Near-Neighbour Lattice Networks

Aleksandar Donev

February 2000

Contents

1 Module Lattice_Geometry

- Setting up the Geometry ........................................ 1
- Creating the Lattice ............................................. 2
- Procedure CreateLatticeNetwork ............................... 2
- Initial Allocation or Network Arrays .......................... 3
- Lattice Geometry Calculation .................................. 4
- Connected-Component Extraction ............................... 4
- Reallocating and Packing the Network Arrays ............... 5
- Destroying the Lattice .......................................... 5

2 Formatting rules for HPF/F90 files ............................ 6


1 Module \textit{Lattice\_Geometry}

This module will set up a lattice with a simple (one-site) basis and incorporate appropriate boundary conditions. Although it is meant to be primarily used for hypercube lattices, with minor to no modifications the source code is applicable to any \(n\)-dimensional simple lattice, such as a triangular lattice. The lattice is returned as a general network, i.e. a heads and tails array \texttt{heads\_tails} (from \textit{Network\_Data\_Structures}), with appropriate coordinate information for the nodes in the array \texttt{nodes\_coords}, as well as status identifiers telling what kind of arc/node each one is in the arrays \texttt{arc\_status} and \texttt{nodes\_status}.

The routine \texttt{InitializeLattice} reads the parameter file and should be called once in the beginning, while the very bulky and complex routine \texttt{CreateLatticeNetwork} can be called many times to create lattices with different parameters (such as dilution or sizes of the lattice), so long as it is always called subsequently after a call to \texttt{DestroyLatticeNetwork}, which deallocates the arrays that are created in \texttt{CreateLatticeNetwork}.

I will explain some of the important publicly available module data as I go along. For now, I want to say that after the call to \texttt{CreateLatticeNetwork} each arc and node will have a status which is related to their place in the geometry. The possible kinds are defined as one-byte integers below, such as \texttt{interior\_arc} or \texttt{plate\_arc} or \texttt{injection\_node} etc. The arrays \texttt{arc\_status} and \texttt{nodes\_status} are one-byte integer arc and nodal arrays which can contain an integer from 0-255 giving the status (kind) indicator for each node and arc. The module \textit{Lattice\_Geometry} defines the status of arcs and nodes based on their geometrical position and role. The module \textit{Network\_Geometry} will use this module and base the creation of the arrays in \textit{Network\_Data\_Structures} upon these geometrical statuses.

Basically, a dummy node is labeled as \texttt{empty\_node}. A node in the interior of the lattice is a \texttt{interior\_node}, while a node that is on a plate boundary is a \texttt{plate\_node}. A node can also be connected to \(S\) or \(T\), as in \texttt{to\_source\_node} or \texttt{to\_sink\_node}, or it can be a \texttt{source\_node} or a \texttt{sink\_node}. It may also be an node where flow is injected (\texttt{injection\_node}), or where flow is taken out (\texttt{outflow\_node}). The number of injection and outflow nodes is recorded in \texttt{n\_injection\_nodes} and \texttt{n\_outflow\_nodes} because flow must be conserved so these will be needed when assigning supplies/demands later on. \textit{Only sources and sinks are treated as special nodes} since it is likely that these will have fixed-potential or other special boundary conditions applied to them.

Similarly, an arc can be a \texttt{dummy\_arc} or an interior \texttt{interior\_arc}, it can be a wrap-around arc \texttt{periodic\_bc\_arc} (these would not be plotted usually, for example). On the other hand, they can be arcs connecting the source/sink plates or nodes, \texttt{plate\_arc}, \texttt{source\_arc} or \texttt{sink\_arc}. \textit{Plate, source and sink arcs are treated as special arcs}, because it is expected these should be zero cost arcs later on.

\textit{Higher numbers for the status indicators superseed smaller ones}, that is, if a node is both a plate and an injection node, it will be marked as \texttt{injection\_node}.
"WEAVE.I90" 1.1

MODULE Lattice_Geometry
 USE Precision
 USE Error_Handling
 USE System_Monitors
 USE Random_Numbers
 USE Graph_Algorithms
 USE Network_Data_Structures
 USE Network_Graphics

IMPLICIT NONE
PUBLIC :: InitializeLattice, CreateLatticeNetwork, DestroyLatticeNetwork

PRIVATE

INTEGER, PARAMETER, PUBLIC :: n_directions = NDIR
   // Number of arcs originating from each lattice site (lattice coordination number).
INTEGER, DIMENSION (n_dim), PUBLIC, SAVE :: lengths  // The sizes of the lattice box
CHARACTER, PUBLIC, SAVE :: component_extraction = 'C'  // Either "Cluster" or "Backbone"
INTEGER, DIMENSION (n_dim), PUBLIC, SAVE :: source_layer_thickness = 1, sink_layer_thickness = 1,
   injection_layer_thickness = 1, outflow_layer_thickness = 1
INTEGER, DIMENSION (n_dim, n_directions), PARAMETER :: displacements = RESHAPE(SOURCE =
   (/ NO_DISPLACEMENTS /), SHAPE = (/ n_dim, n_directions /))
INTEGER, PARAMETER, PUBLIC :: bc_periodic = 1, bc_injection = 2, bc_plate = 3, bc_source_sink = 4
LOGICAL, DIMENSION (4, n_dim), PUBLIC, SAVE :: boundaries = F
   // Boundary conditions, for each dimension specify order whether the boundary is: periodic,
   injection, plate or source-sink (ST) boundary.
INTEGER (KIND = i_up), DIMENSION (n_dim), PUBLIC :: sources, sinks
INTEGER (KIND = i_up), PUBLIC, SAVE :: n_injection_nodes, n_outflow_nodes  // Counters
REAL, PUBLIC, SAVE :: node_dilution = -1.0, arc_dilution = -1.0  // Dilutions for nodes and arcs
INTEGER (KIND = i_byte), DIMENSION (:), ALLOCATABLE, PUBLIC :: arcs_status, nodes_status
   // Geometric status of each arc and node. I keep these in bytes to save memory!
INTEGER (KIND = i_byte), PARAMETER, PUBLIC :: empty_node = 0, interior_node = 1,
   plate_node = 2, to_source_node = 3, to_sink_node = 4, injection_node = 5,
   outflow_node = 6, source_node = 7, sink_node = 8  // Possible kinds of nodes for nodes_status
INTEGER (KIND = i_byte), PARAMETER, PUBLIC :: empty_arc = 0, interior_arc = 1,
   periodic_be_arc = 2, plate_arc = 3, source_arc = 4, sink_arc = F
   // Possible kinds of arcs for arcs_status

NAMELIST/LatticeOptions/ lengths, component_extraction, source_layer_thickness,
   sink_layer_thickness, injection_layer_thickness, outflow_layer_thickness, node_dilution,
   arc_dilution, boundaries  // Geometry parameters in options file.

CONTAINS
   ( InitializeLattice 1.3 )
   ( CreateLatticeNetwork 1.6 )
   ( DestroyLatticeNetwork 1.24 )

END MODULE Lattice_Geometry
1.1 Setting up the Geometry

The main job of the routine InitializeLattice is to read the namelist LatticeOptions from the options file. The main parameters that can and should be set via this file are:

*lengths* which is an array of size *n_dim* containing the lengths of the lattice box in number of sites

*component_extraction* tells how to extract the actual network from the randomly diluted lattice (which may be disconnected, for example). Either the largest connected cluster (connected component) can be extracted, "Cluster", or the largest backbone (multi-cycle) if "Backbone". This backbone cluster does not have dangling ends and is made up entirely of cycles.

