Cholesky Factorization and PCG Solvers for the Dual Newton System

Aleksandar Donev

July 2001

1 Module Dual Network Solvers

This module provides routines for solving a random-resistor network quadratic subproblem within SSCNO using either a direct solver (from the TAUCS library), or an iterative solver, with several novel support-graph partitioners. There is more work and improvement that can be done here!

This documentation is out of date and incomplete.

This module has routines for solving the system of Newton equations during the course of the network optimization:

\[(AD_A A^T)x = b\]

It primarily uses the iterative PCG routines for the solution, with several incomplete factorization and diagonal preconditioners. The organization of the solver routines follows the design principles set forth in the module Conjugate Gradient. Here I extend the CG Solver type into a Dual Network System data-type with a pointer to it Dual System Handle that can be used to identify systems.

As discussed in Conjugate Gradient, we have to restrict solver systems to integers, so here I make a pointer array dual systems of available systems (i.e. solvers) and make it public, so the user can allocate as many solvers as needed and manipulate the systems himself. By default \[\text{MAX}(10, \text{IKN})\] solvers are allocated in InitializeDualSolver if no systems are already allocated.

The available preconditioners for now are (other than plain CG with no precond):

- **diagonal precond** will use the diagonal \(\text{Diag}(AD_A A^T)\) as a preconditioner
- **MST_BHBT precond** will use an MST preconditioner of the form \(BD_B B^T\)
- **MST_QR precond** is a bridge between the MST and diagonal preconditioners which performs an incomplete QR factorization of \(AD_A A^T\) in the form \(FD_F F^T\) where \(F\) is a generalized basis.
- **MST_LDLT precond** uses an exact \(FD_F F^T\) Cholesky factorization of \(BD_B B^T + \text{Diag}(ND_N N^T)\) as a preconditioner. This is the recommended preconditioner for high-accuracy solvers.
MODULE Dual_Network_Solvers
    USE Precision        // Kind parameters
    USE Error_Handling
    USE System_Monitors
    USE Graph_Algorithms
    USE Network_Matrix_Operations
    USE Network_Data_Type
    USE SCOTCH_Interface
    USE CHACO_Interface
    USE CHACO_Constants
    USE TAUCS_Interface
    USE Support_Trees
    USE Conjugate_Gradient
    USE Vector_Operations
    USE Network_Graphics     // Temporary for debugging
IMPLICIT NONE
PUBLIC :: InitializeDualSolver, InitializeDualPreconditioner, DestroyDualSolver,
    DestroyDualPreconditioner, SolveDualSystem_PCG, SolveDualSystem_LLt
PUBLIC :: Dual_Network_Preconditioner, Dual_Network_Factorization, Dual_System_Timers,
    Dual_Network_System, Dual_System_Handle
    // Derived data-types
PRIVATE
INTEGER, PARAMETER, PUBLIC    :: no_precond = 0, diagonal_precond = 1, MST_BHBt_precond = 2,
    MST_QR_precond = 3, MST_LDLt_precond = 4, ST_precond = 5, ST_MST_precond = 6,
    TAUCS_MWB_precond = 7, TAUCS_LLt_precond = 8
    // Possible dual preconditioners—more coming soon, maybe
PUBLIC :: Support_Tree_Arrays, Support_Tree_Preconditioner
INTEGER, PARAMETER, PUBLIC    :: Default_ordering = 0, Matching_ordering = 1,
    SCOTCH_ordering = 2, CHACO_ordering = 3
    // Supported partitioning strategies for support-tree preconditioners
TYPE Support_Tree_Arrays
    INTEGER (KIND = i_up), DIMENSION (:), _DYNAMIC :: _NULLIFIED (nodes_mapping)
    // The final partitioning (mapping) of the nodes
LOGICAL (KIND = L_up), DIMENSION (:), _DYNAMIC :: _NULLIFIED (crossing_arcs_mask)
    // Selects the arcs that do not cross across partitions (internal arcs)
REAL (KIND = r_up), DIMENSION (:), _DYNAMIC :: _NULLIFIED (ST_arcs_resistances),
    _NULLIFIED (ST_potentials_and_flows)
ENDTYPEN

TYPE Support_Tree_Preconditioner
  _TYPE (SCOTCH_Mapping_Ordering), POINTER :: scotch_mapping = _null     // Maybe pointers?
  _TYPE (CHACO_Mapping_Partitioning), POINTER :: chaco_mapping = _null
    // Could be POINTER?
  _TYPE (Directed_Graph), POINTER :: graph = _null
  _TYPE (Support_Tree_Mapping) :: mapping
  _TYPE (Support_Tree_Arrays), POINTER :: arrays = _null
INTEGER (KIND = i_up) :: height = 0, max_height = 0, degree = 2

2
INTEGER (kind = i_wp) :: n_leaves = 0, n_nodes = 0
INTEGER :: mapping_method = Default_ordering  // Default ordering
LOGICAL :: CHACO_sequence = F  // Do full partitioning or just sequencing
LOGICAL :: support_special nodes = T, allocated_ST_arrays = F
ENDTYPE

TYPE Dual_Network_Preconditioner
  INTEGER :: method = diagonal_precond  // Preconditioner choice
  _TYPE (Spanning_Tree), POINTER :: tree = _NULL  // For preconditioning, B
  _TYPE (Support_Tree_Preconditioner), POINTER :: support_tree = _NULL
  _TYPE (TAUCS_Network_Preconditioner), POINTER :: taucs_preconditioner = _NULL
  LOGICAL :: use_spanning_tree = F, use_support_tree = F, reoptimize_ST_MST = T,
  allocated_taucs = F, allocated_tree = F, allocated_support_tree = F
  REAL (kind = r_wp), DIMENSION (:), DYNAMIC :: _NULLIFIED (nodes_resistances),
  _NULLIFIED (tree_arcs_excess_voltages), _NULLIFIED (nodes_conductances),
  _NULLIFIED (nodes_multiphers)
  // Arrays used for preconditioning
ENDTYPE

INTEGER, PARAMETER, PUBLIC :: TAUCS_fact_Llt = TAUCS_incomplete_fact,
  TAUCS_fact_ll = TAUCS_complete_fact_ll, TAUCS_fact_mf = TAUCS_complete_fact_mf
  // Supported direct factorizations for now
INTEGER, PARAMETER, PUBLIC :: no_fact_ordering = 0, TAUCS_fact_ordering = 1,
  SCOTCH_fact_ordering = 2
  // Node ordering methods available

TYPE Dual_Network_Factorization
  INTEGER :: factorization = TAUCS_fact_mf, ordering = SCOTCH_fact_ordering  // Choices
  _TYPE (FillReducingOrdering), POINTER :: graph_ordering
  _TYPE (SCOTCH_MappingOrdering), POINTER :: scotch_ordering = _NULL
  _TYPE (TAUCS_Network_Preconditioner), POINTER :: taucs_factorization = _NULL
  LOGICAL :: reorder_nodes = T, allocated_ordering = F, allocated_scotch_ordering = F,
  allocated_factorization = F
  INTEGER (kind = i_wp), DIMENSION (:), DYNAMIC :: _NULLIFIED (nodes_reordering),
  _NULLIFIED (nodes_renumbering)  // Permutation arrays for the nodes
  // REAL (kind = r_wp), DIMENSION (:), DYNAMIC :: ENDTYPE

TYPE Dual_System_Timers
  _TYPE (CG_Timers) :: pcg_timers  // For timing the CG components
  INTEGER :: precond_initialization_timer = -1, precond_reoptimization_timer = -1,
    precond_factorization_timer = -1, pcg_timer = -1, lb_ordering_timer = -1,
    lb_factorization_timer = -1, lb_solver_timer = -1
  // Timing the solvers
ENDTYPE

INTEGER, PARAMETER, PUBLIC :: dual_network_LLt = 1, dual_network_PCG = 2
  // For now one can either use Cholesky factorization or preconditioned conjugate gradient

TYPE Dual_Network_System  // A system for a dual linear system and its solver
  INTEGER :: PIN = 1  // A unique identifier
  INTEGER :: method = dual_network_LLt  // Also dual_network_PCG
  _TYPE (CG_System) :: system  // extends (CG_System) in F2x
  _TYPE (Directed_Graph), POINTER :: graph = _NULL  // The graph G, contains A
  _TYPE (Network_Arrays), POINTER :: arrays = _NULL  // All the arrays needed
  _TYPE (Dual_Network_Preconditioner) :: preconditioner  // A preconditioner
  _TYPE (Dual_Network_Factorization) :: factorization  // A complete Cholesky factorization
ENDTYPE
1.1 Macro Shortcuts

Since I decided to use derived-data types in this implementation extensively, I define several macros to simplify typing by avoiding some of the structure components. Only the parent derived-data type is included here. These could have also been implemented as true function macros, but some other time maybe...

