Using the LSNNO library

Aleksandar Donev, Corey Mussollf
June 2001

1 Module LSNNO_Interface

This documentation is not finished!

This module contains the basic routines for interfacing with the LSNNO network optimization library. The routine CreateLSNNO allocates the arrays needed by this library, and DestroyLSNNO deallocates them. The main routine CallLSNNO copies the arrays from Network_Data_Structures into arrays of the proper form for LSNNO (in particular the nodes must be renumbered to start from 1, which is done here by transferring all special nodes and arcs to the end of the list), and then calls the routine LSNNO to solve the network-optimization problem, returning the solution into the array arcs_flows. DestroyLSNNO can thus be called immediately after calling CallLSNNO, since the solution will be copied into the flow array.

Please note that LSNNO does not return estimates for the lagrange multipliers (i.e. the node potentials node_potentials). These can be calculated from the flows using spanning trees and the routines in the module Graph_Algorithms, and this is done in the function CalculateLSNNOPotentials. This function is documented in its own section.

The namelist LSNNOOptions is used to hold some of the user input parameters. It is read as usual with the initialization routine InitializeLSNNO. The important public variables in this namelist are:

\texttt{initial\_flow} tells how to compute the initial flow. For now the only supported choices are 'I' for a likely infeasible fixed value of all the flows fixed at 1 (LSNNO will generate a feasible guess--choosing 0 here would be a bad choice because most gradients vanish at 0 and we would get a false convergence stop) and 'U' for user-provided, meaning that the present value of arcs\_flows will be used for the guess.

\texttt{MAXIT, WHAT, FREQ, PREC and NIPST} are explained in the LSNNO manual. The default values will usually be enough.

\texttt{EPSF} is also from LSNNO's list of variables. I single it out here because it is very important--it gives the 2-norm of the reduced gradient used for convergence testing.

\texttt{ELST\_low\_cost, ELST\_high\_cost and ELST\_regular\_cost} determine the way the Hessians and gradients are calculated and updated in LSNNO. The parameter \texttt{ELST} is documented with all its values in the LSNNO manual. Here I decided to allow different choices for the three different kinds of arcs, since low and high cost arcs may be best handled with update, rather than analytic formulae. \texttt{ELST = 1} is preferred for regular-cost arcs!
Unfortunately LSNO requires some fixed-name external routines, such as \textit{RHS}, \textit{XUPPER} and \textit{XLOWER} to be provided. These are placed outside of the module here. When I get time I may rewrite LSNO to correct for this by making these dummy routine \texttt{EXTERNAL} arguments.