*node_dilution and arc_dilution* which are default reals and give the desired dilutions for the nodes and/or arcs in the graph. For example, a *node_dilution* = 0.75 means that 25 (100 – 75) percent of the nodes will be removed in the initial phase (during the creation of the lattice). More nodes will likely be diluted when a single connected-component is extracted from the graph. *Negative dilutions imply no dilution at all.*

*boundaries* which is a logical array of size (4, *n_dim*) where there is one column for each dimension and each column contains four indicators that say, in order, whether the boundary along the dimension is: periodic, injection, plate or source-sink boundary (these are given by *bc_periodic* = 1, *bc_injection* = 2, *bc_plate* = 3, *bc_source_sink* = 4 above).

So a first column of 0, 0, 0 in boundaries says that the boundary is free (no periodic BC’s), that the nodes on the boundaries *x = 1* will be injection sites and connected in a plate with special arcs, while the nodes with *x = lengthsi* will be outflow sites and on a plate, and that there will be no source-sink pair in the *x*-direction.

*sources and sinks* are two pointer-like arrays that store the locations of the sources and sinks in each dimension. These will be needed when setting up the supplies and demands for the special nodes, since one source/sink can only belong to one dimension. Notice that an injection/outflow node can belong to more than one dimension (such as corner or edge nodes), and therefore I do not attempt to distinguish these by dimension, but rather count them all together in *n_injection_nodes* and *n_outflow_nodes*.

*source_layer_thickness, sink_layer_thickness, injection_layer_thickness and outflow_layer_thickness* give the thickness (in terms of lattice coordinates) of the layers to which the source and sink are connected and the thicknesses of the layers of injection/outflow nodes, for each dimension (just use 0 or 1 for dimensions which do not have an ST or injection/outflow boundary). Note that these can be zero, which implies that there is no sink/source or injection/outflow sites on that side of the lattice box. This allows a great variety of geometries to be simulated with this program. Notice that plate boundaries always have thickness 1 since I see no point in having a thicker plate.

Two important geometry parameters have been left out as *parameters* to be inputted via FWEB’s macros, and can not be changed at run time. These are the number of dimensions *n_dim* and the type of lattice (square nearest-neighbor, second nearest-neighbor, eight-neighbor, triangular, etc.), which is determined by the *n_directions* *n_dim*-dimensional vectors stored as rows in the array *displacements*. These displacements give the displacements (in lattice coordinates) of the neighbors of each node. For example, in a nearest neighbour hypercubic lattice, these are just the unit vectors (1,0,...), (0,1,...), etc., but for other lattices these may be more complicated. The choice to make these fixed parameters was in order to facilitate compiler optimization of the short loops they create and avoid a lot of small allocatable arrays.
\[ \text{(InitializeLattice 1.3) \equiv} \]

```fortran
SUBROUTINE InitializeLattice()
    IMPLICIT NONE
    REWIND (UNIT = program_options_unit) / / Rewind the options file
    READ (UNIT = program_options_unit, NML = LatticeOptions, IOSTAT = error_status)
    IF (error_status \neq 0) CALL CriticalError (message = "NAMELIST LatticeOptions was not read
       successfully" || " from file " || TRIM(options_file), caller = "InitializeLattice")
    WRITE (UNIT = message_log_unit, NML = LatticeOptions)
END SUBROUTINE InitializeLattice
```

This code is used in section 1.1.

### 1.2 Creating the Lattice

To create a lattice as a network, we need to assign the heads-and-tails arrays for the network, along with
the correct number of nodes \( \text{n\_nodes} \) and arcs \( \text{n\_arcs} \), and in our case assign coordinates to the nodes in
\( \text{n\_coordinates} \). One has to choose a default numbering for the nodes, and in my case this is done by
numbering the points (sites) of the lattice in the \( x \) direction first, then in \( y \), etc. The following two macros
do the conversion from coordinates to site number and vice-versa:

\[ \"\text{WEAVE.f90}\" \text{ 1.5 \equiv} \]

```fortran
@m _COORDSToPOINT(point\_coords, point\_num, dim)
    point\_num = 1
    DO dim = 1, n\_dim
        point\_num = point\_num + (point\_coords\_dim - 1) * lengths\_product\_dim
    END DO

@m _POINTToCOORDS(point\_num, point\_coords, dim, point)
    point = point\_num - 1
    DO dim = n\_dim, 1, -1
        point\_coords\_dim = point / lengths\_product\_dim + 1 \ / / Integer coefficient
        point = point - (point\_coords\_dim - 1) * lengths\_product\_dim
        \ / / Integer remainder-\text{MOD} operation inlined
    END DO
```
1.2.1 Procedure \textit{CreateLatticeNetwork}

Here is the routine \textit{CreateLatticeNetwork}. I can not comment too much on it since it is rather involved and coded according to my own standards. I also partially had future parallelization with HPF or OpenMP in mind, but this is a different issue.

The routine will first allocate enough space (meaning full lattice size) for all needed arrays, then it will calculate the connectivity (heads-tails array) for the lattice and the coordinates of the nodes, all the while carefully considering the boundary conditions and marking each node and arc with the correct status. The number of nodes and arcs after this phase will correspond closely to the desired dilution. Then, the routine calls a connected-components-labeling algorithm from \textit{Graph Algorithms} to find and label the largest connected-component in the graph. Finally, it will extract this component, throwing away all arcs and nodes which do not belong to it (suitably renumbering them). If there is no dilution then the rather costly extraction step is skipped.

\begin{verbatim}
(CreateLatticeNetwork 1.6) \equiv

SUBROUTINE CreateLatticeNetwork()
IMPLICIT NONE
INTEGER (KIND = i_up), DIMENSION (:), ALLOCATABLE :: points_count, nodes_labels, nodes_count
/* We need some temporary buffers for reallocating some of the arrays later on: */
INTEGER (KIND = i_up), DIMENSION (:,:), ALLOCATABLE :: heads_tails_buffer
REAL, DIMENSION (:,:), ALLOCATABLE :: coords_buffer
INTEGER (KIND = i_byte), DIMENSION (:,:), ALLOCATABLE :: status_buffer
INTEGER (KIND = i_up) :: n_points, n_links, n_special_points, n_special_links, n_selected_nodes,
n_selected_arcs \quad \text{// Counters of sites and edges}
INTEGER (KIND = i_up) :: link, point, node, arc, neighbor_point, node_counter, arc_counter,
point_counter, point_temp, selected_label, selected_size
INTEGER :: dim, direction, dim_temp \quad \text{// Temporaries and loop counters}
INTEGER (KIND = i_byte) :: arc_status, node_status \quad \text{// Temporaries}
INTEGER (KIND = i_up), DIMENSION (n_dim) :: lengths_product
INTEGER, DIMENSION (n_dim) :: neighbor_coords, point_coords \quad \text{// Temporaries}
REAL :: dice, real_temp \quad \text{// Random dice for dilution}
LOGICAL :: dilute_nodes, dilute_arcs, extraction_success, node_on_plate, special_arc, renumber_nodes
\quad \text{// Indicators}

(AllocateArrays 1.8)
(AssignValuesToArray 1.9)
(SelectConnectedComponent 1.10)

