Creation of the Network Optimization problem

Aleksandar Donev

February 2000

Contents

1 Module Network_Geometry ........................................ 1
   1.1 Creation and Destruction of Network Arrays .................. 3
   1.2 Creating the Network Optimization Problem .................. 4
   1.3 The Status of Special Nodes and Arcs ........................ 5
   1.4 Data and Computation Reordering ............................. 7
   1.5 Supplies-Demands Assignment ................................. 9
   1.6 Array Allocation and Deallocation ........................... 11

2 Formatting rules for HPF/F90 files ............................... 13
1 \textbf{Module Network\_Geometry}

This module provides routines that finish the job begun in \textit{Lattice\_Geometry} in setting up the network optimization problem. The module \textit{Lattice\_Geometry} creates a near-neighbour lattice in the form of a graph, i.e. it sets up the connectivity (via \textit{heads\_tails}) and marks the geometrical status of each arc and node.

To make a network optimization problem out of this, we need to assign values to the supplies-demand vector $\mathbf{b}$, assign cost parameter for calculating the arc cost functions and assign initial guesses for the node potentials and arc flows if these are required by the algorithm. The module \textit{Network\_Geometry} allocates all but the cost-related network arrays, assigns the supplies-demands for the nodes based on the network geometry and the geometrical status indicators calculated in \textit{Lattice\_Geometry}, and also applies computation (node renumbering) and data reordering (arc permuting) to the lattice network to improve cache performance, using the module \textit{Space\_Filling\_Curves}.

Here are the important global variables in this module. Most of them are part of the namelist \texttt{ProblemOptions}, which can be used to change their default values via the options file:

- \texttt{source\_arcs\_cost}, \texttt{sinks\_arcs\_cost} and \texttt{plate\_arcs\_cost} can have the values \textquote{L} for low-cost arcs, \textquote{H} for high-cost, \textquote{R} for regular-cost or \textquote{D} for dummy arcs. They tell what the status will be of the arcs connected to the $S$ and $T$ node and the plate arcs for each dimension. The values for dimensions that do not have special arcs are ignored, but must be present in the namelist file.

- \texttt{sources\_status} and \texttt{sinks\_status} can be \textquote{R} for regular, \textquote{F} for fixed-potential or \textquote{D} for dummy nodes. They give the node status of the $ST$ nodes for different dimensions.

- \texttt{sources\_flow\_capacity} and \texttt{sinks\_flow\_capacity} give the supply-demand for all the $ST$ nodes.

- \texttt{injection\_flow\_capacity} and \texttt{outflow\_flow\_capacity} give the total flow injected into or taken out from the injection/outflow nodes. Note that there is no dimension here, since all injection/outflow nodes are lumped together. \textit{It is the user's responsibility to insure that the total flow taking into account all present sources and sinks and injection/outflow nodes be 0}. This module checks this within a low precision and will abort if this is not true. Other than this, no restriction on the magnitude or sign of the flows is given, which gives great flexibility in making exotic geometries.

- \texttt{injection\_flow\_distribution} and \texttt{injection\_flow\_parameters} are related to the distribution of the flows injected into the injection nodes. The distribution can be \textquote{U} for uniform, \textquote{G} or \textquote{N} for gaussian (uniform) or \textquote{F} for a fixed (non-random) supply-demand at each injection node. For a uniform distribution, \texttt{injection\_flow\_parameters} gives the range of the uniformly distributed values for the injected supplies-demands. For a gaussian distribution, the first entry in \texttt{injection\_flow\_parameters} is the mean of the distribution and the second value is the standard deviation. For a fixed value of the flow this parameter is meaningless.

This program will insure that the total sum of all the injected flows is equal to \texttt{injection\_flow\_capacity} by rescaling the mean and standard deviation of the desired distribution to make the mean flow equal to \texttt{injection\_flow\_capacity/n\_injection\_nodes} and tempering with one of the values to compensate for any misbalance that occurred due to the random character of the values.