"\texttt{WEAVE.f90}" 1.0.1 \equiv

\begin{verbatim}
MODULE LSNO_Interface 
USE Precision; USE Error_Handling; USE System_Monitors; USE Network_Data_Structures; USE Network_Data_Types; IMPLICIT NONE 
PRIVATE 
PUBLIC::InitializeLSNO, CreateLSNO, DestroyLSNO, CallLSNO, CalculateLSNNOptions 
/* See the LSNO user manual for an explanation of these arrays--they have the same names as in the manual. I made all of these arrays \texttt{PRIVATE}. The ones that are public are part of the namedist \texttt{LSNNOptions} (so that the user can change them easily) */ \texttt{INTEGER, ALLOCATABLE, DIMENSION (:)} :: \texttt{TO}, \texttt{FR} \texttt{INTEGER} :: \texttt{NA}, \texttt{NN}, \texttt{NEL}, \texttt{LFUVAL}, \texttt{LIWK}, \texttt{LWK}, \texttt{IPDEVc}, \texttt{LFC}, \texttt{INFO}, \texttt{IFLAG}, \texttt{INFORM} \texttt{INTEGER, ALLOCATABLE, DIMENSION (:)} :: \texttt{ELVAR}, \texttt{ELPTr}, \texttt{GPtr}, \texttt{HPTr}, \texttt{IWK}, \texttt{ELSt}, \texttt{FCALC}, \texttt{XSt} \texttt{INTEGER} :: \texttt{IFC(3)}, \texttt{IT(3)} \texttt{REAL} (\texttt{KIND = r_{dp}}) :: \texttt{FX} \texttt{REAL} (\texttt{KIND = r_{dp}}), \texttt{ALLOCATABLE, DIMENSION (:)} :: \texttt{FUVAL, WK, X} \texttt{CHARACTER, PUBLIC, SAVE :: initial\_flow = 'I' // 'I' or 'U'} \texttt{INTEGER, PUBLIC, SAVE :: MAXIT = 1000, WHAT = 2, FREQ = 1} \texttt{ // Maximal number of major iterations} \texttt{REAL (KIND = r_{dp})}, \texttt{PUBLIC, SAVE :: EPSF = 1.0 \cdot 10^{-3} r_{dp}} \texttt{ // Precision for reduced gradient} \texttt{LOGICAL, PUBLIC, SAVE :: PREC = T, NIPST = T} \texttt{ // Algorithm choices} \texttt{INTEGER, PUBLIC, SAVE :: ELSt\_low\_cost = 1, ELSt\_high\_cost = 1, ELSt\_regular\_cost = 1} \texttt{ // Hessian-gradient calculation and updating for different arcs} \texttt{NAMELIST/LSNNOptions/ initial\_flow, MAXIT, EPSF, ELSt\_low\_cost, ELSt\_high\_cost, ELSt\_regular\_cost, PREC, NIPST, WHAT, FREQ} \texttt{INTERFACE} \texttt{ // Interface to the FORTRAN 77 LSNO} \texttt{SUBROUTINE LSNO(NA, NN, NEL, ELVAR, ELPTR, ELSt, FR, TO, X, XSt, FX, FUVAL, LFUVAL, GPtr, HPTr, INFORM, FCALC, LFC, IFC, IT, MAXIT, EPSF, PREC, NIPST, IWK, LIWK, WK, LWK, IPDEVc, WHAT, FREQ, INFO, IFLAG)} \texttt{USE Precision} \texttt{IMPLICIT NONE} \texttt{INTEGER, INTENT (IN)} :: \texttt{NA}, \texttt{NN}, \texttt{NEL}, \texttt{LFUVAL}, \texttt{MAXIT}, \texttt{LIWK}, \texttt{LWK}, \texttt{IPDEVc}, \texttt{WHAT}, \texttt{FREQ} \texttt{INTEGER, INTENT (OUT)} :: \texttt{LFC, INFO, IFLAG} \texttt{INTEGER, INTENT (INOUT)} :: \texttt{INFORM} \texttt{INTEGER, DIMENSION (NA)}, \texttt{INTENT (IN)} :: \texttt{FR, TO} \texttt{INTEGER, DIMENSION (NEL)}, \texttt{INTENT (IN)} :: \texttt{ELVAR} \texttt{INTEGER, DIMENSION (NEL + 1)}, \texttt{INTENT (IN)} :: \texttt{ELPTr}
\end{verbatim}
INTEGER, DIMENSION (NEL + 1), INTENT (OUT) :: GPTR, HPTR
INTEGER, DIMENSION (3) :: IFC, IT
INTEGER, DIMENSION (IWK) :: IWK
INTEGER, DIMENSION (NEL), INTENT (INOUT) :: ELST, FCALC
INTEGER, DIMENSION (NA), INTENT (INOUT) :: XST
REAL (KIND = r_dp), INTENT (IN) :: EPSF
REAL (KIND = r_dp), INTENT (OUT) :: FX
REAL (KIND = r_dp), DIMENSION (LFUVAL), INTENT (IN) :: FUVAL
REAL (KIND = r_dp), DIMENSION (LWK), INTENT (OUT) :: WK
REAL (KIND = r_dp), DIMENSION (NA), INTENT (INOUT) :: X
LOGICAL, INTENT (IN) :: PREC, NIPST
END SUBROUTINE
END INTERFACE

CONTAINS

< InitializeLSNNO 1.1.1>
< CreateLSNNO 1.2.1>
< CallLSNNO 1.3.1>
< DestroyLSNNO 1.2.2>
< CalculateLSNOPOtentials 1.5.1>

END MODULE LSNNO_Interface

< ExternalRoutines 1.4.1>

1.1 User Options namelist LSNNOOptions

This routine reads the options in the options file related to LSNNO:

< InitializeLSNNO 1.1.1> ≜

SUBROUTINE InitializeLSNNO()

REWIND (UNIT = program_options_unit)  // Rewind the options file
READ (UNIT = program_options_unit, NML = LSNNOOptions, IOSTAT = error_status)
IF (error_status /= 0) CALL CriticalError (message = "NAMELIST LSNNOOptions was not read successfully" || " from file " || TRIM(options_file), caller = "InitializeLSNNO")
WRITE (UNIT = message_log_unit, NML = LSNNOOptions)
END SUBROUTINE InitializeLSNNO

This code is used in section 1.0.1.