END SUBROUTINE CreateLatticeNetwork
\end{verbatim}

This code is used in section 1.1.
1.2.2 Initial Allocation or Network Arrays

The following piece of code finds the full (maximal) size of the lattice as well as the maximal number of special arcs and nodes, and then allocates the network arrays with these full sizes. It would have been nice if all reallocations were avoided in the case of a fully connected network. But counting the number of arcs that will be present under the plethora of boundary conditions and general lattices I allow turned out to be a formiddable task, especially because of the arcs that cross the box boundaries (which can, but do not have to be, wrapped around in a periodic fashion). The following piece of code was placed below and caused a lot of debugging trouble because of double-counting of certain edge and vertex arcs:

\[
\text{(Buggy Code 1.7) } \equiv \\
\text{do dim }= 1, n_{\text{dim}} \quad // \text{Check boundaries for crossing arcs} \\
\text{PeriodicArcs: if (not boundaries_{bc \text{ periodic}, dim}) then } \quad // \text{Plate boundary} \\
\text{do direction }= 1, n_{\text{directions}} \\
\quad \text{if (displacements_{dim, direction} } \neq 0) \text{ then } \quad // \text{An arc that crosses the box!} \\
\quad \quad n_{\text{links}} = n_{\text{links}} - n_{\text{points}} / \text{lengths}_{\text{dim}} \quad // \text{These are absent} \\
\quad \text{end if} \\
\text{end do} \\
\text{end if PeriodicArcs} \\
\text{end do}
\]
Here is the code which works and is simple, but does require reallocation in certain cases:

\[
\langle \text{AllocateArrays} \rangle \\
\text{lengths} \cdot \text{product}_k = 1 \quad // \quad \text{lengths} \cdot \text{product}_k = \prod_{i=1}^{k} \text{lengths}_i \\
\text{DO dim = 2, n_dim} \quad // \quad \text{lengths} \cdot \text{product} \text{ is needed when numbering the nodes} \\
\quad \text{lengths} \cdot \text{product}_{\text{dim}} = \text{lengths} \cdot \text{product}_{\text{dim-1}} \times \text{lengths}_{\text{dim-1}} \\
\text{END DO} \\
\text{n_points} = \text{PRODUCT(lengths)} \quad // \quad \text{Total number of sites in the lattice} \\
\text{n_links} = \text{n_points} \star \text{n_directions} \quad // \quad \text{Total number of connections} \\
\text{sources} = 0; \text{ sinks} = 0; \\
\quad // \quad \text{The node numbers for the sources and sinks-needed later on} \\
\text{/* Boundary conditions have to be scanned for possible plate or source/sink boundary conditions,} \\
\text{which require special arcs and nodes:*} \\
\text{n_special_points} = 0; \text{n_special_links} = 0; \\
\text{DO dim = 1, n_dim} \quad // \quad \text{Check boundaries for all dimensions} \\
\text{PlateArcs: IF (boundaries bc_plate, dim) THEN} \quad // \quad \text{Plate boundary} \\
\quad \text{DO direction = 1, n_directions} \\
\quad \quad \text{IF (displacements dim, direction} \equiv 0) \quad // \quad \text{An arc in plate!} \\
\quad \quad \quad \text{n_special_links} = \text{n_special_links} + 2 \star \text{n_points} / \text{lengths}_{\text{dim}} \quad // \quad \text{Zero-cost arcs} \\
\quad \quad \quad \text{n_links} = \text{n_links} - 2 \star \text{n_points} / \text{lengths}_{\text{dim}} \quad // \quad \text{Possible special arcs} \\
\quad \quad \text{END IF} \\
\quad \text{END DO} \\
\text{END IF PlateArcs} \\
\text{SourcesSinks: IF (boundaries bc_source_sink, dim) THEN} \quad // \quad \text{Add a source} \\
\quad \text{IF (source_layer_thickness dim > 0) THEN} \\
\quad \quad \text{n_special_links} = \text{n_special_links} + \text{source_layer_thickness}_{\text{dim}} \star \text{n_points} / \text{lengths}_{\text{dim}} \quad // \quad \text{Arcs from S} \\
\quad \quad \text{n_special_points} = \text{n_special_points} + 1 \quad // \quad \text{S is a special nodes} \\
\quad \text{END IF} \\
\quad \text{IF (sink_layer_thickness dim > 0) THEN} \quad // \quad \text{Add a sink} \\
\quad \quad \text{n_special_links} = \text{n_special_links} + \text{sink_layer_thickness}_{\text{dim}} \star \text{n_points} / \text{lengths}_{\text{dim}} \quad // \quad \text{Arcs from S} \\
\quad \quad \text{n_special_points} = \text{n_special_points} + 1 \quad // \quad \text{S is a special nodes} \\
\quad \text{END IF} \\
\text{END IF SourcesSinks} \\
\text{END IF} \\
\text{/* Now we need to allocate all the arrays with their maximal possible sizes: */} \\
\text{ALLOCATE (nodes_coords (n_dim, -n_special_points : n_points), stat = error_status)} \\
\text{CALL RecordAllocation (n_elements = n_dim \star (n_points + n_special_points + 1), mold = 1.0,} \\
\text{ caller = "CreateLatticeNetwork", alloc_status = error_status)} \\
\text{nodes_coords (1 : n_dim, 0) = 0.0 \quad // \quad \text{Dummy node}} \\
\text{ALLOCATE (nodes_status (n_special_points : n_points), stat = error_status)} \\
\text{CALL RecordAllocation (n_elements = (n_points + n_special_points + 1), mold = 1i_byte,} \\
\text{ caller = "CreateLatticeNetwork", alloc_status = error_status)} \\
\text{ALLOCATE (heads_tails (2, -n_special_links : n_links), stat = error_status)} \\
\text{CALL RecordAllocation (n_elements = 2 \star (n_links + n_special_links + 1), mold = 1i_wp,} \\
\text{ caller = "CreateLatticeNetwork", alloc_status = error_status)} \\
\text{heads_tails (1 : 2, 0) = 0 \quad // \quad \text{Initialize zeroth arc to be loop to the dummy node}} \\
\text{ALLOCATE (arc_status (n_special_links : n_links), stat = error_status)} \\
\text{CALL RecordAllocation (n_elements = (n_links + n_special_links + 1), mold = 1i_byte,} \\
\text{ caller = "CreateLatticeNetwork", alloc_status = error_status)} \\
\text{nodes_status = empty_node, arc_status = empty_arc \quad // \quad \text{Initialize dummy ones}
This code is used in section 1.6.

1.2.3 Lattice Geometry Calculation

The connectivity of the network is calculated in an involved way because we allow for both node and arc dilution. The major loop is over nodes, and for each node I calculate its neighbors and assign the proper heads-and-tails and nodal coordinates. This piece of code is a monster when it comes to parallelization due to the constant use of indirections:
\[\text{AssignValuesToArrays 1.9} \equiv\]
\[
dilute_{\text{nodes}} = (\text{node\_dilution} < 1.0) \land (\text{node\_dilution} > 0.0) \quad // \text{Dilution is in (0,1)}
\]
\[
dilute_{\text{arcs}} = (arc\_dilution < 1.0) \land (arc\_dilution > 0.0) \quad // \text{Dilution is in (0,1)}
\]

/* When nodes are diluted, their numbering changes from the default because some of them will be
absent (dilated). We need to make a mask and a prefix counting operation to find the new
numbering of the nodes, and store this for later use: */

\textbf{IF} (dilute\_nodes) \textbf{THEN} // points\_count = COUNT\_PREFIX(points\_mask) in essence
\textbf{ALLOCATE} (points\_count(n\_points), \textit{stat} = error\_status)
\textbf{CALL} RecordAllocation(n\_elements = n\_points, mold = 1\_wp, caller = "CreateLatticeNetwork",
alloc\_status = error\_status)
\textbf{END IF}

\textbf{points\_count} = 0 // Zero means a diluted-empty node
\textbf{points\_counter} = 0 // Just a counter for points\_count

\textbf{DilateChannels: DO} point = 1, n\_points // See which nodes are diluted
\textbf{node\_on\_plate} = T
\textbf{POINT\_TO\_COORDS}(point, point\_coords, dim\_temp, point\_temp) // Get the coordinates

\textbf{PlateNodes: DO} dim = 1, n\_dim // Is this a plate node?
\textbf{IF} (boundaries \_bc\_plate, dim) \textbf{THEN} // Is this point on a plate?
\textbf{IF} (point\_coords \_dim \equiv 1 | point\_coords \_dim \equiv lengths\_dim)
\textbf{node\_on\_plate} = T // Plate nodes are not diluted
\textbf{END IF}
\textbf{END IF}

