductances(arc)
        edges_conductances(second) = edges_conductances(second) + arcs_conductances(arc)
    END DO
END DO

END SUBROUTINE ComputeCCSLaplacian
\end{verbatim}

This code is used in section 1.2.0.1.
@m  CASE_TYPE TYPE
@m  TYPE TYPE
@m  _NULL > NULL()
@m  _PRIVATE PRIVATE
@m  _SIZE(array, _kind,...)
    sIFELSE (_0, 0, INT(_SIZE(array), KIND=_kind), INT(_SIZE(array, #), KIND=_kind))
@m  _MAXLOC(array, _kind,...)
    sIFELSE (_0, 0, INT(_MAXLOC(array), KIND=_kind), INT(_MAXLOC(array, #), KIND=_kind))
@m  _MINLOC(array, _kind,...)
    sIFELSE (_0, 0, INT(_MINLOC(array), KIND=_kind), INT(_MINLOC(array, #), KIND=_kind))
@m  _LBOUND(array, _kind,...) sIFELSE (_0, 0, INT(_LBOUND(array, DIM=1), KIND=_kind),
    INT(_LBOUND(array, #), KIND=_kind))
@m  _UBOUND(array, _kind,...) sIFELSE (_0, 0, INT(_UBOUND(array, DIM=1), KIND=_kind),
    INT(_UBOUND(array, #), KIND=_kind))
@m  _GenericInterface(generic_name,...)
    INTERFACE generic_name MODULE PROCEDURE #.
END INTERFACE generic_name
@m  _Declare_i_word(...)  INTEGER :: #.
@m  _Declare_i_wp(...)
    INTEGER (KIND = i_wp) :: #.
@m  _Declare_r_wp(...)
    REAL (KIND = r_wp) :: #.
@m  _Declare_r_sp(...)
    REAL (KIND = r_sp) :: #.
@m  _Declare_r_dp(...)
    REAL (KIND = r_dp) :: #.
@m  _FullExtent (_rank) : DO (DIM, 2, _rank) { }, { }
@m  _VarSequence(_variable, _start, _end)
    _variable # _start _DO (DIM, $VAL(_start + 1), _end) { , _variable & DIM }
@m  _NestedLoopStart(_variable, array, _rank, _kind)
    _DO (_DIM, _rank, 1, -1) { _DO _variable & DIM = _LBOUND(array, _kind, DIM = _DIM),
    _UBOUND(array, _kind, DIM = _DIM) }
@m  _NestedLoopEnd(_rank) _DO (_DIM, 1, _rank) { END _DO }
@m  _Dummy(...)  
@m  _DisplayArray(message, array)
    IF (_SIZE(array) <= 20) THEN
        WRITE(message, print_unit, "(A)" message
        WRITE(message, print_unit, "(205.2)") array
    END IF