- \texttt{outflow\_flow\_distribution} and \texttt{outflow\_flow\_parameters} are as \texttt{injection\_flow\_distribution} and \texttt{injection\_flow\_parameters} only for outflow sites.
\textit{SFC} determines whether a Hilbert ('H') or Morton ('M') space-filling curve is used to renumber the nodes in the graph for near-optimal cache performance of the algorithm. Any other value will cause a random numbering of the nodes to be chosen (this can be used to test the benefits of data reordering on the overall execution time). For the computation reordering, there are no choices—the arcs are always ordered in order of their heads first, then tails (a lexicographical sort), which a good choice in this case.

Notice that special nodes and special arcs are ordered in order of their special status, and the number of elements of each special kinds are counted and recorded.
"WEAVE.F90" 1.0.1

MODULE Network_Geometry
USE Precision
USE Error_Handling
USE System_Monitors
USE Random_Numbers
USE Sorting_Ranking
USE Space_Filling_Curves
USE Network_Data_Structures
USE Lattice_Geometry
IMPLICIT NONE
PUBLIC :: InitializeNetwork, DestroyNetwork, CreateNetwork
PRIVATE
 CHARACTER, DIMENSION (n_dim), PUBLIC, SAVE :: sources arcs_cost = 'L', sink arcs_cost = 'L',
 sources status = 'R', sinks status = 'R'
 CHARACTER, PUBLIC, SAVE :: plate arcs cost = 'L'
 / All plate arcs must be identical in cost to avoid complications
REAL (KIND = r_wp), PUBLIC, DIMENSION (n_dim), SAVE :: sources flow capacity = 0.0, r_wp,
 sinks flow capacity = 0.0, r_wp    / The supply/demand of the ST nodes—the user is to insure
 that the total supply-demand is 0.0
REAL (KIND = r_wp), PUBLIC, SAVE :: injection flow capacity = 1.0, r_wp,
 outflow flow capacity = -1.0, r_wp    / The total supply/demand of the
 injection and outflow nodes
 CHARACTER, PUBLIC, SAVE :: injection flow distribution = 'F', outflow flow distribution = 'F'
 / "Fixed", "Uniform", "Normal" (also "Gaussian") or "Special" distribution for the
 supplies-demands
REAL (KIND = r_wp), PUBLIC, DIMENSION (2), SAVE :: injection flow parameters = (/ 1.0, r_wp,
 0.0, r_wp /), outflow flow parameters = (/ 1.0, r_wp, 0.0, r_wp /)
 / The parameters for the distribution of the supplies/demands for injection and outflow nodes
 CHARACTER, PUBLIC, SAVE :: SFC = 'H'
 / The type of space-filling curve used in data reordering
/* Just some temporaries to avoid working with characters: */
INTEGER (KIND = i Byte), DIMENSION (n_dim), SAVE :: sources arcs cost, sink arcs cost,
 sources status, sinks status    / Converted to integers from characters
INTEGER (KIND = i Byte), SAVE :: plate arcs cost
 / All plate arcs must be identical in cost to avoid complications
NAMELIST/Problem Options/ sources status, sinks status, source arcs cost, sink arcs cost,
 plate arcs cost, sources flow capacity, sinks flow capacity, injection flow capacity,
 outflow flow capacity, injection flow distribution, outflow flow distribution,
 injection flow parameters, outflow flow parameters, SFC
CONTAINS
( InitializeNetwork 1.1.1)
( DestroyNetwork 1.6.2)
( CreateNetwork 1.2.1)

END MODULE Network_Geometry
1.1 Creation and Destruction of Network Arrays

As in other modules, the initialization routine InitializeNetwork reads the options file for the namelist ProblemOptions and converts some of the character values into byte-integers:

\[
\text{\langle InitializeNetwork 1.1.1 \rangle} \equiv \\
\text{SUBROUTINE InitializeNetwork()} \\
\quad \text{INTEGER (KIND = i,itype) :: node, arc} \\
\quad \text{INTEGER :: dim} \\
\quad \text{DO dim = 1, n_dim} \quad // \text{By default each source and sink has in/out flow of 1.0 units} \\
\quad \quad \text{IF (boundaries bc_source, sink, dim) THEN} \\
\quad \quad \quad \text{sources flow capacity dim = 1.0} \quad // \text{The default inflow} \\
\quad \quad \quad \text{sinks flow capacity dim = -1.0} \quad // \text{Default outflow} \\
\quad \quad \text{END IF} \\
\quad \text{END DO} \\
\quad \text{REWIND (UNIT = program_options_unit)} \quad // \text{Rewind the options file} \\
\quad \text{READ (UNIT = program_options_unit, NML = ProblemOptions, ISTAT = error_status)} \\
\quad \text{IF (error status \neq 0) CALL CriticalError (message = "NAMELIST ProblemOptions was not read successfully" || " from file " || TRIM(options_file), caller = "InitializeNetwork")} \\
\quad \text{WRITE (UNIT = message_log_unit, NML = ProblemOptions)} \\
\quad \text{/* Convert the character status identifiers into integers for ease of use: */} \\
\quad \text{DO dim = 1, n_dim} \\
\quad \quad \text{CONVERTARCCHARACTERSTATUS(source arcs cost dim, source arcs cost dim)} \\
\quad \quad \text{CONVERTARCCHARACTERSTATUS(sink arcs cost dim, sink arcs cost dim)} \\
\quad \quad \text{CONVERTNODECHARACTERSTATUS(sources status dim, sources status dim)} \\
\quad \quad \text{CONVERTNODECHARACTERSTATUS(sinks status dim, sinks status dim)} \\
\quad \text{END DO} \\
\quad \text{CONVERTARCCHARACTERSTATUS(plate arcs cost, plate arcs cost)} \\
\text{END SUBROUTINE InitializeNetwork}
\]

This code is used in section 1.0.1.
These macros will convert user-input as characters into integer status identifiers:

```
"WEAVE.F90" 1.1.2 =
@m _CONVERT_ARC_CHARACTER_STATUS(character_status, integer_status)
    SELECT CASE (character_status)
        CASE ('D', 'd') // Dummy arc
            integer_status = dummy_arc
        CASE ('L', 'l') // Low cost arc
            integer_status = low_cost_arc
        CASE ('H', 'h') // High-cost arc
            integer_status = high_cost_arc
        CASE DEFAULT
            integer_status = regular_cost_arc
    ENDSSELECT

@m _CONVERT_NODE_CHARACTER_STATUS(character_status, integer_status)
    SELECT CASE (character_status)
        CASE ('D', 'd') // Dummy node
            integer_status = dummy_node
        CASE ('F', 'f') // Fixed-potential node
            integer_status = fixed_potential_node
        CASE DEFAULT
            integer_status = regular_node
    ENDSSELECT
```

### 1.2 Creating the Network Optimization Problem

The procedure `CreateNetwork` will assign values to the supplies-demands and indicate for all special arcs what the type of cost is—low, high or regular. It does not generate the actual cost parameters, since this depends on the actual cost function and is done in `Cost_Parameters`. It also applies computation and data reordering to the network graph. The optional argument `reordering_timer` can be used to pass a free clock to be used in timing the piece of code that does the reordering, since this is one of the most expensive pieces of the network creation and it is worth to time it for our records. This clock is not reset here—this is the user’s choice and responsibility.
\{CreateNetwork 1.2.1\} ≡

\textbf{SUBROUTINE CreateNetwork (reordering\_timer)}
\begin{verbatim}
INTEGER, INTENT (IN), OPTIONAL :: reordering\_timer
  // An ID of an available timer to time the computation and data reordering section

INTEGER (KIND = i\_wp), DIMENSION (:) :: nodes\_reordering, nodes\_renumbering,
  arcs\_reordering  // Permutations for reordering the arcs and the nodes

INTEGER (KIND = i\_wp) :: node, arc, special\_node, special\_arc, source, sink

INTEGER :: dyn

REAL (KIND = r\_wp) :: injection\_flow\_scaling, injection\_flow\_mean, injection\_flow\_std,
  injection\_flow\_lb, injection\_flow\_ub, injection\_flow, outflow\_flow\_scaling,
  outflow\_flow\_mean, outflow\_flow\_std, outflow\_flow\_lb, outflow\_flow\_ub,
  outflow\_flow, total\_injection\_flow, total\_outflow\_flow,
  total\_flow, mean\_flow

WRITE (message\_log\_unit, *) "Creating a connected lattice network with:"
WRITE (message\_log\_unit, *) "n\_arcs=" , n\_arcs, " n\_nodes=" , n\_nodes, " n\_special\_nodes=" ,
  n\_special\_nodes, " n\_special\_arcs=" , n\_special\_arcs