\textbf{END DO PlateNodes}

\textbf{IF} (!node\_on\_plate) \textbf{THEN} // Dilute the node if dice says so
\textbf{CALL} RandomUniform(dice) // In [0,1]
\textbf{IF} (dice > node\_dilution) \textbf{CYCLE} DilateChannels // Throw away the node
\textbf{END IF}
\textbf{point\_counter} = point\_counter + 1
\textbf{points\_count\_point} = point\_counter
\textbf{END DO DilateChannels}
\textbf{END IF}

n\_nodes = 0; n\_arcs = 0; n\_special\_nodes = 0; n\_special\_arcs = 0;
// Initialize

\textbf{SourceSinkNodes: DO} dim = 1, n\_dim // Check the boundary conditions for special nodes
\textbf{IF} (boundaries \_bc\_source\_sink, dim) \textbf{THEN} // Initialize the source and sink
\textbf{IF} (source\_layer\_thickness \_dim > 0) \textbf{THEN} // Add S
n\_special\_nodes = n\_special\_nodes + 1 // S node
sources\_dim = -n\_special\_nodes // Save this location for later use
nodes\_status = -n\_special\_nodes = source\_node // This is a source node
\textbf{nodes\_coords} \_1..\_n\_dim, n\_special\_nodes + 0.5 * REAL(lengths\_dim + 1)
// An attempt to plot ST nodes nicely.
\textbf{nodes\_coords} \_dim, n\_special\_nodes = 1.0 - 0.1 * REAL(lengths\_dim)
// Place the sink or source 10 percent to the side of the boundary
\textbf{END IF}
\textbf{IF} (sink\_layer\_thickness \_dim > 0) \textbf{THEN} // Add T
n\_special\_nodes = n\_special\_nodes + 1 // T node
sinks\_dim = -n\_special\_nodes // Store for later use
nodes\_status = -n\_special\_nodes = sink\_node // This is a sink node
\textbf{nodes\_coords} \_1..\_n\_dim, n\_special\_nodes = 0.5 * REAL(lengths\_dim + 1)
\textbf{nodes\_coords} \_dim, n\_special\_nodes = 1.1 * REAL(lengths\_dim) // For plotting
\textbf{END IF}
\textbf{END IF}
\textbf{END IF}
END DO SourceSinkNodes

/\* Now I start the major loop over nodes and add the arcs to the neighbors one by one: */
ForAllPoints: DO point = 1, n_points // FORALL masked with points_mask
    node_status = interior_node
    IF (dilute_nodes) THEN // Check dice mask for dilution
        IF (points_count_point = 0) CYCLE ForAllPoints // This site is diluted
        n_nodes = points_count_point // The new node number
    ELSE // Go on
        n_nodes = point // No need to renumber nodes
    END IF

.PointToCoords(point, point_coords, dim_temp, point_temp) // Get the coordinates
nodes_coords: DO dim = 1, n_dim // Check if this is a plate or injection node
    IF (boundaries bc_plate, dim) THEN // Check for plate node
        IF (point_coords_dim = 1 | point_coords_dim = lengths_dim) THEN
            node_status = MAX(node_status, plate_node) // Plate node
        END IF
    END IF

    IF (boundaries bc_injection, dim) THEN // Is this node an injection site?
        IF (point_coords_dim = injection_layer_thickness_dim) THEN
            node_status = MAX(node_status, injection_node) // Injection site
        ELSE IF (point_coords_dim = outflow_layer_thickness_dim) THEN
            node_status = MAX(node_status, outflow_node) // Outflow site
        END IF
    END IF

    IF (boundaries bc_source_sink, dim) THEN // Check for connections to ST nodes
        IF (point_coords_dim = source_layer_thickness_dim) THEN
            // Connect to S-here all source arcs start at S and all sink arcs end at T by convention
            node_status = MAX(node_status, to_source_node)
            n_special_arcs = n_special_arcs + 1
            arcs_status = arcs_status = MAX(arcs_status - n_special_arcs, source_arc)
        IF (dilute_nodes) THEN
            heads_tails_1:2, -n_special_arcs = (sources_dim, points_count_point)
        ELSE
            heads_tails_1:2, -n_special_arcs = (sources_dim, point)
        END IF
    ELSE IF (point_coords_dim = lengths_dim - sink_layer_thickness_dim) THEN // Connect to T
        node_status = MAX(node_status, to_sink_node)
        n_special_arcs = n_special_arcs + 1
        arcs_status = arcs_status = MAX(arcs_status - n_special_arcs, sink_arc)
        IF (dilute_nodes) THEN
            heads_tails_1:2, -n_special_arcs = (point_count_point, sinks_dim)
        ELSE
            heads_tails_1:2, -n_special_arcs = (point, sinks_dim)
        END IF
    END IF
END DO PlateOrInjectionNodes

nodes_status = node_status // Kind of node
Near-Neighbour Lattice Networks

 /* Now I start adding the arcs that originate at this node (have the node as their head) one by one by looking at all the directions specified in displacements: */

 ForAllDirections: do direction = 1, n_directions // Different directions
 arc_status = interior_arc
 special_arc = F // An ordinary arc
 neighbor_coords = point_coords + displacements, direction // The neighbour coordinates

 Plates: do dim = 1, n_dim // Check for special arcs
 BoundaryPlate: if (boundaries_bc_plate, dim) then // On the plate?
 if ((point_coordsᵈᵐ ≡ lengthsᵈᵐ) \& (neighbor_coordsᵈᵐ ≡ lengthsᵈᵐ)) \| ((point_coordsᵈᵐ ≡ 1) \& (neighbor_coordsᵈᵐ ≡ 1)) then
 special_arc = T // A plate arc
 arc_status = MAX(plate_arc, arc_status) // I will not dilute plate arcs
 end if
 end if BoundaryPlate
 end do Plates

 if (~special_arc \& dilute_arcs) then // Throw dilution dice for arcs
 call RandomUniform(dice) // In [0,1]
 if (dice > arc_dilation) cycle ForAllDirections // This arc is diluted
 end if
 end

 Periodics: do dim = 1, n_dim // Check if crossing a periodic boundary
 BoundaryCrossing: if ((neighbor_coordsᵈᵐ > lengthsᵈᵐ) \| (neighbor_coordsᵈᵐ < 1)) then // Outside of square
 if (boundaries_bc_periodic, dim) then
 if (neighbor_coordsᵈᵐ > lengthsᵈᵐ) \&

 neighbor_coordsᵈᵐ = MOD(neighbor_coordsᵈᵐ, lengthsᵈᵐ)
 // Wrap around the lattice in this dimension

 if (neighbor_coordsᵈᵐ < 1) \&

 neighbor_coordsᵈᵐ = lengthsᵈᵐ + MOD(neighbor_coordsᵈᵐ, lengthsᵈᵐ) // Wrap around
 arc_status = MAX(periodic_bc_arc, arc_status) // A wrap-around arc
 else
 cycle ForAllDirections // This arc is not allowed
 end if
 end if BoundaryCrossing
 end do Periodics

COORDS_TO_POINT(neighbor_coords, neighbor_point, dim_temp)

 if (dilute_nodes) then // Nodes are diluted--there is danger of empty tail! Also, the nodes were
 renumbered in points_count
 if (points_countneighbor_point ≡ 0) cycle ForAllDirections // The tail is empty
 if (~special_arc) then // A regular arc
 n_arcs = n_arcs + 1
 heads_fails 1:2, n_arcs = (points_count_point, points_countneighbor_point)
 arcs_status n_arcs = arc_status
 else // A special arc
 n_special_arcs = n_special_arcs + 1
 heads_fails 1:2, n_special_arcs = (points_count_point, points_countneighbor_point)
 arcs_status n_special_arcs = arc_status
 end if
 else // There is no node dilution, danger of empty tail or node renumbering
 if (~special_arc) then

\[ n_{\text{arcs}} = n_{\text{arcs}} + 1 \]
\[ \text{heads.tails}_{1:2}, n_{\text{arcs}} = (\text{point, neighbor.point}) \]
\[ \text{arcs.status}_{n_{\text{arcs}}} = \text{arc.status} \]
ELSE
\[ n_{\text{special.arcs}} = n_{\text{special.arcs}} + 1 \]
\[ \text{heads.tails}_{1:2}, -n_{\text{special.arcs}} = (\text{point, neighbor.point}) \]
\[ \text{arcs.status}_{-n_{\text{special.arcs}}} = \text{arc.status} \]
END IF
END IF
END DO ForAllDirections
END DO ForAllPoints
IF (dilate.nodes) THEN  // Free the prefix count array
CALL RecordAllocation(n.elements = -size(points.count, i.wp), mold = 1_i.wp)
DEALLOCATE(points.count)
END IF

This code is used in section 1.6.