"WEAVE.f90" 1.0.1.1
@m PIN dual_system % PIN
@m _system dual_system % system
@m _system(PIN) dual_systems(PIN) % system
@m _allocated_solver dual_system % allocated_solver
@m _graph dual_system % graph
@m _arrays dual_system % arrays
@m _solver system % solver
@m _solver_method dual_system % method
@m _rooted_graph dual_system % rooted_graph
@m _ground_dummy dual_system % ground_dummy
@m _grounding_conductance dual_system % grounding_conductance
@m _factorization dual_system % factorization
@m _reorder_nodes _factorization % reorder_nodes
@m _fact_method _factorization % factorization
@m _order_method _factorization % ordering
@m _taucs_factorization _factorization % taucs_factorization
@m _scotch_ordering _factorization % scotch_ordering
@m _allocated_ordering _factorization % allocated_ordering
1.2 Initializing and “Destroying” the Solvers

Of course, a few arrays need to be allocated before using the solvers. Here the initialization routine InitializeDualSolver is a bit more careful than usual in that some of the arrays and pointers may already be allocated or associated with arrays that the user had available. In most cases however separate arrays will need to be allocated for the CG temporaries and this routine will do this. The deallocation routine DestroyDualSolver is not that careful though, since checking all association and allocation statuses is too complicated of a task.

1.2.1 InitializeDualSolver

This routine will initialize and prepare for usage a dual solver. It is the user’s responsibility to pass a valid associated pointer dual_pointer which also has the component tree, and therefore its child graph associated, so that at least the head and tails arrays and the number of nodes must be given (this pointer will be copied internally and used later, so the actual target must have the TARGET attribute and remain in existence throughout the usage of the system). The user must make sure a valid PIN number is assigned in the system dual_system % system for the system at hand. Whether the rest of the tree data-structures are associated or not depends solely on whether the user plans to use a tree preconditioner. The rest of the arrays and array pointers can be either non-associated (i.e. nullified), in which case this routine will allocate them. If the pointers are associated then they are left as they are. An associated pointer is assumed to have all of its components valid (or to be made valid by the user)—only unassociated pointer are allocated explicitly.

⟨InitializeDualSolver 1.2.1⟩ ≡

```
SUBROUTINE InitializeDualSolver(dual_system)
  _TYPE(Dual_Network_System), INTENT(INOUT), TARGET :: dual_system // Could also be POINTER
  // It needs to be TARGET because of internal pointer references to itself
  _TYPE(Directed_Graph), POINTER :: graph
  _TYPE(Spanning_Tree), POINTER :: tree
  _TYPE(CG_Solver), POINTER :: solver
  _TYPE(Network_Arrays), POINTER :: arrays
  _TYPE(SCOTCH_Mapping_Ordering), POINTER :: scotch_ordering
  _TYPE(TAU_C Netzwerk_Preconditioner), POINTER :: taucx_factorization
  INTEGER(KIND = i_int) :: n_special_nodes, n_nodes, n_special_arcs, n_arcs
  LOGICAL :: valid_problem // A logical indicator
  INTEGER :: alloc_status

  graph => _graph
  arrays => _arrays

  IF (.NOT. ASSOCIATED(graph)) THEN
```

6
valid_problem = \text{true}
ELSE
    IF (\text{~NON}\_\text{NULL}(\text{heads}\_\text{tails})) valid_problem = \text{true}
END IF
ELSE
    IF (\text{~ASSOCIATED}(\text{arrays})) THEN
        valid_problem = \text{true}
    END IF
ELSE
    // All needed arrays in \text{arrays} must be allocated externally
    IF (\text{~NON}\_\text{NULL}(\text{arcs}\_\text{conductances}) \land \text{~NON}\_\text{NULL}(\text{nodes}\_\text{excess}\_\text{flows}) \land \text{~NON}\_\text{NULL}(\text{nodes}\_\text{excess}\_\text{potentials})) valid_problem = \text{false}
END IF
IF (valid_problem) THEN
    CALL CriticalError(message = "Network system not fully specified", caller = "InitializeDualSolver")
RETURN
END IF

\text{n}_\text{special}\_\text{nodes} = \text{n}_\text{special}\_\text{nodes}
\text{n}_\text{nodes} = \text{n}_\text{nodes}
\text{n}_\text{special}\_\text{arcs} = \text{n}_\text{special}\_\text{arcs}
\text{n}_\text{arcs} = \text{n}_\text{arcs}

\text{// This code allows for a special dummy rooting node at index 0, which will not be solved for in}
\text{n}_\text{nodes}\_\text{excess}\_\text{potentials}
\text{//}
IF ((\text{n}_\text{special}\_\text{nodes} \geq 0) \land (\text{n}_\text{nodes} \geq 0)) THEN
    \text{rooted}\_\text{graph} = \text{true}
ELSE
    \text{ground}\_\text{dummy} = \text{false}
END IF

\text{Use}\text{PCG}: IF (\text{solver}\_\text{method} \equiv \text{dual}\_\text{network}\_\text{pcg}) THEN
    \text{// A conjugate-gradient solver is needed}
    IF (\text{~ASSOCIATED}(\text{dual}\_\text{system} \% \text{solver})) THEN
        \text{allocated}\_\text{solver} = \text{true}
    \text{ALLOCATE} (\text{dual}\_\text{system} \% \text{solver})
END IF
\text{solver} \Rightarrow \text{dual}\_\text{system} \% \text{solver}
\text{solver} \% \text{timers} = \text{timers} \% \text{pcg}\_\text{timers}
\text{solver} \% lb = \text{n}_\text{special}\_\text{nodes}
\text{solver} \% ub = \text{n}_\text{nodes}
\text{solver} \% \text{cg}_\text{x} = \text{n}_\text{nodes}\_\text{excess}\_\text{potentials}
\text{system} \% \text{cg}_\text{b} = \text{n}_\text{nodes}\_\text{excess}\_\text{flows}
\text{AllocateNodalArray} (\text{cg}_\text{nodes}\_\text{flows}, 1.0, \text{~NON}\_\text{NULL}; \text{InitializeDualSolver})
\text{AllocateNodalArray} (\text{cg}_\text{nodes}\_\text{potentials}, 1.0, \text{~NON}\_\text{NULL}; \text{InitializeDualSolver})
\text{AllocateNodalArray} (\text{cg}_\text{residual}\_\text{flows}, 1.0, \text{~NON}\_\text{NULL}; \text{InitializeDualSolver})
\text{AllocateNodalArray} (\text{cg}_\text{residual}\_\text{potentials}, 1.0, \text{~NON}\_\text{NULL}; \text{InitializeDualSolver})
IF (\text{~ASSOCIATED} (\text{matrix})) THEN
    \text{ALLOCATE} (\text{matrix})
END IF
\text{// In F2x \text{matrix} \% \text{MatrixVectorMultiplication} \Rightarrow \text{DualMatrixMultiply}}
IF (\text{~ALLOCATED(\text{dual}\_\text{systems}))) THEN \text{// Make a pool of solvers}
    \text{ALLOCATE} (\text{dual}\_\text{systems} (1 : \text{MAX}(10, \text{PIN})))
END IF
ELSE UsePCG    // Use complete Cholesky factorization
IF (~ASSOCIATED(_taucs_factorization)) THEN
   _allocated_factorization = T
   ALLOCATE (_taucs_factorization, stat = alloc_status)
END IF
_taucs_factorization ⇒ _taucs_factorization

_taucs_factorization % graph ⇒ graph
_taucs_factorization % conductances ⇒ _ars_conductances
_taucs_factorization % excess_potentials ⇒ _nodes_excess_potentials
_taucs_factorization % excess_flows ⇒ _nodes_excess_flows
_taucs_factorization % preconditioner = TAUCS_LLT