\{ AssignSpecialStatuses 1.3.1\}
\begin{verbatim}
IF (PRESENT(reordering\_timer)) \textbf{CALL} StartTimer (reordering\_timer)
\{ ReorderDataAndComputation 1.4.1\}
\end{verbatim}
\begin{verbatim}
IF (PRESENT(reordering\_timer)) \textbf{CALL} StopTimer (reordering\_timer)
\{ AllocateNetworkArrays 1.6.1\}
\{ AssignSuppliesDemands 1.5.1\}
\end{verbatim}

\textbf{END SUBROUTINE CreateNetwork}
\end{verbatim}

This code is used in section 1.0.1.

### 1.3 The Status of Special Nodes and Arcs

This section of the code assigns a status (in special\_nodes\_status and special\_arcs\_status) to the special arcs
and nodes based on their geometric status as determined in Lattice\_Geometry.
\[\text{AssignSpecialStatuses} \ 1.3.1\] \equiv

\begin{verbatim}
ALLOCATE (special_nodes_status(-n_special_nodes : 0), stat = error_status)
CALL RecordAllocation(n_elements = (n_special_nodes + 1), mold = 1_{\text{byte}},
caller = "CreateNetwork", alloc_status = error_status)
ALLOCATE (special_arcs_status(-n_special_arcs : 0), stat = error_status)
CALL RecordAllocation(n_elements = (n_special_arcs + 1), mold = 1_{\text{byte}}, caller = "CreateNetwork",
alloc_status = error_status)
/* I will use the node status array special_nodes_status as a temporary so that I register which source
or sink node a given source or sink arc is connected to */
DO dim = 1, n_{\text{dim}}
   IF (boundaries \_bc_source_sinks, dim) THEN
      source = sources_{\text{dim}}
special_nodes_status_{\text{source}} = source_arcs_cost_{\text{dim}}
sink = sinks_{\text{dim}}
special_nodes_status_{\text{sink}} = sink_arcs_cost_{\text{dim}}
   END IF
END DO
n_{\text{regular_cost_arcs}} = 0; n_{\text{high_cost_arcs}} = 0; n_{\text{low_cost_arcs}} = 0
special_arcs_status_0 = dummy_{\text{arc}}
DO special_{\text{arc}} = -n_{\text{special_arcs}}, -1
   SELECT CASE (arc_status_{\text{special_arcs}})
      CASE (plate_{\text{arc}})
special_arcs_status_{\text{special_arcs}} = plate_{\text{arcs_cost}}
   CASE (source_{\text{arc}})
      source = heads_{\text{tails}}_1, special_{\text{arc}}  // The head is the source node
      special_arcs_status_{\text{special_arcs}} = special_nodes_status_{\text{source}}  // Stored result
   CASE (sink_{\text{arc}})
      sink = heads_{\text{tails}}_2, special_{\text{arc}}  // The tail is the sink node
      special_arcs_status_{\text{special_arcs}} = special_nodes_status_{\text{sink}}  // Stored result
   CASE DEFAULT  // Should not really happen at all
      special_arcs_status_{\text{special_arcs}} = regular_{\text{cost_arcs}}  // Default is regular
   END SELECT
   SELECT CASE (special_arcs_status_{\text{special_arcs}})  // Count the three flavors of arcs
      CASE (low_{\text{cost_arcs}})
         n_{\text{low_cost_arcs}} = n_{\text{low_cost_arcs}} + 1
      CASE (high_{\text{cost_arcs}})
         n_{\text{high_cost_arcs}} = n_{\text{high_cost_arcs}} + 1
      CASE (regular_{\text{cost_arcs}})
         n_{\text{regular_cost_arcs}} = n_{\text{regular_cost_arcs}} + 1
   END SELECT
END DO
special_nodes_status_0 = dummy_{\text{arc}}
special_nodes_status = n_{\text{special_nodes}}(-1) = regular_{\text{node}}  // The default
DO dim = 1, n_{\text{dim}}
   IF (boundaries \_bc_source_sinks, dim) THEN
      source = sources_{\text{dim}}
special_nodes_status_{\text{source}} = sources_status_{\text{dim}}
sink = sinks_{\text{dim}}
   END IF
END DO
\end{verbatim}
$\texttt{special\_nodes\_status}_\text{sink} = \texttt{sinks\_status\_dim}$

END IF

END DO

n\_fixed\_potential\_nodes = 0; n\_regular\_nodes = 0

DO special\_node = n\_special\_nodes, -1

SELECT CASE (special\_nodes\_status\_special\_node)

CASE (fixed\_potential\_node)  // Fixed-potential special node

n\_fixed\_potential\_nodes = n\_fixed\_potential\_nodes + 1

CASE (regular\_node)  // A regular kind of special node

n\_regular\_nodes = n\_regular\_nodes + 1

END SELECT

END DO

This code is used in section 1.2.1.