1.2.4 Connected-Component Extraction

The following code section calls the routine ConnectedComponents to find the largest connected component and extracts it. This is skipped if there was no dilution. Then the code recreate (reallocates) the lattice network with just the largest connected component extracted from the original lattice. Even when there is no dilution for certain combinations of boundary conditions the network arrays will need to be reallocated:
\(\text{SelectConnectedComponent} \ 1.10\) \equiv

\[\begin{align*}
n_{\text{injection} \_ \text{nodes}} & = 0 \quad \text{// We have to count these because of the constraint } \sum b_i = 0 \\
n_{\text{outflow} \_ \text{nodes}} & = 0
\end{align*}\]

\(\text{if (dilate} \_ \text{nodes} \mid \text{dilate} \_ \text{areas}) \text{ then } // \text{Extract a single connected component}
\)

\begin{verbatim}
WRITE (message_log_unit, *) "Starting connected-component extraction with:"
WRITE (message_log_unit, *) "n_arcs", n_arcs, " n_nodes", n_nodes, " n_special_nodes",
   " n_special_arcs", " n_special_arcs"
ALLOCATE (nodes_mask (-n_special_nodes : n_nodes), stat = error \_ status)
CALL RecordAllocation (n_elements = n_special_nodes + n_nodes + 1, mold = T_{i,wp},
   caller = "CreateNetwork", alloc \_ status = error \_ status)
ALLOCATE (arc \_ mask (-n_special_arcs : n_arcs))
CALL RecordAllocation (n_elements = n_special_arcs + n_arcs + 1, mold = T_{i,wp},
   caller = "CreateNetwork", alloc \_ status = error \_ status)
ALLOCATE (nodes \_ labels (-n_special_nodes : n_nodes), stat = error \_ status)
   // The connected-components labels for the nodes
CALL RecordAllocation (n_elements = n_special_nodes + n_nodes + 1, mold = l_{i,wp},
   caller = "CreateLatticeNetwork", alloc \_ status = error \_ status)
node \_ labels (0) = 0 // Dummy node
\end{verbatim}

\begin{verbatim}
SELECT CASE (component \_ extraction)
   CASE (B', B')
   CALL BackboneCycles (heads \_ tails = heads \_ tails(1 : n_special_arcs : n_arcs),
      node \_ offset = n_special_nodes, supernodes = nodes \_ labels, largest \_ supernode = selected \_ label,
      largest \_ supernode \_ size = selected \_ size) // Label the backbone cycles
   WRITE (message_log_unit, *) "The largest backbone component has: ", selected \_ size, "nodes"
   CASE DEFAULT
   CALL ConnectedComponents (heads \_ tails = heads \_ tails(1 : n_special_arcs : n_arcs),
      node \_ offset = n_special_nodes, labels = nodes \_ labels, largest \_ component \_ label = selected \_ label,
      largest \_ component \_ size = selected \_ size) // Label the connected components
   WRITE (message_log_unit, *) "The largest connected component has: ", selected \_ size, "nodes"
\end{verbatim}

ENDSELECT

\[n_{\text{selected} \_ \text{nodes}} = 0 \quad // \text{The number of nodes in the extracted component. All special nodes and arcs must be in the selected connected component, so I don't count them explicitly}
\]

\[n_{\text{selected} \_ \text{arcs}} = 0 \quad // \text{All special arcs are present by default}
\]

\[\text{extraction} \_ \text{success} = T\]

// Now we mask as \(T\) all the nodes and arcs that belong to the largest component. I also check whether this component contains all the special nodes (sources and sinks and plate nodes in it), and whether there is at least one injection and outflow site in it (if these were requested in boundaries). If not, this means the lattice is too dilute and the program is terminated with a critical error: */

\(\text{MaskSelectedNodes: do node = -n_{\text{special} \_ \text{nodes}}, n_{\text{nodes}} // Select only the nodes in the major component}
\)

\(\text{SelectNodes: if (nodes} \_ \text{labels\_node } \equiv \text{selected} \_ \text{label}) \text{ then } // \text{Select this node}
\)

\begin{verbatim}
nodes \_ mask \_ node = T_{i,wp}
   if (node > 0) n_{\text{selected} \_ \text{nodes}} = n_{\text{selected} \_ \text{nodes}} + 1
   if (nodes \_ status \_ node \equiv \text{injection} \_ \text{node}) \text{ then } // \text{Count the injection/outflow nodes}
      n_{\text{injection} \_ \text{nodes}} = n_{\text{injection} \_ \text{nodes}} + 1
   else if (nodes \_ status \_ node \equiv \text{outflow} \_ \text{node}) \text{ then } // \text{We want to count these}
\end{verbatim}
\[ n_{\text{outflow~nodes}} = n_{\text{outflow~nodes}} + 1 \]

\textbf{ELSE}  // If this was a special or plate node, then abort the mission
\textbf{IF} \((\text{node} < 0) \lor (\text{nodes~status\_node} \equiv \text{plate\_node})\) \textbf{THEN}
\textbf{EXIT MaskSelectedNodes}  // This is a critical error!
\textbf{END IF}
\textbf{nodes\_mask\_node} = \text{FAIL}
\textbf{END IF SelectNodes}
\textbf{END DO MaskSelectedNodes}

\textbf{CheckConnectedness:} \textbf{IF} \(\neg \text{extraction~success} \) \textbf{THEN}  // The graph is too diluted
\textbf{CALL CriticalError} (message = "The sources/sinks and/or plate nodes were not in the largest connected component", caller = "CreateLatticeNetwork")
\textbf{ELSE}
\textbf{DO} \text{dim} = 1, \text{n\_dim}
\textbf{IF} \(\text{boundaries}_{\text{injection}, \text{dim}}\) \textbf{THEN}  // There were injection boundaries
\textbf{IF} \((\text{injection\_layer\_thickness\_dim} > 0) \land (\text{n\_injection\_nodes} \equiv 0)\)
\textbf{CALL CriticalError} (message = "No injection sites in the largest component", caller = "CreateLatticeNetwork")
\textbf{ELSE}
\textbf{IF} \((\text{outflow\_layer\_thickness\_dim} > 0) \land (\text{n\_outflow\_nodes} \equiv 0)\)
\textbf{CALL CriticalError} (message = "No outflow sites in the largest component", caller = "CreateLatticeNetwork")
\textbf{END IF}
\textbf{END IF CheckConnectedness}
\textbf{END DO}
\textbf{CALL RecordAllocation} \((\text{n\_elements} = -\text{size(\text{nodes\_labels}}\_\text{i\_wp}), \text{mold} = 1\_\text{i\_wp})\)
\textbf{DEALLOCATE} \(\text{nodes\_labels}\)  // We don't need these any more