_taucs_factorization % factorization = fact_method

// With direct factorization, one node in each connected-component must be eliminated manually.
// Here I assume that there is only one connected component. Usually the dummy (node 0) will be grounded, which is the default here: */
IF (~NON_NULL(_taucs_factorization % eliminated_nodes)) THEN
   // Should I ever let the user set eliminated_nodes?
   ALLOCATE (taucs_factorization % eliminated_nodes (1 : 1), stat = alloc_status)

GroundGraph: IF (_rooted_graph) THEN    // Eliminate only the dummy node
   _taucs_factorization % eliminated_nodes = (0 /)
ELSE
   // Choose the first special node as grounded
   _taucs_factorization % eliminated_nodes = (~n_special_nodes /)
END IF GroundGraph

END IF

_taucs_factorization % diagonal_conductance = grounding_conductance
    // Grounding arc for dummy node

IF ((_order_method ≡ SCOTCH_fact_ordering) | (_order_method ≡ no_fact_ordering))
   _taucs_factorization % ordering_method = "none"    // Inhibit reordering in TAUCS

_reorder_nodes = F
IF (_order_method ≡ SCOTCH_fact_ordering) THEN

   _reorder_nodes = T
IF (~ASSOCIATED(_graph_ordering)) THEN
   _allocated_ordering = T
   ALLOCATE (_graph_ordering, stat = alloc_status)
END IF
_AllocateNodalArray(_nodes_renumbering, 1, _wp, ~NON_NULL, "InitializeDualSolver")
_AllocateNodalArray(_nodes_reordering, 1, _wp, ~NON_NULL, "InitializeDualSolver")
_graph_ordering % nodes_renumbering ⇒ _nodes_renumbering
_graph_ordering % nodes_reordering ⇒ _nodes_reordering

_taucs_factorization % graph_ordering ⇒ _graph_ordering
END IF

UseSCOTCH: IF (_order_method ≡ SCOTCH_fact_ordering) THEN

    // We need to interface to SCOTCH now
IF (~ASSOCIATED(_scotch_ordering)) THEN
   _allocated_scotch_ordering = T
   ALLOCATE (_scotch_ordering, stat = alloc_status)
END IF
_scotch_ordering ⇒ _scotch_ordering
scotch_ordering % graph => graph
scotch_ordering % map_graph = F
scotch_ordering % order_graph = T
scotch_ordering % graph_ordering => _graph_ordering

IF (fact_strategy = "n") THEN
  fact_strategy = "m"
END IF
WRITE (message_log_unit, "+" "Using SCOTCH fill-reducing ordering ", TRIM(fact_strategy)
CALL StartTimer (jlt_order_timer)
CALL SCOTCH_InitializeMappingOrdering (mapping_ordering = scotch_ordering)
CALL SCOTCH_ComputeMappingOrdering (mapping_ordering = scotch_ordering)
CALL StopTimer (jlt_order_timer)
CALL SCOTCH_DestroyMappingOrdering (mapping_ordering = scotch_ordering)
IF (allocated_scotch_ordering) THEN
  DEALLOCATE (scotch_ordering, stat = alloc_status)
END IF

@#if 0
  InitGraphics("Fill-reducing node numbering")
  CALL PlotNetwork2D ( heads_tails = heads_tails ; 1 : , node_offset = 1,
                  node_coords = _nodes_coordinates ; 1 : , node_values = REAL(_nodes_renumbering 1 :
                  , t_wp), node_size_range =(/ -HUGE(1.0_wp) / 20, HUGE(1.0_wp) / 10 ), vector_type = 0 )
  ENDGraphics
@#endif
END UseSCOTCH

WRITE (message_log_unit, "+" "Initializing a complete factorization structure in TAUCS."
CALL TAUCS_InitializePreconditioner (taucs_preconditioner = taucs_factorization)
END if UsePCG

END SUBROUTINE InitializeDualSolver

This code is used in section 1.0.0.1.

1.2.2 Initializing the Preconditioner

If an iterative solver is wanted, a preconditioner needs to be created. Here I have implemented in a test-
version several novel support-graph preconditioners, and I use SCOTCH, CHACO and TAUCS. Support-
graph preconditioners work well, but they are somewhat complicated and require a lot of additional data-
structures, so this section is rather entangled. In the future a separate routine CreatePreconditioner should
be added to this file that will be called if only certain numerical values change, but not the graph itself. At
present some initialization is done here and it does not use numerical values...

(InitializeDualPreconditioner 1.2.2.1) ≡

SUBROUTINE InitializeDualPreconditioner (dual_system)

9
_TYPE (Dual_Network_System), INTENT (inout), TARGET : dual_system    // Could also be POINTER
  // It needs to be TARGET because of internal pointer references to itself

INTEGER :: alloc_status
_TYPE (Directed_Graph), POINTER :: graph
_TYPE (Spanning_Tree), POINTER :: tree
_TYPE (Network_Arrays), POINTER :: arrays
_TYPE (CG_Solver), POINTER :: solver
_TYPE (TAUCS_Network_Preconditioner), POINTER :: taucs_preconditioner

INTEGER (kind = i_int) :: n_special_nodes, n_nodes, n_special_arcs, n_arcs, node, arc

IF (_solver_method /= dual_network_pcg) THEN
  CALL Warning("Preconditioner initialization attempted without PCG",
               caller = "InitializeDualPreconditioner")
  RETURN
END IF

graph => _graph
n_special_nodes = n_special_nodes
n_nodes = n_nodes
n_special_arcs = n_special_arcs
n_arcs = n_arcs
arrays => arrays
solver => system % solver

CALL StartTimer (_precond_init_timer)

IF (_precond_method == MST_BHH.precond) OR
   (_precond_method == ST_MST.precond) OR
   (_precond_method == MST_QR.precond) OR
   (_precond_method == MST_LDLt.precond) THEN
  // Create an MST for the preconditioner
  USE PanningTree = T
END IF

IF (_precond_method == ST.precond) OR
   (_precond_method == ST_MST.precond) THEN
  // Create a support tree for the preconditioner
  USE SupportTree = T
END IF

/* Allocate some arrays needed by the preconditioner, depending on which one we have. In F2x one
   needs to associate the matrix % Preconditioner with the correct preconditioning procedure, depending
   on the value of _precond_method. */

IF (_precond_method == MST_QR.precond) OR
   (_precond_method == MST_LDLt.precond) OR
   (_precond_method == diagonal.precond) OR
   (_precond_method == ST.precond) OR
   (_precond_method == ST_MST.precond) THEN
  _AllocateNodalArray (_nodes_resistances, 1.0, wp, _NON_NULL, "InitializeDualPreconditioner")
END IF

IF (_use_support_tree) THEN    // Initialize a support tree
  WRITE (message_log_unit, *) "Initializing a support tree for the network preconditioner"
END IF

BuildST: IF (_use_support_tree) THEN    // Initialize a support tree
  WRITE (message_log_unit, *) "Initializing a support tree for the network preconditioner"
IF (~ASSOCIATED(_support_tree)) THEN
  allocated_ST = T
  ALLOCATE (_support_tree, stat = alloc_status)
END IF

_delay_tree => _support_tree

IF (~ASSOCIATED(_ST_arrays)) THEN
  allocated_ST_arrays = T
  ALLOCATE (_ST_arrays, stat = alloc_status)
END IF

_AllocateNodalArray(_nodes_mapping, li_wp, ~NON_NULL, "InitializeDualPreconditioner")
_AllocateArcArray(_crossing_arcs_mask, Ti_wp, ~NON_NULL, "InitializeDualPreconditioner")

IF (_mapping_method ≡ CHACO_ordering) THEN
  IF (~ASSOCIATED(_chaco_mapping)) THEN
    ALLOCATE (_chaco_mapping, stat = alloc_status)
  END IF
  IF (ASSOCIATED(_nodes_coordinates)) THEN
    _chaco_mapping % coordinates => _nodes_coordinates, 1:n_nodes
  END IF
ELSE IF (_mapping_method ≡ SCOTCH_ordering) THEN
  IF (~ASSOCIATED(_scotch_mapping)) THEN
    ALLOCATE (_scotch_mapping, stat = alloc_status)
  END IF
END IF