### 1.4 Data and Computation Reordering

This is the code that renumbers the nodes based on their positions along a space-filling curve, then updates the heads-and-tails array to correct for the new numbering, and finally permutes the arc arrays in lexicographical head-tail order.
\[\text{ReorderDataAndComputation}\ 1.4.1 \equiv\]
\begin{verbatim}
ALLOCATE (nodes_reordering(-n_special_nodes : n_nodes), STAT = error_status) // For nodes
CALL RecordAllocation(n_elements = (n_special_nodes + n_nodes + 1), mold = iwp, 
caller = "CreateNetwork", alloc_status = error_status)

nodes_reordering := 0

SELECT CASE (SFC)
CASE ('h', 'H', 'm', 'M') // The user requested data reordering
CALL SFCOrder (points_coords := nodes_coords; 1:n_nodes, SFC_order := nodes_reordering; 1:n_nodes, 
SFC := SFC, coords_min = (/ (1.0, dim = 1, n_dim) /), coords_max = REAL (lengths) )
// Number the nodes along an SFC
CASE DEFAULT
CALL DisorderPermutation(disorder = 1.0, permutation = nodes_reordering; 1:n_nodes)
// Random numbering for the nodes
END SELECT

IF (n_special_nodes > 0) THEN
CALL QuickRank(array := special_nodes_status; -n_special_nodes(-1), 
permutation := nodes_reordering; -n_special_nodes(-1), pivot_selection := 'R')
nodes_reordering := n_special_nodes(-1) = nodes_reordering; n_special_nodes(-1) - (n_special_nodes + 1)
special_nodes_status := n_special_nodes(-1) = special_nodes_status(nodes_reordering; -n_special_nodes(-1))
END IF

nodes_coords := nodes_coords(:, nodes_reordering)
nodes_status := nodes_status(nodes_reordering)

ALLOCATE (nodes_renumbering(-n_special_nodes : n_nodes), STAT = error_status) // For nodes
CALL RecordAllocation(n_elements = (n_special_nodes + n_nodes + 1), mold = iwp, 
caller = "CreateNetwork", alloc_status = error_status)

DO node := -n_special_nodes, n_nodes
nodes_renumbering(nodes_reordering_node) = node // Inverse of the permutation
END DO

CALL RecordAllocation(n_elements = -SIZE(nodes_renumbering, iwp), mold = iwp)
DEALLOCATE (nodes_renumbering)

DO dim = 1, n_dim // Sources and sinks have been renumbered
IF (boundaries bc_source, sink, dim) THEN
source := sources dim
sources_dim := nodes_renumbering(source
sink := sinks dim
sinks_dim := nodes_renumbering(sink
END IF
END DO

DO arc := -n_special_arcs, n_arcs // Nodes have been renumbered
heads_tails;, arc := nodes_renumbering(heads_tails;, arc
END DO

CALL RecordAllocation(n_elements = -SIZE(nodes_renumbering, iwp), mold = iwp)
DEALLOCATE (nodes_renumbering)

/* Now we reorder and permute the nodes and arcs: */
ALLOCATE (arcs_reordering(-n_special_arcs : n_arcs), STAT = error_status)
CALL RecordAllocation(n_elements = (n_special_arcs + n_arcs + 1), mold = iwp, 
caller = "CreateNetwork", alloc_status = error_status)
\end{verbatim}
§1.4.1–§1.5.1[#11–#13]  Creation of the Network Optimization problem

Supplies-Demands Assignment

\[
\text{arcs\_reordering}_0 = 0
\]