1.2 Allocating and Deallocation the LSNNO Arrays
The following two routines will allocate all the arrays needed by LSNNO and also deallocate them:

\[
\text{(CreateLSNNO 1.2.1) } \equiv
\]

**SUBROUTINE CreateLSNNO()**

*IMPLICIT NONE*

**INTEGER** :: arc, special_arc, node, special_node, head, tail, alloc_status

\[
NA = n_{arcs} + n_{special\_arcs}
\]

\[
NN = n_{nodes} + n_{special\_nodes}
\]

\[
NEL = NA
\]

\[
LFUVAL = 3 * NA
\]

**ALLOCATE** (to \(NA\), \(stat = alloc\_status\))

**CALL** RecordAllocation\( (n_{elements} = NA, \ mold = 1, \ caller = "CreateLSNNO", \ alloc\_status = alloc\_status)\)

**ALLOCATE** (FR\( (NA)\), \(stat = alloc\_status\))

**CALL** RecordAllocation\( (n_{elements} = NA, \ mold = 1, \ caller = "CreateLSNNO", \ alloc\_status = alloc\_status)\)

**ALLOCATE** (\(X(NA)\), \(stat = alloc\_status\)) /\ The network flow

**CALL** RecordAllocation\( (n_{elements} = NA, \ mold = 1.0\_ip, \ caller = "CreateLSNNO", \ alloc\_status = alloc\_status)\)

**ALLOCATE** (ELVAR\( (NEL)\), \(stat = alloc\_status\))

**CALL** RecordAllocation\( (n_{elements} = NEL, \ mold = 1, \ caller = "CreateLSNNO", \ alloc\_status = alloc\_status)\)

**ALLOCATE** (ELPTR\( (NEL + 1)\), \(stat = alloc\_status\))

**CALL** RecordAllocation\( (n_{elements} = NEL + 1, \ mold = 1, \ caller = "CreateLSNNO", \ alloc\_status = alloc\_status)\)

**ALLOCATE** (ELST\( (NEL)\), \(stat = alloc\_status\))

**CALL** RecordAllocation\( (n_{elements} = NEL, \ mold = 1, \ caller = "CreateLSNNO", \ alloc\_status = alloc\_status)\)

**ALLOCATE** (XST\( (NA)\), \(stat = alloc\_status\))

**CALL** RecordAllocation\( (n_{elements} = NA, \ mold = 1, \ caller = "CreateLSNNO", \ alloc\_status = alloc\_status)\)

**ALLOCATE** (GPTR\( (NEL + 1)\), \(stat = alloc\_status\))

**CALL** RecordAllocation\( (n_{elements} = NEL + 1, \ mold = 1, \ caller = "CreateLSNNO", \ alloc\_status = alloc\_status)\)

**ALLOCATE** (HPTR\( (NEL + 1)\), \(stat = alloc\_status\))

**CALL** RecordAllocation\( (n_{elements} = NEL + 1, \ mold = 1, \ caller = "CreateLSNNO", \ alloc\_status = alloc\_status)\)

**ALLOCATE** (FCALC\( (NEL)\), \(stat = alloc\_status\))

**CALL** RecordAllocation\( (n_{elements} = NEL, \ mold = 1, \ caller = "CreateLSNNO", \ alloc\_status = alloc\_status)\)

**ALLOCATE** (FUVAL\( (LFUVAL)\), \(stat = alloc\_status\))

**CALL** RecordAllocation\( (n_{elements} = LFUVAL, \ mold = 1.0\_ip, \ caller = "CreateLSNNO", \ alloc\_status = alloc\_status)\)

\[
FR_{1:n\_arcs} = \text{heads\_tails}_{1:1:n\_arcs} \quad \text{// Heads array}
\]

\[
TO_{1:n\_arcs} = \text{heads\_tails}_{2:1:n\_arcs} \quad \text{// Tails array}
\]

\[
\text{DO } \text{s\_special\_arc} = 1, n_{special\_arcs} \quad \text{// For special nodes we must apply renumbering}
\]

4
head = heads, tails, special

tail = heads, tails, special

IF (head < 0) THEN    // Special node
    FR_{n, arcs + special} = nodes + ABS(head)
ELSE
    FR_{n, arcs + special} = head
END IF

IF (tail < 0) THEN    // Special node
    TO_{n, arcs + special} = nodes + ABS(tail)
ELSE
    TO_{n, arcs + special} = tail
END IF

/* Auxiliary pointer arrays for LSNNO: */
DO arc = 1, NA     //  These are trivial because of the strict separability of f(x)
    ELVAR(arc) = arc
    ELPTR(arc) = arc
END DO

ELPTR(NA + 1) = NA + 1     // Extra element

END SUBROUTINE CreateLSNNO

This code is used in section 1.0.1.