\textbf{MaskSelectedArcs: DO} \text{arc} = 1, \text{n\_arcs}  // Mask empty arcs
\textbf{IF} \(\text{nodes\_mask(heads\_tails}_{\text{1, arc}}) \land \text{nodes\_mask(heads\_tails}_{\text{2, arc}})\) \textbf{THEN}
\text{n\_selected\_arcs} = \text{n\_selected\_arcs} + 1
\text{arc\_mask\_arc} = \text{FAIL}  // Arc is in the right component
\textbf{ELSE}
\text{arc\_mask\_arc} = \text{FAIL}  // Throw away this arc
\textbf{END IF}
\textbf{END DO MaskSelectedArcs}

\textbf{/*} Now I check whether reallocation, packing and node renumbering are really necessary and then perform them if they are. This section of code is really too complicated for what it is worth, since the cost of the reallocation is not that great: */
\textbf{renumber\_nodes} = \text{FAIL}  // Are nodes renumbered?
\textbf{IF} \((\text{n\_selected\_nodes} \neq \text{n\_nodes}) \lor (\text{n\_special\_nodes} \neq \text{n\_special\_points})\) \textbf{THEN}  // Pack
\textbf{WRITE} (message\_log\_unit, *) "Reallocating node arrays..."
(AllocateNodeBuffers 1.19)
\textbf{IF} \((\text{n\_selected\_nodes} \neq \text{n\_nodes})\) \textbf{THEN}  // Repacking/renumbering of nodes is needed
\textbf{renumber\_nodes} = \text{FAIL}
\textbf{WRITE} (message\_log\_unit, *) "Repacking node arrays..."
\textbf{ALLOCATE} \((\text{n\_nodes\_count} = \text{n\_special\_nodes} \_\text{n\_nodes}), \text{STAT} = \text{error\_status})\)
\textbf{CALL RecordAllocation} \((\text{n\_elements} = \text{n\_special\_nodes} + \text{n\_nodes} + 1, \text{mold} = 1\_\text{i\_wp},\)\)
\text{caller = "CreateLatticeNetwork", alloc\_status = error\_status})
(ReallocatePackedNodes 1.13)
(PackNodeArraysIntoBuffers 1.21)
ELSE  // Rather unlikely!
  (CopyNodeArraysIntoBuffers 1.22)
END IF
(ReallocateNodeArraysUsingBuffers 1.20)
END IF
CALL RecordAllocation (n_elements = -SIZE(nodes_mask, i_wp), mold = T1_wp)
DEALLOCATE (nodes_mask)
IF ((n_selected_arcs ≠ n_links) || (n_special_arcs ≠ n_special_links)) THEN
  // We need to repackage the arc arrays into smaller arrays
WRITE (message_log_unit, *) "Reallocating arc arrays...
  (AllocateArcBuffers 1.11)
IF (n_selected_arcs ≠ n_arcs) THEN
  WRITE (message_log_unit, *) "Repacking arc arrays...
  IF (renumber_nodes) THEN
    (PackArcsWithRenumbering 1.14)
  ELSE
    (PackArcsWithoutRenumbering 1.15)
  END IF
ELSE  // Very unlikely!
  IF (renumber_nodes) THEN
    (CopyArcsWithRenumbering 1.16)
  ELSE
    (CopyArcsWithoutRenumbering 1.17)
  END IF
END IF
(ReallocateArcArraysUsingBuffers 1.12)
END IF
IF (renumber_nodes) THEN  // Deallocate the prefix-count array
  CALL RecordAllocation (n_elements = -SIZE(nodes_count, i_wp), mold = l1_wp)
DEALLOCATE (nodes_count)  // We don’t need these any more
END IF
CALL RecordAllocation (n_elements = -SIZE(arcs_mask, i_wp), mold = T1_wp)
DEALLOCATE (arcs_mask)
n_nodes = n_selected_nodes  // Update n
n_arcs = n_selected_arcs  // Update m
ELSE  // If there is no dilution there is no need for connected-components extraction
  n_selected_nodes = n_nodes  // All nodes are connected
  n_selected_arcs = n_arcs  // All arcs are in the selected component
ENDIF
IF ((n_nodes ≠ n_points) || (n_special_nodes ≠ n_special_points)) THEN  // Nodes
  WRITE (message_log_unit, *) "Reallocating node arrays...
  (AllocateNodeBuffers 1.19)
  (CopyNodeArraysIntoBuffers 1.22)
  (ReallocateNodeArraysUsingBuffers 1.20)
ENDIF
IF ((n_arcs ≠ n_links) || (n_special_arcs ≠ n_special_links)) THEN  // Reallocate for arcs
  WRITE (message_log_unit, *) "Reallocating arc arrays...
  (AllocateArcBuffers 1.11)
  (CopyArcsWithoutRenumbering 1.17)
1.2.5 Reallocating and Packing the Network Arrays

Now that we know which nodes and arcs to keep and which to throw away, we need to actually pack the network arrays and reallocate them with the correct sizes. Since Fortran does not allow direct reallocation, this piece here is rather messy and long, and temporary reallocation buffers must be used at the cost of one extra memory transfer. Care must also be taken because nodes are again renumbered because some will be thrown away, and the head and tails arrays must be updated to reflect this. In HPF, the routine PACK or an analog will be used to do this, at a rather large cost of memory movement and lots of indirections.

This code will allocate the copy buffers required for packing the arc-arrays:

```fortran
ALLOCATE (heads_tails_buffer(2, -n_special_arcs : n_selected_arcs), stat = error_status)
CALL RecordAllocation (n_elements = 2 * (n_special_arcs + n_selected_arcs + 1), mold = 1i_wp, 
caller = "CreateLatticeNetwork", alloc_status = error_status)
ALLOCATE (status_buffer(-n_special_arcs : n_selected_arcs), stat = error_status)
CALL RecordAllocation (n_elements = n_special_arcs + n_selected_arcs + 1, mold = 1i_byte, 
caller = "CreateLatticeNetwork", alloc_status = error_status)
heads_tails_buffer(1:2, 0) = 0    ! Null element
status_buffer(0) = empty_arc    ! Dummy arc
```

This code is used in section 1.10.
Once the buffers have been assigned values (see macros below), the network arrays need to be reallocated and the buffers copied into the new arrays:

\[
\langle \text{Reallocation of Buffers 1.12} \rangle \equiv \\
\text{CALL RecordAllocation} (n_{\text{elements}} = -\text{SIZE}(\text{heads_tails}, i_{\text{wp}}), ~ \text{mold} = 1_{\text{wp}}) \\
\text{DEALLOCATE}(\text{heads_tails}) \quad /\!/ \text{Start the reallocation process} \\
\text{ALLOCATE}(\text{heads_tails}(2, -n_{\text{special arcs}} : n_{\text{selected arcs}}), ~ \text{STAT} = \text{error status}) \quad /\!/ \text{New size} \\
\text{CALL RecordAllocation} (n_{\text{elements}} = 2 \ast (n_{\text{special arcs}} + n_{\text{selected arcs}} + 1), ~ \text{mold} = 1_{\text{wp}}, \\
\quad \text{caller} = \text{"CreateLatticeNetwork"}, ~ \text{alloc_status} = \text{error status}) \\
\text{heads_tails} = \text{heads_tails_buffer} \quad /\!/ \text{Copy the buffer back into the array} \\
\text{CALL RecordAllocation} (n_{\text{elements}} = -\text{SIZE}(\text{heads_tails_buffer}, i_{\text{wp}}), ~ \text{mold} = 1_{\text{wp}}) \\
\text{DEALLOCATE}(\text{heads_tails_buffer}) \\
\text{CALL RecordAllocation} (n_{\text{elements}} = -\text{SIZE}(\text{arcs_status}, i_{\text{wp}}), ~ \text{mold} = 1_{\text{byte}}) \\
\text{DEALLOCATE}(\text{arcs_status}) \quad /\!/ \text{Start the reallocation process} \\
\text{ALLOCATE}(\text{arcs_status}(-n_{\text{special arcs}} : n_{\text{selected arcs}}), ~ \text{STAT} = \text{error status}) \quad /\!/ \text{New size} \\
\text{CALL RecordAllocation} (n_{\text{elements}} = n_{\text{special arcs}} + n_{\text{selected arcs}} + 1, ~ \text{mold} = 1_{\text{byte}}, \\
\quad \text{caller} = \text{"CreateLatticeNetwork"}, ~ \text{alloc_status} = \text{error status}) \\
\text{arcs_status} = \text{status_buffer} \quad /\!/ \text{Copy the buffer} \\
\text{CALL RecordAllocation} (n_{\text{elements}} = -\text{SIZE}(\text{status_buffer}, i_{\text{wp}}), ~ \text{mold} = 1_{\text{byte}}) \\
\text{DEALLOCATE}(\text{status_buffer})
\]

This code is used in section 1.10.