\text{CALL QuickRank}(\text{array} = \text{heads\_fails}_1, 1:\text{\_n\_arcs}, \text{permutation} = \text{arcs\_reordering}_1:\text{\_n\_arcs}, \text{pivot\_selection} = 'U')

\text{CALL ShellInsertionRank}(\text{array} = \text{heads\_fails}_1, 1:\text{\_n\_arcs}, \text{permutation} = \text{arcs\_reordering}_1:\text{\_n\_arcs}, \text{partially\_ranked} = T, \text{method} = "Insertion")

\text{IF (n\_special\_arcs > 0) THEN}

\text{CALL QuickRank}(\text{array} = \text{special\_arcs\_status} - \text{n\_special\_arcs}\_(-1), \text{permutation} = \text{arcs\_reordering} - \text{n\_special\_arcs\_(-1)}, \text{pivot\_selection} = 'R')

\text{arcs\_reordering} - \text{n\_special\_arcs\_(-1)} = \text{arcs\_reordering} - \text{n\_special\_arcs\_(-1)} - (\text{n\_special\_arcs} + 1)

\text{special\_arcs\_status} - \text{n\_special\_arcs\_(-1)} = \text{special\_arcs\_status}(\text{arcs\_reordering} - \text{n\_special\_arcs\_(-1)})

\text{END IF}

\text{heads\_fails} = \text{heads\_fails}_1, \text{arcs\_reordering}

\text{arcs\_status} = \text{arcs\_reordering}

\text{CALL RecordAllocation(\text{n\_elements} = -\text{SIZE}(\text{arcs\_reordering}, i_{\text{wp}}), \text{mold} = l_{\text{wp}})}

\text{DEALLOCATE(\text{arcs\_reordering})}

\text{This code is used in section 1.2.1.}

1.5  Supplies-Demands Assignment

The supplies-demands for the nodes are assigned based on their geometrical status. Note that only \( ST \) and injection/outflow nodes are given non-zero supply-demand. This is not a requirement though. Also notice that the value of one of the supplies-demands is tinkered with so that the total has the correct value.

\[ \langle \text{AssignSuppliesDemands 1.5.1} \rangle \equiv \]

\[ \text{supplies\_demands} = 0_{\text{\_wp}} \]

\text{DO dim = 1, n\_dim}  // Assign supplies-demands for sources and sinks

\text{IF (boundaries bc\_source\_sink, dim) THEN}

\text{source} = \text{sources dim}

\text{supplies\_demands source} = \text{sources flow\_capacity dim}

\text{sink} = \text{sinks dim}

\text{supplies\_demands sink} = \text{sinks flow\_capacity dim}

\text{END IF}

\text{END DO}

\text{\_IN\_OUT\_Supplies\_Demands(injection)}

\text{\_IN\_OUT\_Supplies\_Demands(outflow)}

\text{total\_flow} = \text{SUM(supplies\_demands)}

\text{WRITE(message log\_unit, *) "The total supply/demand is: ", total\_flow}

\text{IF (ABS(total\_flow) > 100.0_{\text{\_wp}} * \text{Epsilon}(1.0_{\text{\_wp}})) THEN}  // Excessive imbalance in b

\text{CALL CriticalError(message = "The sum of supplies and demands was not zero within loose precision:", caller = "CreateNetwork")}

\text{END IF}

\text{This code is used in section 1.2.1.}
This macro deals with assigning supplies-demands for both the injection and the outflow sites:

"WEAVE.f90" 1.5.2
@m INOUT_SUPPLIES_DEMANDS (inout)
pliesDemands, inout: IF (n_inout & nodes > 0) THEN
    // There are injection/outflow sites that need to be assigned supplies/demands
    inout & flow = inout & flow capacity / REAL(n_inout & nodes, KIND = r_wp)
SELECT CASE (inout & flow_distribution)
  CASE ('U', 'u') // Uniform distribution
    mean_flow = 0.5_r_wp * (inout & flow_parameters_1 + inout & flow_parameters_2)
  IF (ABS(mean_flow) > 100.0_r_wp * EPSILON(1.0_r_wp)) THEN
    inout & flow scaling = inout & flow / mean_flow
  ELSE
    inout & flow scaling = 1.0_r_wp
  END IF
  inout & flow lb = inout & flow scaling * inout & flow_parameters_1
    // Lower bound for uniformly-distributed flow
  inout & flow ub = inout & flow scaling * inout & flow_parameters_2 // Upper bound
  CASE ('N', 'n', 'g', 'G')
    mean_flow = inout & flow_parameters_1
  IF (ABS(mean_flow) > 100.0_r_wp * EPSILON(1.0_r_wp)) THEN
    inout & flow scaling = inout & flow / mean_flow
  ELSE
    inout & flow scaling = 1.0_r_wp
  END IF
  inout & flow mean = inout & flow scaling * inout & flow_parameters_1 // Mean
  inout & flow std = inout & flow scaling * inout & flow_parameters_2 // STD
ENDSELECT