(\textbf{DestroyLSNNO 1.2.2}) \equiv

\begin{verbatim}
SUBROUTINE DestroyLSNNO
IMPLICIT NONE
INTEGER :: alloc_status

DEALLOCATE (TO, STAT = alloc_status)
CALL RecordAllocation(n_elements = -NA, mold = 1, caller = "DestroyLSNNO",
  alloc_status = alloc_status)
DEALLOCATE (FR, STAT = alloc_status)
CALL RecordAllocation(n_elements = -NA, mold = 1, caller = "DestroyLSNNO",
  alloc_status = alloc_status)
DEALLOCATE (X, STAT = alloc_status)
CALL RecordAllocation(n_elements = -NA, mold = 1, caller = "DestroyLSNNO",
  alloc_status = alloc_status)
DEALLOCATE (ELVAR, STAT = alloc_status)
CALL RecordAllocation(n_elements = -(NEL), mold = 1, caller = "DestroyLSNNO",
  alloc_status = alloc_status)
DEALLOCATE (ELPTR, STAT = alloc_status)
CALL RecordAllocation(n_elements = -(NEL + 1), mold = 1, caller = "DestroyLSNNO",
  alloc_status = alloc_status)
DEALLOCATE (ELST, STAT = alloc_status)
CALL RecordAllocation(n_elements = -NEL, mold = 1, caller = "DestroyLSNNO",
  alloc_status = alloc_status)
DEALLOCATE (XST, STAT = alloc_status)
CALL RecordAllocation(n_elements = -NA, mold = 1, caller = "DestroyLSNNO",
  alloc_status = alloc_status)
\end{verbatim}

5
DEALLOCATE (GPTR, STAT = alloc_status)
CALL RecordAllocation(n_elements = -(NEL + 1), mold = 1, caller = "DestroyLSNNO",
 alloc_status = alloc_status)
DEALLOCATE (HPTR, STAT = alloc_status)
CALL RecordAllocation(n_elements = -(NEL + 1), mold = 1, caller = "DestroyLSNNO",
 alloc_status = alloc_status)
DEALLOCATE (PCALC, STAT = alloc_status)
CALL RecordAllocation(n_elements = -NEL, mold = 1, caller = "DestroyLSNNO",
 alloc_status = alloc_status)
DEALLOCATE (FUVAL, STAT = alloc_status)
CALL RecordAllocation(n_elements = -LFUVAL, mold = 1.0_rdp, caller = "DestroyLSNNO",
 alloc_status = alloc_status)
IF (ALLOCATED(WK)) THEN
DEALLOCATE (WK, STAT = alloc_status)
CALL RecordAllocation(n_elements = -LWK, mold = 1.0_rdp, caller = "DestroyLSNNO",
 alloc_status = alloc_status)
END IF
IF (ALLOCATED(IWK)) THEN
DEALLOCATE (IWK, STAT = alloc_status)
CALL RecordAllocation(n_elements = -LIWK, mold = 1, caller = "DestroyLSNNO",
 alloc_status = alloc_status)
END IF
END SUBROUTINE DestroyLSNNO

This code is used in section 1.0.1.

1.3 Calling LSSNNO

The routine CallLSNNO will actually call LSNNO to solve the network optimization problem. Some of the arrays need to be transformed to the form expected by LSNNO prior to the call. In particular, all special nodes and arcs are transferred to the end of the arc and node arrays, so that the nodal elements (\((-1) : (n_{special\ nodes} + 1)\)) is copied into the range \((n_{nodes} + 1) : (n_{nodes} + n_{special\ nodes})\), and the same goes for arcs as well.