END IF

_InitializeST: IF ((_mapping_method ≡ SCOTCH_ordering) || (_mapping_method ≡ CHACO_ordering) || (_mapping_method ≡ Matching_ordering)) THEN
  // _ST_graph needs to be associated
  IF (_support_special_nodes || ~rooted_graph) THEN
    // Support all the nodes
    _support_special_nodes = T  // When not rooted this is a must
    _ST_graph => graph  // For now I include the special arcs and nodes
    _ST_nodes_mapping => _nodes_mapping
    _ST_weights => dual_system % arcs_conductances
  ELSE  // We can skip the special nodes if we want to
    ALLOCATE (_ST_graph, stat = alloc_status)
    _ST_graph % heads_tails => heads_tails:1,2, 1:n_arcs
    _ST_graph % n_special_nodes = -1
    _ST_graph % n_special_arcs = -1
    _ST_graph % n_nodes = n_nodes
    _ST_graph % n_arcs = n_arcs
    _ST_nodes_mapping => _nodes_mapping, 1:n_nodes
    _ST_weights => dual_system % arcs_conductances, 1:n_arcs
    _nodes_mapping % n_special_nodes:0 = 0  // Special nodes are rooted
    _crossing_arcs_mask % n_special_arcs:0 = F:1_wp  // Just in case
  END IF
ELSE IF (_mapping_method ≡ CHACO_ordering) THEN
  //
END IF
END IF

END IF

IF (_mapping_method ≡ SCOTCH_ordering) THEN
  _scotch_mapping % graph => _ST_graph
  _scotch_mapping % ST_mapping => _ST_mapping
END IF
ELSE IF (_mapping_method ≡ CHACO_ordering) THEN

END IF

11
chaco_mapping % graph \to ST_graph
chaco_mapping % ST_mapping \to ST_mapping

END IF

IF (\text{Height} < 0) THEN
  \text{height} = \text{INT}(\log(\text{REAL}(n\_nodes)) / \log(\text{REAL}(\text{degree})), i\_wp)
IF (\text{support\_tree} \% \text{max\_height} > 0) THEN // Bound the height from above
  \text{height} = \text{MAX}(l_{i\_wp}, \text{MIN}(\text{support\_tree} \% \text{max\_height}, \text{height}))
ELSE
  \text{height} = \text{MAX}(l_{i\_wp}, \text{height})
END IF

n\_ST\_leaves = \text{degree}^\text{height}
n\_ST\_nodes = ((n\_ST\_leaves - 1) * \text{degree}) / (\text{degree} - 1)

AllocateArray(ST\_arcs\_resistances, 0, n\_ST\_nodes, 1.0, i\_wp, "NULL;
  "InitializeDualPreconditioner")
AllocateArray(ST\_potentials\_and\_flows, 0, n\_ST\_nodes, 1.0, i\_wp, "NULL;
  "InitializeDualPreconditioner")

SELECT CASE (mapping\_method)
case (SCOTCH\_ordering)
  scotch\_mapping % map\_graph = T
  scotch\_mapping % order\_graph = F
IF (ST\_architecture \equiv "") THEN
  WRITE (ST\_architecture, ") "leaf", \_height, \_height, " 1"
END IF
IF (ST\_strategy \equiv "") THEN
  ST\_strategy = "b{job=t, map=t, poli=S} || strat=m{low=(g x), asc=f{type=b}
  x, rat=0.95, type=h, vert=100}"
END IF
WRITE(message\_log\_unit, ") "Using SCOTCH partitioning with architecture ",
  TRIM(ST\_architecture), " and strategy ",TRIM(ST\_strategy)
call SCOTCH\_Initialize\_Mapping\_Ordering(mapping\_ordering = scotch\_mapping)

CASE (CHACO\_ordering) // Use CHACO to partition

CHACOSequence: IF (support\_tree \% CHACO\_sequence) THEN

chaco\_mapping % strategy % sequence = 1 // For now only sequencing will be supported
chaco\_mapping % strategy % rqi\_flag = CHACO\_Lanczos
chaco\_mapping % strategy % vmax = \text{MAX}(l_{i\_wp}, n\_nodes / 10_{i\_wp})
chaco\_mapping % architecture % adims\_tot = n\_ST\_leaves
WRITE(message\_log\_unit, ") "Using spectral CHACO to sequence the nodes into",
  n\_ST\_leaves, " slices."
ELSE

chaco\_mapping % strategy % sequence = 0
chaco\_mapping % architecture % architecture = CHACO\_meshID // A mesh architecture
chaco\_mapping % architecture % mesh\_dims\_1 = n\_ST\_leaves
IF (chaco\_mapping % strategy % global\_method \equiv CHACO\_dummy) THEN
  IF (ASSOCIATED(nodes\_coordinates)) THEN
    chaco\_mapping % strategy % global\_method = CHACO\_inertial
  ELSE

12
if (chaco_mapping strategy global_method = CHACO_linear)
end if
end if
if (chaco_mapping strategy local_method = CHACO_dummy) then
  chaco_mapping strategy local_method = CHACO_LocalKL
end if
if (chaco_mapping strategy ndims = 1)
if (chaco_mapping strategy vmax = 10) // Seems reasonable
  write (message_log_unit, *) "Using CHACO partitioning with 1D mesh of length ", n_ST_leafs
end if
endif CHACOSquence
end if
end select
end if BuildST
buildmST: if (use_spanning_tree) then // Initialize a spanning tree
  write (message_log_unit, *) "Initializing a spanning forest for the network preconditioner"
IF (~ASSOCIATED(_tree)) THEN
    _allocated_tree = T
    ALLOCATE (_tree)
END IF

_tree = _tree
// I assume here that the graph is connected!
_tree % graph = graph   // For these preconditioners the graph is the same
_tree % PIN = graph % PIN   // Copy the PIN
AllocateNodalArray(tree_arcs_excess_volages, 1.0_widget, ~NON_NULL, "InitializeDualPreconditioner")
AllocateNodalArray(tree_nodes_parents, l_widget, ~NON_NULL, "InitializeDualPreconditioner")
AllocateNodalArray(tree_nodes_cardinalities, l_widget, ~NON_NULL, "InitializeDualPreconditioner")
AllocateArray(tree_nodes_ordering, 0, n_nodes + n_special_nodes, l_widget, ~NON_NULL, "InitializeDualPreconditioner")
AllocateNodalArray(tree_nodes_orientations, l_widget, ~NON_NULL, "InitializeDualPreconditioner")
AllocateArcArray(tree_arcs_mask, l_widget, ~NON_NULL, "InitializeDualPreconditioner")

IF (~precond_method EQUIVALENT ST_MST_ precond) THEN
    /* We need to initialize the spanning forest now to a legal forest if reoptimization is desired. 
       Here I choose to start from every node in a tree by itself, even though it is quite possible to make
       an initial partitioning of the graph here. However, it seems this additional work can have
       little effect, since the same will be done on the first iteration of the dual solver anyway: */
    IF (~reoptimize_ST_MST) THEN   // Initialize the forest
        DO node = -n_special_nodes, n_nodes
            _tree.nodes.parents[node] = node
        END DO
        _tree.nodes.orientations = no_parent
    END IF
ELSE
    CALL BuildNetworkMST(arc_offset = tree % n_special_arcs, node_offset = tree % n_special_nodes,
                          heads_tails = tree % heads_tails, orientations = tree.nodes.orientations,
                          level_ordering = tree.nodes.ordering, parents = tree.nodes.parents,
                          cardinalities = tree.nodes.cardinalities, path_labels = tree.path_labels,
                          tree_mask = tree_arcs_mask, from_scratch = T, tree.type = any_tree)
END IF
END IF BuildMST

InitializeTAUCS: IF (~precond_method EQUIVALENT TAUCS_LLT_precond) OR (~precond_method EQUIVALENT TAUCS_MWB_precond) THEN
    IF (~ASSOCIATED(taucs_preconditioner)) THEN
        _allocated_taucs = T
        ALLOCATE (taucs_preconditioner, STAT = alloc_status)
    END IF
    taucs_preconditioner = _tau_cond
    taucs.graph = graph
    taucs.preconditioner % conductances = _arcs_conductances
    taucs.preconditioner % excess_potentials = _cg_residual_potentials
    taucs.preconditioner % excess_flows = _cg_residual_flows
    // As before, I eliminate/ground the dummy node
    IF (~NON_NULL(taucs.preconditioner % eliminated_nodes)) THEN