total & inout & flow = 0.0_r_wp // I keep a running sum to insure zero total flow
DO node = 1, n_nodes
  IF (nodes_status node = inout & node) THEN
    SELECT CASE (inout & flow_distribution)
      CASE ('U', 'u') // Uniform distribution
        CALL RandomUniform(supplies_demands_node, range = (inout & flow lb, inout & flow ub))
      CASE ('N', 'n', 'g', 'G') // Normal distribution
        CALL RandomNormal(supplies_demands_node, mean std = (inout & flow mean, inout & flow std))
      CASE DEFAULT
        supplies_demands_node = inout & flow
    ENDSELECT
    total & inout & flow += supplies_demands_node
  END IF
END DO
CorrectFlow & inout: DO node = 1, n_nodes // Correct for any flow misbalance
  IF (nodes_status node = inout & node) THEN
    supplies_demands_node = supplies demands_node + inout & flow capacity - total & inout & flow
  EXIT CorrectFlow & inout
END IF
END DO CorrectFlow & inout
END IF SuppliesDemands & inout
1.6 Array Allocation and Deallocation

The routine DestroyNetwork deallocates the network arrays, which are allocated in CreateNetwork. I have delayed the allocation of these arrays as long as possible to avoid unnecessary reordering.

\[
\langle \text{AllocateNetworkArrays 1.6.1} \rangle \equiv
\]

\[
\begin{align*}
\text{ALLOCATE} & \quad (\text{nodes\_mask} \quad (-n\_special\_nodes : n\_nodes), \quad \text{STAT} = \text{error\_status}) \\
\text{CALL} & \quad \text{RecordAllocation} (n\_elements = n\_special\_nodes + n\_nodes + 1, \quad \text{mold} = T_{i,n}, \quad \text{caller} = \text{"CreateLatticeNetwork"}, \quad \text{alloc\_status} = \text{error\_status}) \\
& \quad \text{nodes\_mask} = T_{i,0}; \quad \text{nodes\_mask}_0 = F_{i,0} \\
\text{ALLOCATE} & \quad (\text{arcs\_mask} \quad (-n\_special\_arcs : n\_arcs), \quad \text{STAT} = \text{error\_status}) \\
\text{CALL} & \quad \text{RecordAllocation} (n\_elements = n\_special\_arcs + n\_arcs + 1, \quad \text{mold} = T_{i,n}, \quad \text{caller} = \text{"CreateLatticeNetwork"}, \quad \text{alloc\_status} = \text{error\_status}) \\
& \quad \text{arcs\_mask} = T_{i,n}; \quad \text{arcs\_mask}_0 = F_{i,n} \\
\text{ALLOCATE} & \quad (\text{nodes\_potentials} \quad (-n\_special\_nodes : n\_nodes), \quad \text{STAT} = \text{error\_status}) \\
\text{CALL} & \quad \text{RecordAllocation} (n\_elements = n\_special\_nodes + n\_nodes + 1, \quad \text{mold} = 1.0_{i,n}, \quad \text{caller} = \text{"CreateNetwork"}, \quad \text{alloc\_status} = \text{error\_status}) \\
\text{ALLOCATE} & \quad (\text{supplies\_demands} \quad (-n\_special\_nodes : n\_nodes), \quad \text{STAT} = \text{error\_status}) \\
\text{CALL} & \quad \text{RecordAllocation} (n\_elements = n\_special\_nodes + n\_nodes + 1, \quad \text{mold} = 1.0_{i,n}, \quad \text{caller} = \text{"CreateNetwork"}, \quad \text{alloc\_status} = \text{error\_status}) \\
\text{ALLOCATE} & \quad (\text{arcs\_flows} \quad (-n\_special\_arcs : n\_arcs), \quad \text{STAT} = \text{error\_status}) \\
\text{CALL} & \quad \text{RecordAllocation} (n\_elements = n\_special\_arcs + n\_arcs + 1, \quad \text{mold} = 1.0_{i,n}, \quad \text{caller} = \text{"CreateNetwork"}, \quad \text{alloc\_status} = \text{error\_status})
\end{align*}
\]