The argument Element Costs is the routine that calculates the values and derivatives of the arc cost functions. It follows the same interface, given below in the macro ElementalCostsInterface, is explained elsewhere in this library (see for example the module Power Cost Functions and its routine PowerElementalCosts), so that the same elemental costs function can be used with LSNNO as well. After LSNNO returns with success the solution is copied into the vector arcs flows. This solution can not be verified here. Use the routine CalculateLSNNO Potentials for that. Also notice that some of the routines, such as RHS, are declared outside the module here, as LSNNO requires.

\[\text{CallLSNNO 1.3.1} \equiv\]

\textbf{SUBROUTINE} CallLSNNO (ElementalCosts)
\textbf{USE} NetworkDataTypes

6
IMPLICIT NONE

_ELEMENTALCOSTSINTERFACE

_TYPE(Network_SC_Cost) :: dummy_cost_function
INTEGER :: alloc_status
INTEGER :: element, node, special_arc, special_node, head, tail
REAL(KIND = r_wp) :: dummy1 // Dummy argument for ElementalCosts
INTEGER(KIND = i_wp) :: arc_index, arc

IPDEV = message_log_unit // Log the output from LSNNO
LWK = 0 // Find out the needed workspace size dynamically
ALLOCATE (WK (LWK), stat = alloc_status)
LIWK = 0
ALLOCATE (IWK (LIWK), stat = alloc_status)

/* Hessian-update choice: */

ELST = ELST_regular_cost // Hessian-updating formula
DO specialArc = 1, n_specialArcs // Hessian-updating for special arcs
  SELECT CASE (specialArcs_status - specialArc)
    CASE (low_cost_arc)
      ELST_vec = ELST_low_cost
    CASE (high_cost_arc)
      ELST_vec = ELST_high_cost
    CASE DEFAULT
      ELST_vec = ELST_regular_cost
  END SELECT
END DO

XST = -1 // All arc flows are unbounded!
/* Set up the initial guess for the flow vector \tilde{x}: */
IF (initial_guess \equiv 'I' | initial_guess \equiv 'I') THEN
  // Infeasible initial guess-let LSNNO use its simplex code
  X = 1.0_wp
ELSE // Use the current value of arcs_flows
  X = arcs_flows
END IF
/* Now actually start the reverse-communication DO loop that calls LSNNO */
INFORM = 0

ReverseLoop: DO
  CALL LSNNO(NA, NN, NEL, ELVAR, ELPTR, ELST, FR, TO, X, XST, FX, FUVAL,
             LFUVAL, GPTR, HPTR, INFORM, FCALC, LFC, IFC, IT, MAXIT, EPSF, PREC,
             NIPST, IWK, LIWK, WK, LWK, IPDEV, WHAT, FREQ, INFO, IFLAG)