When nodes are packed, they are renumbered, according to \textit{nodes_labels} = \textit{COUNT PREFIX}(\textit{nodes_mask}).

Here is this operation inlined:

\[
\langle \text{Renumber of Packed Nodes 1.13} \rangle \equiv \\
\text{nodes_count}_0 = 0 \\
\text{DO } \text{node} = -n_{\text{special nodes}}, ~ -1 \\
\quad \text{nodes_count}_{\text{node}} = \text{node} \quad /\!/ \text{Since they are all present} \\
\text{END DO} \\
\text{node_counter} = 0 \\
\text{DO } \text{node} = 1, ~ n_{\text{nodes}} \quad /\!/ \text{COUNT PREFIX} \\
\quad \text{IF } (\text{nodes}_{\text{mask}}_{\text{node}}) \text{ THEN } \quad /\!/ \text{Count this node} \\
\quad \quad \text{node_counter} = \text{node_counter} + 1 \\
\quad \quad \text{nodes_count}_{\text{node}} = \text{node_counter} \\
\quad \text{END IF} \quad /\!/ \text{No need to count the rest of them} \\
\text{END DO}
\]

This code is used in section 1.10.
How the arc or node arrays are packed into the buffer depends on whether there is dilution or not and whether nodes were renumbered (which has to be reflected in the new heads and tails arrays). Here is the piece of code that will assign values to the buffers when there is dilution:

\[
\text{PackArcsWithRenumbering } 1.14 \equiv \\
\text{do } \text{arc } = -n_{\text{special arcs}}, -1 \quad // \text{Not a real pack—just a copy} \\
\text{heads\_tails\_buffer } 1:2, \text{ arc } = \text{nodes\_count(heads\_tails } 1:2, \text{ arc)} \quad // \text{Renumber} \\
\text{end do} \\
\text{status\_buffer } -n_{\text{special arcs}}(:-1) = \text{arcs\_status } -n_{\text{special arcs}}(:-1) \quad // \text{No need to PACK} \\
\text{arc\_counter } = 0 \\
\text{packArcsRenumbered: do } \text{arc } = 1, n_{\text{arcs}} \\
\quad \text{if } (\text{arc\_mask } \text{arc}) \quad // \text{Pack this arc} \\
\quad \quad \text{arc\_counter } = \text{arc\_counter } + 1 \\
\quad \quad \text{heads\_tails\_buffer } 1:2, \text{ arc\_counter } = \text{nodes\_count(heads\_tails } 1:2, \text{ arc)} \\
\quad \quad \text{status\_buffer } \text{arc\_counter } = \text{arcs\_status } \text{arc} \\
\quad \end if \\
\text{end do PackArcsRenumbered} \\
\]

This code is used in section 1.10.

When nodes are not renumbered packing is a bit easier:

\[
\text{PackArcsWithoutRenumbering } 1.15 \equiv \\
\text{do } \text{arc } = -n_{\text{special arcs}}, -1 \quad // \text{Not a real pack—just a copy} \\
\text{heads\_tails\_buffer } 1:2, \text{ arc } = \text{heads\_tails } 1:2, \text{ arc} \\
\text{end do} \\
\text{status\_buffer } -n_{\text{special arcs}}(:-1) = \text{arcs\_status } -n_{\text{special arcs}}(:-1) \\
\text{arc\_counter } = 0 \\
\text{PackArcs: do } \text{arc } = 1, n_{\text{arcs}} \\
\quad \text{if } (\text{arc\_mask } \text{arc}) \quad // \text{Pack this arc} \\
\quad \quad \text{arc\_counter } = \text{arc\_counter } + 1 \\
\quad \quad \text{heads\_tails\_buffer } 1:2, \text{ arc\_counter } = \text{heads\_tails } 1:2, \text{ arc} \\
\quad \quad \text{status\_buffer } \text{arc\_counter } = \text{arcs\_status } \text{arc} \\
\quad \end if \\
\text{end do PackArcs} \\
\]

This code is used in section 1.10.

When there is no dilution, there is no need to perform a connected-components extraction and to pack the arrays, but simply reallocate them by copying the old contents to the buffers, deallocating, again allocating and recopying the buffer back into the arrays:

\[
\text{CopyArcsWithRenumbering } 1.16 \equiv \\
\text{do } \text{arc } = -n_{\text{special arcs}}, n_{\text{selected arcs}} \\
\text{heads\_tails\_buffer } 1:2, \text{ arc } = \text{nodes\_count(heads\_tails } 1:2, \text{ arc)} \\
\text{end do} \\
\text{status\_buffer } = \text{arcs\_status } -n_{\text{special arcs}}:n_{\text{selected arcs}} \\
\]

This code is used in section 1.10.
\langle \text{Copy Arcs Without Renumbering} \, 1.17 \rangle \equiv

\begin{align*}
\text{heads\_tails\_buffer} & = \text{heads\_tails}_{1:2} - n_{\text{special\_arcs}} \times n_{\text{selected\_arcs}} \\
\text{status\_buffer} & = \text{arcs\_status} - n_{\text{special\_arcs}} \times n_{\text{selected\_arcs}}
\end{align*}

This code is used in section 1.10.

Packing the nodes is very similar to packing the arcs, only there is no need to update any arrays or renumber anything, but just pack them (renumbering of the nodes was taken care of above). Here are the corresponding pieces of code:

\langle \text{Allocate Node Buffers} \, 1.19 \rangle \equiv

\begin{align*}
\text{ALLOCATE} (\text{coords\_buffer}, n_{\text{dim}}, -n_{\text{special\_nodes}} : n_{\text{selected\_nodes}}, \text{stat} = \text{error\_status}) \\
\text{CALL} \text{RecordAllocation} (n_{\text{elements}} = n_{\text{dim}} \times (n_{\text{special\_nodes}} + n_{\text{selected\_nodes}} + 1), \text{mold} = 1.0, \\
\text{caller} = \text{"CreateLatticeNetwork"}, \text{alloc\_status} = \text{error\_status}) \\
\text{ALLOCATE} (\text{status\_buffer}, -n_{\text{special\_nodes}} : n_{\text{selected\_nodes}}, \text{stat} = \text{error\_status}) \\
\text{CALL} \text{RecordAllocation} (n_{\text{elements}} = n_{\text{special\_nodes}} + n_{\text{selected\_nodes}} + 1, \text{mold} = 1_{\text{byte}}, \\
\text{caller} = \text{"CreateLatticeNetwork"}, \text{alloc\_status} = \text{error\_status}) \\
\text{coords\_buffer}_{1:n_{\text{dim}}, 0} = 0.0 \quad \text{// Null element} \\
\text{status\_buffer}_{0} = \text{empty\_node} \quad \text{// Dummy node}
\end{align*}

This code is used in section 1.10.

