Main Input/Output Parameters for SSCNO

Aleksandar Donev
August 2001

1 Derived Type Hierarchy

All input parameters to the higher-level SSCNO routines are derived Fortran 95 types. These have a complicated hierarchy, and have default initial values for the most part. Certain components of these derived types must be set to give the input to SSCNO, these are marked with “Compulsory input” below. Others do not have to, but I highly recommend that you do set them to good values for your particular problem (since I have no good defaults), and these are marked with “Input”. Some can be changed by the user before calling the SSCNO routines to provide additional functionality or fine-tune the performance, and these are marked with “Optional input”.

Below I give a summary of the hierarchy main data-types in SSCNO along with their default and possible values (which are usually integer parameters declared in the SSCNO modules). Please note that some of the components are pointers, initially nullified, so that you must explicitly allocate them before trying to change any of the values in them. Also, notice that the SSCNO routines do not initialize the values of these parameters to their defaults (this would be a complex task). By using pointers in Fortran 95, you can make the compiler do that—just deallocate the old Type (DualNetworkProblem) and allocate a brand new one, which will have correctly initialized values.

Finally, note that not all components are shown below—I tried to isolate the ones the user might want to or should change. The other components are still accessible, but should probably not be changed hastily!

1.1 Type DualNetworkProblem

This is the top-most type in SSCNO at the moment. It provides a non-linear strictly convex network optimization problem and gets certain results back:

"WEAVE.f90" 1.1 ≡

 TYPE DualNetworkProblem
   INTEGER :: PIN = 1   // Input
   INTEGER :: method = SSCNO_TDN   // Input, SSCNO_TDN or SSCNO_TSQP
   INTEGER (KIND = i_up) :: n_fixed_potentials = 0   // Input

1
1.2 Type Network_Problem

This type isolates the components that are in common to any network optimization problem, regardless of the method of solution (even though SSCNO only provides dual-based methods for now), such as the underlying graph, the arc and nodal vectors, cost function, etc. The structure of these will improve once F2k becomes standardized. Here I allow for Fortran 95 compilation mode by allowing dynamic arrays to be either POINTER or ALLOCATABLE via the macro DYNAMIC and its associated utility macros:

"WEAVE.f90":

```
TYPE Network_Problem  // extensible in F2k
  _TYPE (Directed_Graph) :: graph
  _TYPE (Directed_Graph) :: subgraph, special_subgraph
  _TYPE (Network_SC_Cost) :: cost_function
  _TYPE (Network_Arrays) :: arrays
ENDTYPE  // Network_Problem

TYPE Directed_Graph  // extensible in F2k
  INTEGER :: n_nodes = 0, n_arcs = 0, n_special_nodes = 0, n_special_arcs = 0
    // Compulsory input
  INTEGER (KIND = i_wp), DIMENSION (:,:), POINTER :: heads, tails = NULL
    // Compulsory input
ENDTYPE  // Directed_Graph
```
1.3 Type Dual Line Search

This type deals with the line-search phase of the dual algorithms in SSCNO. These are far from perfected or even finished as of yet, so there aren’t too many options here:

"WEAVE.F90" 1.3 ≡

1.4 Type Dual Network System
This is the type associated with the solver for a random-resistor (quadratic program) solver. Since most of the efficiency depends critically on the values in this derived type, close attention should be paid to some of these options. Two main methods are provided, direct Cholesky Factorization, and iterative Preconditioned Conjugate Gradients, which are separate data-types documented below:

"WEAVE.f90" 1.4 

TYPE Dual_Network_System
  INTEGER :: method = dual_network_LLt // Input, dual_network_LLt or dual_network_PCG
  _TYPE (CG_System) :: system // extends (CG_System) in F2x
  _TYPE (Dual_Network_Preconditioner) :: preconditioner
  _TYPE (Dual_Network_Factorization) :: factorization
  _TYPE (Dual_System_Timers) :: timers // Optional input
ENDTYPE // Dual_Network_System

TYPE Dual_System_Timers
  _TYPE (CG_Timers) :: pch_timers // Optional input
  INTEGER :: precond_initialization_timer = -1, precond_reoptimization_timer = -1,
              precond_factorization_timer = -1, pch_timer = -1, llt_ordering_timer = -1,
              llt_factorization_timer = -1, llt_solver_timer = -1
ENDTYPE // Dual_System_Timers

TYPE CG_Timers
  INTEGER :: precond_factorization_timer = -1, dot_timer = -1, vector_timer = -1,
             multiplication_timer = -1
ENDTYPE // CG_Timers

TYPE CG_System // extensible in F2x
  _TYPE (SPD_Matrix), POINTER :: matrix = NULL // CLASS (SPD_Matrix) in F2x
  _TYPE (CG_Solver), POINTER :: solver = NULL // CLASS (CG_Solver) in F2x
ENDTYPE // CG_System

TYPE CG_Solver // extensible in F2x
  INTEGER :: max_iterations = 0 // Optional input
  INTEGER :: log_unit = 0 // Optional input
  REAL (KIND = r_wp), DIMENSION (:, DYNAMIC :: _NULLIFIED (cg_Vx), _NULLIFIED (cg_Vy),
                                         _NULLIFIED (cg_Vz), _NULLIFIED (cg_residuals) // Optional input
ENDTYPE // CG_Solver

1.4.1 Type Dual_Network_Factorization

This type deals with direct solvers, most useful for 2D problems. It shares some of the data-types with the iterative preconditioners, simply because the TAUCS and SCOTCH libraries can be used in both. The CHACO library is only used in the preconditioners for now.