This code is used in section 1.2.1.
\texttt{\langle \text{DestroyNetwork} \ 1.6.2 \rangle \equiv}

\textbf{SUBROUTINE DestroyNetwork()}

\textbf{IMPLICIT NONE}

\textbf{CALL RecordAllocation(n_elements = -size(nodes\_mask, i\_wp), mold = } T\textsubscript{1\_wp})
\textbf{DEALLOCATE(nodes\_mask)}
\textbf{CALL RecordAllocation(n_elements = -size(arcs\_mask, i\_wp), mold = } T\textsubscript{1\_wp})
\textbf{DEALLOCATE(arcs\_mask)}

\textbf{CALL RecordAllocation(n_elements = -size(special\_nodes\_status, i\_wp), mold = } l\textsubscript{i\_byte})
\textbf{DEALLOCATE(special\_nodes\_status)}
\textbf{CALL RecordAllocation(n_elements = -size(special\_arcs\_status, i\_wp), mold = } l\textsubscript{i\_byte})
\textbf{DEALLOCATE(special\_arcs\_status)}

\textbf{CALL RecordAllocation(n_elements = -size(nodes\_potentials, i\_wp), mold = } 1.0\textsubscript{i\_wp})
\textbf{DEALLOCATE(nodes\_potentials)}
\textbf{CALL RecordAllocation(n_elements = -size(supplies\_demands, i\_wp), mold = } 1.0\textsubscript{i\_wp})
\textbf{DEALLOCATE(supplies\_demands)}
\textbf{CALL RecordAllocation(n_elements = -size(arcs\_flows, i\_wp), mold = } 1.0\textsubscript{i\_wp})
\textbf{DEALLOCATE(arcs\_flows)}

\textbf{END SUBROUTINE DestroyNetwork}

This code is used in section 1.0.1.
2 Formatting rules for HPF/F90 files

These are just same auxilliary formatting rules and useful macros I use from time to time.

@m _SIZE(array, _kind, ...) ifelse (#0, 0, int(size(array), kind=_kind), int(size(array, #), kind=_kind))
@m _MAXLOC(array, _kind, ...) ifelse (#0, 0, int(maxloc(array), kind=_kind), int(maxloc(array, #), kind=_kind))
@m _MINLOC(array, _kind, ...) ifelse (#0, 0, int(minloc(array), kind=_kind), int(minloc(array, #), kind=_kind))
@m _LBIND(array, _kind, ...) ifelse (#0, 0, int(lbound(array, dim=1), kind=_kind), int(lbound(array, #), kind=_kind))
@m _UBIND(array, _kind, ...) ifelse (#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_INTEGER(...) integer :: #.
@m _DECLARE_REAL(...) integer (kind=ip) :: #.
@m _DECLARE_REAL(...) real (kind=rp) :: #.
@m _DECLARE_REAL(...) real (kind=sp) :: #.
@m _DECLARE_REAL(...) real (kind=dp) :: #.
@m _FULLEXTENT(_rank) : do (dim, 2, _rank) { , , }
@m _VARSEQUENCE(_variable, _start, _end)
    _variable##_start##_do (dim, _eval(_start + 1), _end) { , _variable##&dim }
@m _NESTEDLOOPSTART(_variable, _array, _rank, _kind) _do (dim, _rank, 1, -1) { do
    _variable##&DIM = _LBIND(_array, _kind, DIM = DIM), _UBIND(_array, _kind, DIM = DIM)}
@m _NESTEDLOOPEND(_rank) _do (DIM, 1, _rank) { end do }
@m _DUMMY(...) _do
@m _DISPLAYARRAY(message, array)
    if (size(array) le 20) then
      write(message, print_unit, "(A)" message
      write(message, print_unit, "(2065.2)") array
    end if