ALLOCATE (taucs_preconditioner % eliminated_nodes (1 : 1), STAT = alloc_status)

GroundGraph: IF (rooted_graph) THEN    // Eliminate only the dummy node
taucs_preconditioner % eliminated_nodes = (/ 0 /)
ELSE    // Choose the first special node as grounded
taucs_preconditioner % eliminated_nodes = (/ -n_special_nodes /)
END IF GroundGraph

END IF
taucs_preconditioner % diagonal_conductance = grounding_conductance
// Grounding arc for dummy node
IF (_precond_method == TAUCS_LLT_precond) THEN
    WRITE (message_log_unit, *) "Initializing a TAUCS incomplete LLt factorization
    preconditioner."
taucs_preconditioner % precond_method = TAUCS_LLt
taucs_preconditioner % factorization = TAUCS_incomplete_fact
ELSE
taucs_preconditioner % precond_method = TAUCS_MWB
taucs_preconditioner % factorization = TAUCS_complete_fact
    WRITE (message_log_unit, *) "Initializing a TAUCS MWB support-graph preconditioner."
END IF

CALL TAUCSInitializePreconditioner (taucs_preconditioner = taucs_preconditioner)

END IF InitializeTAUCS

CALL StopTimer (_precond_init_timer)

END SUBROUTINE InitializeDualPreconditioner

This code is used in section 1.0.0.1.

1.2.3 DestroyDualSolver

This routine will deallocate the arrays allocated by the previous routine. Here things can get tricky, for example, should I just disassociate the pointer or deallocate the corresponding targets? The approach here is to explicitly deallocate all arrays that could have been allocated by &InitializeDualSolver, other than the global array of dualSystems. This includes all temporaries used by the CG solver, but not the graph or MST itself and the arcs resistance/conductances. A crucial point is that I do not deallocate dual_system itself, since I cannot be sure it was allocated by &ALLOCATE by the user (it could point to a static variable). These kinds of tricky things are up to the user, because only he knows what exactly was done.

(DestroyDualSolver 1.2.3) ≡

SUBROUTINE DestroyDualSolver (dual_system)
    _TYPE (Dual_Network_System), INTENT (INOUT), TARGET :: dual_system

    INTEGER :: alloc_status
    _TYPE (CG_Solver), POINTER :: solver
    _TYPE (Network_Arrays), POINTER :: arrays

    solver => dual_system % solver
    arrays => _arrays

UsePCG: IF (_solver_method == dual_network_pcg) THEN
IF (allocated_solver) THEN
  solver $\Rightarrow$ dual_system % solver
  _DealocateArray(_cg_nodes_flows, 1.0_wp, _NON_NULL)
  _DealocateArray(_cg_nodes_potentials, 1.0_wp, _NON_NULL)
  _DealocateArray(_cg_residual_flows, 1.0_wp, _NON_NULL)
  _DealocateArray(_cg_residual_potentials, 1.0_wp, _NON_NULL)
  DEALLOCATE (dual_system % solver, stat = alloc_status)
END IF

IF (ASSOCIATED(_matrix)) THEN
  DEALLOCATE (_matrix, stat = alloc_status)
END IF

ELSE UsePCG

IF (allocated_ordering) THEN
  _DealocateArray(_nodes_renumbering, 1.0_wp, _NON_NULL, "DestroyDualSolver")
  _DealocateArray(_nodes_reordering, 1.0_wp, _NON_NULL, "DestroyDualSolver")
  DEALLOCATE (_graph_ordering, stat = alloc_status)
END IF

CALL TAUCS_DestroyPreconditioner(taus_preconditioner = _taucs_factorization)
IF (allocated_factorization) THEN
  DEALLOCATE (_taucs_factorization, stat = alloc_status)
END IF

END IF UsePCG

END SUBROUTINE DestroyDualSolver

This code is used in section 1.0.0.1.

1.2.4 Destroying the preconditioner

This routine should also be split to make another routine FreePreconditioner that will release only the numerical portions of the preconditioners.

SUBROUTINE DestroyDualPreconditioner (dual_system)
  _TYPE (Dual_Network_System), INTENT (INOUT), TARGET :: dual_system
  _TYPE (Spanning_Tree), POINTER :: tree
  _TYPE (Support_Tree_Preconditioner), POINTER :: support_tree
  _TYPE (TAUCS_Network_Preconditioner), POINTER :: taus_preconditioner
  INTEGER :: alloc_status

  _DealocateArray(_nodes_resistances, 1.0_wp, _NON_NULL)
  _DealocateArray(_nodes_multipliers, 1.0_wp, _NON_NULL)
  _DealocateArray(_tree_arcs_excess_voltages, 1.0_wp, _NON_NULL)

  tree $\Rightarrow$ _tree
IF (allocated_tree) THEN
   DeallocateArray(_tree_nodes_parents, 1,wp, NULL)
   DeallocateArray(_tree_nodes_cardinalities, 1,wp, NULL)
   DeallocateArray(_tree_nodes_orientations, 1,byte, NULL)
   DeallocateArray(_tree_nodes_ordering, 1,wp, NULL)
   DeallocateArray(_tree_path_labels, 1,wp, NULL)
   DeallocateArray(_tree_arcs_mask, 1,wp, NULL)
END IF

support_tree ⇒ _support_tree
IF (use_support_tree) THEN
   SELECT CASE (_mapping_method)
   CASE (SCOTCH_ordering)
      CALL SCOTCH_DestroyMappingOrdering(mapping_ordering = _scotch_mapping)
   CASE (CHACO_ordering)
      CALL CHACO_DestroyMapping(mapping_partitioning = _chaco_mapping)
   ENDSSELECT
IF (allocated_ST_arrays) THEN
   DeallocateArray(_nodes_mapping, 1,wp, _NON_NULL)
   DeallocateArray(_crossing_arcs_mask, T,wp, _NON_NULL)
   DeallocateArray(_ST_arcs_resistances, 1,wp, _NON_NULL)
   DeallocateArray(_ST_potentials_and_flows, 1,wp, _NON_NULL)
END IF
END IF
IF (allocated_special_nodes) DEALLOCATE (_ST_graph, stat = alloc_status)
END IF
IF (allocated_ST) THEN
   DEALLOCATE (_support_tree, stat = alloc_status)
END IF

taucs_preconditioner ⇒ _taucs_preconditioner
IF (ASSOCIATED(taucs_preconditioner)) THEN
   DEALLOCATE (taucs_preconditioner % eliminated_nodes)
   CALL TAUCS_DestroyPreconditioner(taucs_preconditioner = taucs_preconditioner)
IF (NOT ASSOCIATED(_taucs_graph, _graph)) DEALLOCATE (_taucs_graph, stat = alloc_status)
END IF
IF (allocated_taucs) DEALLOCATE (_taucs_preconditioner, stat = alloc_status)
END SUBROUTINE DestroyDualPreconditioner

This code is used in section 1.0.0.1.

1.3 Solving the Dual Newton System

The dual Newton system of linear equations encountered in convex network optimization has the form:

\[(AHAT')dx = -G\]

where \(H\) is a diagonal hessian arc-length matrix and \(G\) is a node-length gradient vector, and \(dx\) is the Newton step size. Here \(H = D_A\) is determined by \(arcs\_conductances\), \(dx\) is given via \(nodes\_excess\_potentials\) and \(-G\) is given via \(nodes\_excess\_flows\).
The routine `SolveDualSystem_PCG` solves this system using PCG on the Newton system itself, with various choices of dual preconditioners. A valid system (i.e., associated with a valid dual system via `dualSystems(dual_system %, psi) % system`) needs to be passed here. There is also a routine `SolveDualSystemLLt` that uses direct complete Cholesky factorization from TAUCS:

### 1.3.1 Cholesky Factorization

\[
\langle \text{SolveDualSystemLLt 1.3.1.1} \rangle \equiv
\]

```fortran
SUBROUTINE SolveDualSystem_LLt(dual_system)
  IMPLICIT NONE
  _TYPE (Dual_Network_System), INTENT (INOUT), TARGET :: dual_system
  _TYPE (Directed_Graph), POINTER :: graph
  _TYPE (Network_Arrays), POINTER :: arrays
  _TYPE (TAUCS_Network_Preconditioner), POINTER :: taucs_factorization
  INTEGER (KIND = i_up) :: n_special_nodes, n_nodes, n_special_arcs, n_arcs, node, arc
  graph => _graph
  arrays => _arrays
  taucs_factorization => _taucs_factorization
  n_special_nodes = _n_special_nodes
  n_nodes = _n_nodes
  n_special_arcs = _n_special_arcs
  n_arcs = _n_arcs
  CALL StartTimer (Jlt_fact_timer)
  CALL TAUCS_CreatePreconditioner (taucs_preconditioner = taucs_factorization)
  CALL StopTimer (Jlt_fact_timer)
  CALL StartTimer (Jlt_solve_timer)
  CALL TAUCS_ApplyPreconditioner (taucs_preconditioner = taucs_factorization)
  CALL StopTimer (Jlt_solve_timer)
  CALL TAUCS_FreePreconditioner (taucs_preconditioner = taucs_factorization)
END SUBROUTINE SolveDualSystem_LLt
```

This code is used in section 1.0.0.1.