"WEAVE.f90" 1.4.1 

TYPE Dual_Network_Factorization
  INTEGER :: factorization = TAUCS_fact_mf // Optional input
     // TAUCS_fact_LLt, TAUCS_fact_Llt or TAUCS_fact_mf
integer : ordering = SCOTCH_fact_ordering  // Input
   // no_fact_ordering, TAU_CSFactOrdering or SCOTCHFactOrdering
   _TYPE (SCOTCH_Mapping_Ordering), pointer :: scotch_ordering = NULL
   _TYPE (TAUCS_Network_Preconditioner), pointer :: taucs_factorization = NULL
ENDTYPE  // Dual_Network_Factorization

TYPE SCOTCH_Mapping_Ordering
   _TYPE (SCOTCH_Mapping), :: scotch_mapping
   _TYPE (SCOTCH_Ordering) :: scotch_ordering
ENDTYPE

TYPE SCOTCH_Mapping
   CHARACTER (LEN = SCOTCH_MAX_LEN) :: architecture_string = ""  // Optional input
   CHARACTER (LEN = SCOTCH_MAX_LEN) :: mapping_string = ""  // Input
ENDTYPE

TYPE SCOTCH_Ordering
   CHARACTER (LEN = SCOTCH_MAX_LEN) :: ordering_string = ""  // Input
ENDTYPE

TYPE TAUCS_Network_Preconditioner  // Also used for incomplete-factorization preconditioning!
   REAL :: subtree_ratio = 0.0  // Input
   REAL (KIND = r_taucs) :: droptol = 0.0_taucs, subgraphs = 1.0_taucs  // Input
   CHARACTER (LEN = 10) :: ordering_method = "genmed"  // Input
   // "none", "genmed", "metis" or "identity"
ENDTYPE  // TAUCS_Network_Preconditioner

1.4.2 Type Dual_Network_Preconditioner

When using an iterative solver, the preconditioner is the most important to efficiency. Several data-types are associated with the several preconditioning options available in SSCNO:

"WEAVE.f90" 1.4.2 ≡

TYPE Dual_Network_Preconditioner
   integer :: method = diagonal_precond  // Input
   // no_precond, diagonal_precond, MST_BBD_precond, MST_LDLt_precond, or
   // ST_precond, ST_MST_precond, TAUCS_MWB_precond or TAUCS_LLt_precond
   _TYPE (Spanning_Tree), pointer :: tree = NULL  // Optional input
   _TYPE (Support_Tree_Preconditioner), pointer :: support_tree = NULL
   _TYPE (TAUCS_Network_Preconditioner), pointer :: taucs_preconditioner = NULL
   logical :: reoptimize_ST_MST = T  // Optional input
ENDTYPE  // Dual_Network_Preconditioner

TYPE Spanning_Tree
   integer (KIND = i_wp), dimension (:), _dynamic :: NULLIFIED (parents),
   _NULLIFIED (cardinalities), _NULLIFIED (level_ordering)
   real (KIND = r_wp), dimension (:), _dynamic :: NULLIFIED (path_labels)
   integer (KIND = i_byte), dimension (:), _dynamic :: NULLIFIED (orientations)
   logical (KIND = l_wp), dimension (:), _dynamic :: NULLIFIED (mask)
ENDTYPE  // Spanning_Tree
TYPE Support_Tree_Preconditioner
   _TYPE (SCOTCH_Mapping_Ordering), POINTER :: scotch_mapping = NULL
   _TYPE (CHACO_Mapping_Partitioning), POINTER :: chaco_mapping = NULL
   INTEGER (KIND = i_wp) :: height = 0, max_height = 0, degree = 2     // Optional input
   INTEGER :: mapping_method = Default_ordering     // Input
   // Default_ordering, Matching_ordering, SCOTCH_ordering or CHACO_ordering
   LOGICAL :: support_special_nodes = T     // Optional input
ENDTYPE     // Support_Tree_Preconditioner

TYPE CHACO_Mapping_Partitioning
   _TYPE (CHACO_Strategy) :: strategy
   _TYPE (CHACO_Architecture) :: architecture
ENDTYPE     // CHACO_Architecture

TYPE CHACO_Strategy
   INTEGER (KIND = i_word) :: global_method = CHACO_dummy     // Input
     // CHACO_multilevel, CHACO_linear, CHACO_spectral or CHACO_inertial
   INTEGER (KIND = i_word) :: local_method = CHACO_dummy     // Input
     // CHACO_Local_KL or CHACO_Local_none
   INTEGER (KIND = i_word) ::
      sequence = 0, rqi_flag = 0, vmax = 1, ndims = 1     // Optional input
   REAL (KIND = r_fl) :: eigtol = 1 * 10^{-3} r_word     // Optional input
ENDTYPE     // CHACO_Strategy