\langle \text{Reallocate Node Arrays Using Buffers} \, 1.20 \rangle \equiv

\begin{align*}
\text{CALL} \text{RecordAllocation} (n_{\text{elements}} = -\text{SIZE} (\text{nodes\_coords}, i_{\text{wp}}), \text{mold} = 1.0) \\
\text{DEALLOCATE} (\text{nodes\_coords}) \quad \text{// Start the re-allocation process} \\
\text{ALLOCATE} (\text{nodes\_coords}, n_{\text{dim}}, -n_{\text{special\_nodes}} : n_{\text{selected\_nodes}}, \text{stat} = \text{error\_status}) \\
\text{CALL} \text{RecordAllocation} (n_{\text{elements}} = n_{\text{dim}} \times (n_{\text{special\_nodes}} + n_{\text{selected\_nodes}} + 1), \text{mold} = 1.0, \\
\text{caller} = \text{"CreateLatticeNetwork"}, \text{alloc\_status} = \text{error\_status}) \\
\text{nodes\_coords} = \text{coords\_buffer} \quad \text{// Copy buffer} \\
\text{CALL} \text{RecordAllocation} (n_{\text{elements}} = -\text{SIZE} (\text{coords\_buffer}, i_{\text{wp}}), \text{mold} = 1.0) \\
\text{DEALLOCATE} (\text{coords\_buffer}) \\
\text{CALL} \text{RecordAllocation} (n_{\text{elements}} = -\text{SIZE} (\text{nodes\_status}, i_{\text{wp}}), \text{mold} = 1_{\text{byte}}) \\
\text{DEALLOCATE} (\text{nodes\_status}) \quad \text{// Start the re-allocation process} \\
\text{ALLOCATE} (\text{nodes\_status}, -n_{\text{special\_nodes}} : n_{\text{selected\_nodes}}, \text{stat} = \text{error\_status}) \\
\text{CALL} \text{RecordAllocation} (n_{\text{elements}} = n_{\text{special\_nodes}} + n_{\text{selected\_nodes}} + 1, \text{mold} = 1_{\text{byte}}, \\
\text{caller} = \text{"CreateLatticeNetwork"}, \text{alloc\_status} = \text{error\_status}) \\
\text{nodes\_status} = \text{status\_buffer} \quad \text{// Copy buffer} \\
\text{CALL} \text{RecordAllocation} (n_{\text{elements}} = -\text{SIZE} (\text{status\_buffer}, i_{\text{wp}}), \text{mold} = 1_{\text{byte}}) \\
\text{DEALLOCATE} (\text{status\_buffer})
\end{align*}

This code is used in section 1.10.
\( \langle \text{PackNodeArraysIntoBuffers 1.21} \rangle \equiv \)
\[\text{coords}._\text{buffer}[:n_\text{dim}, -1] = \text{nodes}._\text{coords}[:n_\text{dim}, -1] \quad // \text{Just a copy for special nodes} \]
\[\text{status}._\text{buffer}[:-1] = \text{nodes}._\text{status}[:-1] \quad // \text{No need to PACK for special nodes} \]
\[\text{node}_\text{counter} = 0 \]
\text{PackNodes: do \ node = 1, n\_nodes}
\[
\begin{align*}
\text{if (nodes\_mask\_node) then} & \quad // \text{Pack this node} \\
\text{node}_\text{counter} &= \text{node}_\text{counter} + 1 \\
\text{coords}._\text{buffer}[\text{node}_\text{counter}] &= \text{nodes}._\text{coords}[\text{node}_\text{counter}]
\end{align*}
\]
\text{end if}
\text{end do PackNodes}

This code is used in section 1.10.

\( \langle \text{CopyNodeArraysIntoBuffers 1.22} \rangle \equiv \)
\[\text{coords}._\text{buffer} = \text{nodes}._\text{coords}[:n_\text{dim}, -n_\text{special\_nodes}; n_\text{selected\_nodes}] \]
\[\text{status}._\text{buffer} = \text{nodes}._\text{status}[:n_\text{special\_nodes}; n_\text{selected\_nodes}] \]

This code is used in section 1.10.

### 1.3 Destroying the Lattice

The routine \textit{DestroyNetworkLattice} should be called to deallocate the arrays that were allocated in \textit{CreateNetworkLattice}:

\( \langle \text{DestroyLatticeNetwork 1.24} \rangle \equiv \)
\[
\text{subroutine DestroyLatticeNetwork()}
\begin{align*}
\text{implicit none} \\
\text{call-record allocation (n\_elements = -\text{size}(\text{nodes}._\text{coords}, i\_wp), mold = 1.0)} \\
\text{deallocate (nodes}._\text{coords}) \quad // \text{Allocate enough space} \\
\text{call-record allocation (n\_elements = -\text{size}(\text{nodes}._\text{status}, i\_wp), mold = 1_i\_byte)} \\
\text{deallocate (nodes}._\text{status}) \quad // \text{Allocate enough space} \\
\text{call-record allocation (n\_elements = -\text{size}(\text{heads\_tails}, i\_wp), mold = 1_i\_byte)} \\
\text{deallocate (heads\_tails)} \\
\text{call-record allocation (n\_elements = -\text{size}(\text{arcs}._\text{status}, i\_wp), mold = 1_i\_byte)} \\
\text{deallocate (arcs}._\text{status})
\end{align*}
\]

This code is used in section 1.1.
 Formatting rules for HPF/F90 files

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

@m SIZE(array, _kind, ...)
@ifelse (#O, 0, INT(SIZE(array), KIND=_kind), INT(SIZE(array, #), KIND=_kind))
@m MAXLOC(array, _kind, ...)
@ifelse (#O, 0, INT(MAXLOC(array), KIND=_kind), INT(MAXLOC(array, #), KIND=_kind))
@m MINLOC(array, _kind, ...)
@ifelse (#O, 0, INT(MINLOC(array), KIND=_kind), INT(MINLOC(array, #), KIND=_kind))
@m _UBOUND(array, _kind, ...)
@ifelse (#O, 0, INT(_UBOUND(array, DIM=1), KIND=_kind),
             INT(_UBOUND(array, #), KIND=_kind))
@m _LBOUND(array, _kind, ...)
@ifelse (#O, 0, INT(_LBOUND(array, DIM=1), KIND=_kind),
             INT(_LBOUND(array, #), KIND=_kind))
@m _GENERIC_INTERFACE(generic_name, ...)
   INTERFACE generic_name
   MODULE PROCEDURE #.
   END INTERFACE generic_name
@m _DECLARE _WORD(...)
   INTEGER ::= #.
@m _DECLARE _WP(...)
   INTEGER (KIND = i_wp) ::= #.
@m _DECLARE _RP(...)
   REAL (KIND = r_wp) ::= #.
@m _DECLARE _SP(...)
   REAL (KIND = r_sp) ::= #.
@m _DECLARE _DP(...)
   REAL (KIND = r_dp) ::= #.
@m _FULLEXTENT(rank) :: DO (DIM, 2, rank) { , }
@m _VARSEQUENCE(variable, start, end)
   variable# start DO (DIM, $EVAL(start + 1), end) { , variable@&DIM }
@m _NESTED LOOP START (variable, array, rank, kind)
   DO (DIM, rank, 1, -1) { DO variable@&DIM = _LBOUND(array, kind, DIM = DIM),
   _UBOUND(array, kind, DIM = DIM) }
@m _NESTED LOOP END (rank) DO (DIM, 1, rank) { END DO }
@m _DUMMY(...)
@m _DISPLAY ARRAY(message, array)
   IF (SIZE(array) <= 20) THEN
     WRITE(message, print_unit, ("A") message
     WRITE(message, print_unit, ("(205.2)") array
   END IF