### 1.3.2 Preconditioned Conjugate Gradient

\[
\langle \text{SolveDualSystemPCG 1.3.2.1} \rangle \equiv
\]

```fortran
SUBROUTINE SolveDualSystem_PCG(dual_system)
```

18
IMPLICIT NONE

_TYPE (Dual_Network_System), INTENT (INOUT), TARGET :: dual_system
_TYPE (Directed_Graph), POINTER :: graph
_TYPE (Spanning_Tree), POINTER :: tree
_TYPE (Support_Tree_Preconditioner), POINTER :: support_tree
_TYPE (CG_System), POINTER :: system
_TYPE (CG_Solver), POINTER :: solver
_TYPE (Network_Arrays), POINTER :: arrays

INTEGER (KIND = i_wp) :: n_special_nodes, n_nodes, n_special_arcs, n_arcs, node, arc

IF (PIN < LBND(dual_system, i_wp)) THEN
  CALL CriticalError(message = "Not enough dual system handles available or PIN out ofange", caller = "InitializeDualSolver")
  RETURN
ELSE
  IF (ASSOCIATED (dual_system(PIN))) THEN
    CALL CriticalError(message = "Dual system handle already taken", caller = "InitializeDualSolver")
    RETURN
  ELSE
    _dual_system(PIN) = dual_system  // This is the system-emulation strategy in F95
  END IF
END IF

_PIN = PIN  // Save the PIN because the multiplication and preconditioners need it
_PIN = PIN

graph => _graph
tree => _tree
support_tree => _support_tree
system => _system
solver => _solver
arrays => _arrays

n_special_nodes = _n_special_nodes
n_nodes = _n_nodes
n_special_arcs = _n_special_arcs
n_arcs = _n_arcs

/* Initialize the preconditioner (i.e. the diagonal or basis factors, or partitioning, or MST) before
proceeding with the solution process (this could have been a separate routine?). Since we have a
fair number of preconditioners, this is somewhat involved: */
IF (_precond_method = diagonal_precond) THEN
  // The easiest preconditioner to prepare is the diagonal
  CALL StartTimer (_precond_fact_timer)
  CALL Diagonal_ADAt (heads_tails = heads_tails, node_offset = n_special_nodes,
                 arcs_conductances = arcs_conductances, nodes_resistances = _nodes_resistances)
  CALL StopTimer (_precond_fact_timer)
ENDIF

UpdateST: IF (_use_support_tree) THEN
  // We need to compute the partitioning
  IF ((_mapping_method = SCOTCH_ordering) || (_mapping_method = CHACO_ordering)) THEN
    CALL StartTimer (_precond_opt_timer)
    SELECT CASE (_mapping_method)
    CASE (SCOTCH_ordering)

19
CALL SCOTCH_ComputeMappingOrdering(mapping_ordering = scotch_mapping,
    mapping_offset = (n_ST_nodes - n_ST_leafs) + 1)
CASE (CHACO_ordering)
    CALL CHACO_ComputeMapping(mapping_partitioning = chaco_mapping,
        mapping_offset = (n_ST_nodes - n_ST_leafs) + 1)
CASE (Matching_ordering)
    CALL ST_MatchingNodesMapping(node_offset = ST_graph % n_special_nodes,
        arc_offset = ST_graph % n_special_arcs, heads_tails = ST_graph % heads_tails,
        arcs_conductances = ST_weights, nodes_mapping = ST_nodes_mapping,
        n_components = n_ST_leafs, mapping_offset = (n_ST_nodes - n_ST_leafs) + 1,
        heavy_matching = T)
ENDSELECT

// The rooting node must always be grounded in some cases:
IF (ground_dummy ^ rooted_graph) nodes_mapping[0] = 0
CALL ST_ArcsMask(node_offset = ST_graph % n_special_nodes,
    arc_offset = ST_graph % n_special_arcs, heads_tails = ST_graph % heads_tails,
    arcs_mask = crossing_arcs_mask % ST_graph % n_special_arcs, ST_graph % n_arcs,
    nodes_mapping = ST_nodes_mapping, mask_internal_arcs = T)
CALL StopTimer(_precond_reopt_timer)
END IF

// Now calculate the resistances for the support tree
CALL StartTimer(_precond_fact_timer)
IF (_precond_method $e ST_MST_precond) THEN
    // No nodes resistances needed--calculated later on
    CALL ST_PropagateArcsConductances(height = _height, degree = _degree,
        arc_offset = n_special_arcs, node_offset = n_special_nodes, heads_tails = _heads_tails,
        arcs_conductances = arcs_conductances, nodes_mapping = nodes_mapping,
        ST_arcs_resistances = ST_arcs_resistances, ST_nodes_temp = ST_potentials_and_flows)
ELSE
    CALL ST_PropagateArcsConductances(height = _height, degree = _degree,
        arc_offset = n_special_arcs, node_offset = n_special_nodes, heads_tails = _heads_tails,
        arcs_conductances = arcs_conductances, nodes_mapping = nodes_mapping,
        nodes_resistances = nodes_resistances, ST_arcs_resistances = ST_arcs_resistances,
        ST_nodes_temp = ST_potentials_and_flows)
END IF
CALL StopTimer(_precond_fact_timer)
ENDIF

ENDIF UpdateST

UpdateMST: IF (_use_spanning_tree) THEN // Rebuild the MST for the preconditioner
    IF (_precond_method $e ST_MST_precond) THEN
        @if 0
        _InitGraphics("Supported MST with nodes partitioning/mapping")
        _PlotNodesMapping
        _EndGraphics
        @endif
    ENDIF
END IF

20
CALL StartTimer(_precond_reopt_timer)

ReoptimizeMST: IF (_reoptimize_ST_MST) THEN

// We need to break the parenthesis relations that have crossed across partitions
DO node = n_special_nodes
IF (~crossing_arcs_mask(_tree_nodes_parent[node])) THEN
    _tree_nodes_orientations[node] = no_parent
    // Should we zero the parent as well?
END IF
END DO
END IF ReoptimizeMST

CALL BuildNetworkMST (arc_offset = tree % n_special_arcs,
node_offset = tree % n_special_nodes, heads_tails = tree % heads_tails,
arc_mask = crossing_arcs_mask,
orientations = _tree_nodes_orientations, level_ordering = _tree_nodes_ordering,
parents = _tree_nodes_parent, cardinalities = _tree_nodes_cardinalities,
path_labels = _tree_path_labels, tree_mask = _tree_arcs_mask,
from_scratch = _reoptimize_ST_MST, defragment_ordering = F, tree_type = any_tree)

CALL StopTimer(_precond_reopt_timer)

ELSE  // No mask for the arcs is needed
@if 0
    _InitGraphics("Rooted MST")
@endif
ENDIF StartTimer(_precond_reopt_timer)

CALL BuildNetworkMST (arc_offset = tree % n_special_arcs,
node_offset = tree % n_special_nodes, heads_tails = tree % heads_tails,
arc_weights = arcs_conductances,
orientations = _tree_nodes_orientations, level_ordering = _tree_nodes_ordering,
parents = _tree_nodes_parent, cardinalities = _tree_nodes_cardinalities,
path_labels = _tree_path_labels, tree_mask = _tree_arcs_mask, from_scratch = F,
    defragment_ordering = T, tree_type = max_cost_tree)

CALL StopTimer(_precond_reopt_timer)

ENDIF

@if 0
    _PlotMST
    _EndGraphics
@endif

CALL StartTimer(_precond_fact_timer)
IF (_precond_method ≡ MST_QR_precond) THEN  // Factorize the matrices if needed
    _IncompleteFactorization(QR)
ELSE IF (_precond_method ≡ MST_LDLt_precond) THEN
    _IncompleteFactorization(LDLt)
ENDIF
END IF
  CALL StopTimer(_precond_fact_timer)
END IF UpdateMST

CALL StartTimer(_precond_fact_timer)
UpdateTAUCS: IF ((_precond_method => TAUCS_LLt_precond) | (_precond_method => TAUCS_MWB_precond)) THEN
  // TEMPORARY: This is for overhead testing purposes:
  IF (_taucs_preconditioner % dropout <= 0.0, _w) THEN // Use complete factorization
    _taucs_preconditioner % factorization = TAUCS_complete_fact
  ELSE // Use point-incomplete factorization
    _taucs_preconditioner % factorization = TAUCS_incomplete_fact
  END IF
  CALL TAUCS_CreatePreconditioner(taucs_preconditioner = _taucs_preconditioner)
END IF UpdateTAUCS
CALL StopTimer(_precond_fact_timer)

/* Now actually solve the system using PCG: */
CALL StartTimer(_pcg_timer)
SELECT CASE (_precond_method)
  CASE (diagonal_precond) 
    _CallPCG(DualDiagonalPrecond)
  CASE (MST_BHBt_precond) 
    _CallPCG(DualMSTPrecond)
  CASE (MST_QR_precond, MST_LDLt_precond)
    _CallPCG(DualFactorizationPrecond)
  CASE (ST_precond)
    _CallPCG(DualSupportTreePrecond)
  CASE (TAUCS_LLt_precond, TAUCS_MWB_precond)
    _CallPCG(DualTAUCSPrecond)
  CASE (ST_MST_precond)
    _CallPCG(DualSupportedMSTPrecond)
  CASE DEFAULT // No preconditioning
    CALL PreconditionedConjugateGradient(system = system,
      MatrixVectorMultiplication = DualMatrixMultiply)
ENDSELECT
CALL StopTimer(_pcg_timer)

IF (solver % convergence => CG_converged) THEN
  WRITE(messageLog_unit, *) "-- CG converged after ", solver % n_iterations, " iterations with errors ", solver % errors
ELSE
  WRITE(messageLog_unit, *) "-- CG did NOT converge.", " Last recorded errors are ", solver % errors
END IF

IF ((_precond_method => TAUCS_LLt_precond) | (_precond_method => TAUCS_MWB_precond)) THEN
  CALL TAUCS_FreePreconditioner(taucs_preconditioner = _taucs_preconditioner)
END IF

NULLIFY (_dual_system(pin)) // Free the temporary pointer without deallocating anything

END SUBROUTINE SolveDualSystemPCG

See also sections 1.4.1.1, 1.4.2.1, 1.4.3.1, 1.4.4.1, 1.4.5.1, 1.4.6.1, and 1.4.7.1.

This code is used in section 1.0.0.1.

22
1.4 Internal Matrix-Vector Multiplies and Preconditioners

As discussed in \textit{Conjugate Gradient}, the decision to use pointer and allocatable components in this implementation has its cost in performance. To avoid these kind of penalties, I coded all the extensive calculation routines in \textit{Graph Algorithms} and \textit{Network Matrix Operations} in low-level array-based syntax, and here we have wrappers that call these routines passing the targets of the pointers as actual arguments. Since according to the standard it is the user's responsibility to insure that there is no aliasing between actual arguments, no compiler should generate copy-in for these actual pointer arguments. If this is the case, the whole system must be trashed and rebuilt from scratch.

1.4.1 Matrix-Vector Multiplication

\begin{verbatim}
(SolveDualSystemPCG 1.3.21) +≡
SUBROUTINE DualMatrixMultiply(system)  // Matrix-vector multiplication (\(AD_A A^T\))x
  IMPLICIT NONE
  TYPE (CG_System), INTENT (INOUT) :: system  // class (CG_System) in F2x
  TYPE (Dual_Network_System), POINTER :: dual_system
  dual_system = _dual_system(system % PIN)  // The system we are working on-F95 workaround
  CALL Multiply_ADA_Dual(heads_tails = dual_system % _heads_tails,
    node_offset = dual_system % _n_special_nodes,
  
\end{verbatim}
arcs, conductances = dual System % arcs, conductances,
node, flows = system % cg, nodes, flows, nodes, potentials = system % cg, nodes, potentials

/* If we want to explicitly ground the network (numerically problematic for some reason?), this
can be done in CG by adding a diagonal correction to the Laplacian for node 0 (just as when
using direct solvers): */
IF (ground, dummy)
    system % cg, nodes, flows, = system % cg, nodes, flows, + grounding, conductance * system %
    cg, nodes, potentials
END SUBROUTINE DualMatrixMultiply

1.4.2 Diagonal Preconditioner

(SolveDualSystemPCG 1.3.2.1) =>

SUBROUTINE DualDiagonalPrecond (system)   // Diagonal preconditioning
    IMPLICIT NONE
    TYPE (CG_System), INTENT (INOUT) :: system       // CLASS (CG_System) in F2x
    // or better TYPE (DualNetwork_System)

    TYPE (DualNetwork_System), POINTER :: dual system

dual system => dual system (system % PIN)       // The system we are working on

CALL Vector Multiplication (first = _nodes, resistances, second = _system % cg, residual, flows,
    product = _system % cg, residual, potentials)
    // Instead of the possibly aliased cg, residual, potentials = _nodes, resistances * cg, residual, flows

    // We can ground the dummy node via a potential shift:
    IF (ground, dummy) THEN                          // Explicitly ground node 0
        CALL VectorShift (vector = _system % cg, residual, potentials,
            shift = _system % cg, residual, potentials,)
    END IF

@# if PLOT_PRECOND
    InitGraphics("Diagonal precond excess potentials")
    PlotNodesPotentials(_system % cg, residual, potentials)
    EndGraphics
@# endif

END SUBROUTINE DualDiagonalPrecond

1.4.3 Maximal Spanning Tree Basis Preconditioner
SUBROUTINE DualMSTPrecond(system)  // MST-based preconditioning
IMPLICIT NONE
_TYPE (CG_System), INTENT (INOUT) :: system  // CLASS (CG_System) in F2x
    // or better TYPE (Dual_Network_System)
_TYPE (Dual_Network_System), POINTER :: dual_system
_TYPE (Spanning_Tree), POINTER :: tree
_TYPE (CG_Solver), POINTER :: solver
_TYPE (Network_Arrays), POINTER :: arrays
INTEGER (KIND = i_up) :: lb, ub  // Bounds for nodal arrays
dual_system => _dual_system(system % PIN)  // The system we are working on
tree => _tree
solver => _system % solver
arrays => _arrays
/* For now these must be the same as n_special_nodes and n_nodes: */
lb = -tree % n_special_nodes
ub = tree % n_nodes
cg_residual_potentials(tree_nodes->ordering, 0, 0, exp)  // The root is grounded!
CALL BasicDualNewtonSystem(node_offset = tree % n_special_nodes,
    arc_offset = tree % n_special_arcs, heads_tails = tree % heads_tails,
    orientations = tree->orientations, tree_mask = tree->mask,
    parents = tree->parents, level_ordering = tree->ordering,
    tree_arcs_excess_voltages = tree->arcs_excess_voltages,
    nodes_excess_potentials = cg_residual_potentials, nodes_excess_flow = cg_residual_flow,
    arcs_conductances = arcs_conductances)
if (ground_dummy) THEN  // Explicit grounding
    CALL VectorShift(vector = cg_residual_potentials, shift = -cg_residual_potentials)
END IF
@# if PLOT_PRECOND
    InitGraphics("MST precond excess potentials")
    _PlotNodesPotentials(cg_residual_potentials)
    _EndGraphics
@# endif
END SUBROUTINE DualMSTPrecond

1.4.4 Incomplete MST-based factorizations

SUBROUTINE DualFactorizationPrecond(system)  // Either QR or Cholesky MST factorization
IMPLICIT NONE
_TYPE (CG_System), INTENT (IN) :: system  // CLASS (CG_System) in F2x
    // but really of TYPE (Dual_Network_System)
1.4.5 Support-Tree Preconditioners
CALL ST_DualNewtonPreconditioner(height = _height, degree = _degree,
  node_offset = dual_system % n_special_nodes, nodes_mapping = _nodes_mapping,
  nodes_excess_flows = cg_residual_flows, nodes_excess_potentials = cg_residual_potentials,
  nodes_resistances = _nodes_resistances, ST_potentials_and_flows = ST_potentials_and_flows,
  ST_arcs_resistances = ST_arcs_resistances)

IF(_ground_dummy) THEN  // There are ways to avoid this, but this is the easiest:
  CALL VectorShift(vector = cg_residual_potentials, shift = -cg_residual_potentials_0)
END IF

@#if PLOT_PRECOND
  _InitGraphics("ST precond excess potentials")
  _PlotNodesPotentials(cg_residual_potentials)
  _EndGraphics
@#endif

END SUBROUTINE DualSupportTreePrecond

1.4.6 Partitioned and Supported MST Preconditioners

These are novel inventions!

(SolveDualSystemPCG 1.3.21) +≡

SUBROUTINE DualSupportedMSTPrecond(system)  // Combination of MST and ST preconditioning
  IMPLICIT NONE
  _TYPE (CG_System), INTENT (INOUT) :: system  // CLASS (CG_System) in F2x
    // or better TYPE (Dual_Network_System)
  _TYPE (Dual_Network_System), POINTER :: dual_system
  _TYPE (Network_Arrays), POINTER :: arrays
  _TYPE (Directed_Graph), POINTER :: graph
  _TYPE (Spanning_Tree), POINTER :: tree
  _TYPE (Support_Tree_Preconditioner), POINTER :: support_tree
  _TYPE (CG_Solver), POINTER :: solver
  INTEGER :: node, arc

dual_system = _dual_system(system % PIN)  // The system we are working on
arrays = _arrays
solver = _system % solver
graph = _graph
support_tree = _support_tree
tree = _tree

CALL ST_DualNewtonPreconditioner(height = _height, degree = _degree,
  node_offset = _n_special_nodes, nodes_mapping = _nodes_mapping,
  nodes_excess_flows = cg_residual_flows, ST_potentials_and_flows = ST_potentials_and_flows,
  ST_arcs_resistances = ST_arcs_resistances)
  // There is no need to initialize cg_excess_potentials for the root nodes this time:

27
CALL BasicDualNewtonSystem(node_offset = tree % _n_special_nodes,
    arc_offset = tree % _n_special_arcs, heads_tails = tree % _heads_tails,
    orientations = _tree_nodes_orientations, parents = _tree_nodes_parents,
    level_ordering = _tree_nodes_ordering, tree_mask = _tree_arcs_mask,
    nodes_excess_potentials = _cg_residual_potentials, nodes_excess_flows = _cg_residual_flows,
    tree_arcs_excess_voltages = _tree_arcs_excess_voltages, nodes_resistances = _nodes_resistances,
    nodes_multipliers = _nodes_multipliers)

    // Now we need to offset the potentials to account for the rooting arcs properly:
    _cg_residual_potentials = _cg_residual_potentials + _ST_potentials_and_flows(nodes_mapping)

IF (_ground_dummy) THEN // There are ways to avoid this, but this is the easiest:
    CALL VectorShift(vector = _cg_residual_potentials, shift = -_cg_residual_potentials_0)
END IF

@#if PLOT_PRECOND
    _InitGraphics("Supported MST precondition excess potentials")
    _PlotNodesPotentials(_cg_residual_potentials)
    _EndGraphics
@#endif

END SUBROUTINE DualSupportedMSTPrecond

1.4.7 Incomplete-Cholesky factorization preconditioners

(SolveDualSystemPCG 1.3.2.1) +==

SUBROUTINE DualTAUCSPrecond(system)    // Diagonal preconditioning
    IMPLICIT NONE
    TYPE (CG_System), INTENT (INOUT) :: system    // CLASS (CG_System) in F2x
    // or better TYPE (Dual_Network_System)
    TYPE (Dual_Network_System), POINTER :: dual_system
    TYPE (TAUCS_Network_Preconditioner), POINTER :: taucs_preconditioner
dual_system  =>  _dual_system(system % PIN)    // The system we are working on
    taucs_preconditioner  =>  _taucs_preconditioner

    CALL TAUCS_ApplyPreconditioner(taucs_preconditioner = taucs_preconditioner)

IF (_ground_dummy) THEN // We already grounded this node
    // What if the user specified eliminated_nodes and node 0 was not eliminated?
END IF

@#if PLOT_PRECOND
    _InitGraphics("TAUCS precondition excess potentials")
    _PlotNodesPotentials(system % _cg_residual_potentials)
    _EndGraphics
@#endif

END SUBROUTINE DualTAUCSPrecond

28
1.5 Debugging Plotting Macros

"WEAVE.f90" 1.5.0.1 ≡
@m _InitGraphics(title)
    CALL InitNetworkGraphics(file = "Preconditioner.png", file_type = "CONS",
        plot_title = (/ title /), page_size = (/ 5000, 5000 /), label_format = "4E1",
        color_table = "RAIN",
        colorbar_position = "Horizontal", axis_labels_format = (/ "NONE", "NONE", "NONE", "NONE" /))
@m _EndGraphics
    CALL EndNetworkGraphics()
@m _PlotMST CALL PlotNetwork2D (heads_tails = _tree % heads_tails ; 1 : node_offset=-1,
        node_coords = _nodes_coordinates ; 1 : arc_mask=_tree_arcs_mask 1 : arc_values=(! (1.0, arc = 1,
        _tree % n_arcs ) /), arc_size_range = (/ -HUGE(1.0_wp) / 20, HUGE(1.0_wp) / 10 /), vector_type = 0 )
@m _PlotNodesMapping CALL PlotNetwork2D (heads_tails = _heads_tails ; 1 : node_offset=-1,
        node_coords = _nodes_coordinates ; 1 : node_values=REAL(_nodes_mapping 1 :
        ), node_size_range = (/ -HUGE(1.0_wp) / 20, HUGE(1.0_wp) / 10 /), vector_type = 0 )
@m _PlotNodesPotentials(nodes_potentials)
    CALL PlotNetwork2D (heads_tails = dual_system % heads_tails ; 1 : node_offset=-1,
        node_coords = dual_system % _nodes_coordinates ; 1 : node_values=nodes_potentials 1 :
        , node_size_range = (/ -HUGE(1.0_wp) / 20, HUGE(1.0_wp) / 10 /), vector_type = 0 )
CASE_TYPE TYPE
  _TYPE TYPE
  _NULL > NULL()
PRIVATE
_SIZE(array, _kind, ...)
  _IF (#0, 0, INT(_SIZE(array, KIND=_kind), INT(_SIZE(array, #), KIND=_kind))
_MAXLOC(array, _kind, ...)
  _IF (#0, 0, INT(_MAXLOC(array, KIND=_kind), INT(_MAXLOC(array, #), KIND=_kind))
_MINLOC(array, _kind, ...)
  _IF (#0, 0, INT(_MINLOC(array, KIND=_kind), INT(_MINLOC(array, #), KIND=_kind))
_LBOUND(array, _kind, ...)
  _IF (#0, 0, INT(_LBOUND(array, DIM=1), KIND=_kind),
      INT(_LBOUND(array, #), KIND=_kind))
_UBOUND(array, _kind, ...)
  _IF (#0, 0, INT(_UBOUND(array, DIM=1), KIND=_kind),
      INT(_UBOUND(array, #), KIND=_kind))
_INTERFACE(generic_name, ...)
  INTERFACE generic_name MODULE PROCEDURE #.
END INTERFACE generic_name
Declare_i_word(...)
  INTEGER :: #.
Declare_j_wp(...)
  INTEGER (KIND = i_wp) :: #.
Declare_r_wp(...)
  REAL (KIND = r_wp) :: #.
Declare_r_sp(...)
  REAL (KIND = r_sp) :: #.
Declare_r_dp(...)
  REAL (KIND = r_dp) :: #.
FullExtent(rank) :: DO (DIM, 2, _rank) { ; }