  IF (IFLAG == 1) THEN
    WRITE (message_log_unit, *) "Requested size of WK in LSNNO:", INFO
    DEALLOCATE (WK)
    IF (LWK > 0) CALL RecordAllocation(n_elements = -LWK, mold = 1.0_wp)
    LWK = INFO * 10 // This is a HACK—using LWK = INFO causes crashes sometimes
    ALLOCATE (WK (LWK), stat = alloc_status)
    CALL RecordAllocation(n_elements = LWK, mold = 1.0_wp, caller = "CallLSNNO",
                           alloc_status = alloc_status)
  END IF
  CYCLE ReverseLoop
ELSE IF (IFLAG == 2) THEN
WRITE(message_log_unit, *) "Requested size of IWK in LSNN0:", INFO
DEALLOCATE(IWK)
IF(LIWK > 0) CALL RecordAllocation(n_elements = -LIWK, mold = 1)
LIWK = INFO * 10  // This is a HACK—using LWK = INFO causes crashes sometimes
ALLOCATE(IWK(LIWK), STAT = alloc_status)
CALL RecordAllocation(n_elements = LIWK, mold = 1, caller = "CallLSNN0",
 alloc_status = alloc_status)
CYCLE ReverseLoop
ELSE IF( IFLAG ≥ 3 ) THEN
WRITE(error_print_unit, *) "LSNN0’s IFLAG=", IFLAG
WRITE(error_log_unit, *) "LSNN0’s IFLAG=", IFLAG
CALL NonCriticalError("LSNN0 aborted with IFLAG=3", caller = "CallLSNN0")
EXIT ReverseLoop
END IF
IF(INFORM > 0) THEN
SELECT CASE(INFORM)
CASE (1)  // Evaluate the cost functions
CALL ElementalCosts(cost_function = dummy_cost_function, arguments_status = "FDDC",
 arcs_flows = X1:n_arces, arcs_costs = FUVAL1:n_arces, arcs_indices = (/ 1, n_arces /),
 tolerance = EPSILON(1.0_wp))
CALL ElementalCosts(cost_function = dummy_cost_function, arguments_status = "FDDC",
 arcs_flows = X1:n_arces, arcs_costs = FUVAL(n_arces+n_special_arces):(n_arces+1):(-1),
 arcs_indices = (/ -n_special_arces, -1 /), tolerance = EPSILON(1.0_wp))
CASE (2)  // Evaluate the gradients—potentials
CALL ElementalCosts(cost_function = dummy_cost_function, arguments_status = "FDDC",
 arcs_flows = X1:n_arces, arcs_voltages = FUVAL NA+1:NA+n_arces, arcs_indices = (/ 1, n_arces /),
 tolerance = EPSILON(1.0_wp))
CALL ElementalCosts(cost_function = dummy_cost_function, arguments_status = "FDDC",
 arcs_flows = X1:n_arces, arcs_voltages = FUVAL NA+(n_arces+n_special_arces):NA+(n_arces+1):(-1),
 arcs_indices = (/ -n_special_arces, -1 /), tolerance = EPSILON(1.0_wp))
CASE (3)  // Evaluate the Hessians—resistances
CALL ElementalCosts(cost_function = dummy_cost_function, arguments_status = "FDDC",
 arcs_flows = X1:n_arces, arcs_resistances = FUVAL2*NA+1:2*NA+n_arces, arcs_indices = (/ 1, n_arces /),
 tolerance = EPSILON(1.0_wp))
CALL ElementalCosts(cost_function = dummy_cost_function, arguments_status = "FDDC",
 arcs_flows = X1:n_arces, arcs_resistances = FUVAL2*NA+(n_arces+n_special_arces):2*NA+(n_arces+1):(-1),
 arcs_indices = (/ -n_special_arces, -1 /), tolerance = EPSILON(1.0_wp))
CASE (4)  // Evaluate both potentials and resistances
CALL ElementalCosts( cost_function = dummy_cost_function, arguments_status = "FDDC",
 arcs_flows = X1:n_arces,
 arcs_voltages = FUVAL NA+1:NA+n_arces; arcs_resistances = FUVAL2*NA+1:2*NA+n_arces ,
 & arcs_indices = (/ 1, n_arces /), tolerance = EPSILON(1.0_wp) )
CALL ElementalCosts( cost_function = dummy_cost_function, arguments_status = "FDDC",
 arcs_flows = X1:n_arces; arcs_voltages = FUVAL NA+(n_arces+n_special_arces):NA+(n_arces+1):(-1),
 arcs_resistances = FUVAL2*NA+(n_arces+n_special_arces):2*NA+(n_arces+1):(-1) ,
 & arcs_indices = (/ -n_special_arces, -1 /), tolerance = EPSILON(1.0_wp) )
CASE DEFAULT
CALL CriticalError("INFORM>4 in LSNN0", caller = "CallLSNN0")
ENDSELECT
ELSE
WRITE(message_print_unit, *) "LSNN0 returned with success!!!"
 arcs_flows(1: n_arcs) = X1:n_arcs
 arcs_flows(-1):(-n_special_arcs) = Xn_arcs+1:(n_arcs+n_special_arcs)
 arcs_flows(0) = 0.0_dp          // Dummy flow
EXIT ReverseLoop
END IF
END DO ReverseLoop
CALL RecordAllocation(n_elements = SIZE(WK), mold = 1.0_dp, caller = "CallLSNN0")
DEALLOCATE(WK, STAT = alloc_status)
CALL RecordAllocation(n_elements = SIZE(IWK), mold = 1, caller = "CallLSNN0")
DEALLOCATE(IWK, STAT = alloc_status)
END SUBROUTINE CallLSNN0

This code is used in section 1.0.1.

1.4 External Routines

LSNN0 requires some fixed-name external routines, which are given here. These include the supply-demand routine \texttt{RHS}, the lower and upper bounds on the flows (not present in this case) in \texttt{XLOWER} and \texttt{XUPPER}, and a dummy routine \texttt{RANGE}.

\begin{verbatim}
(ExternalRoutines 1.4.1) ≡
FUNCTION RHS(node) RESULT(supply_demand)  // Supply-demands for node
  USE Precision
  USE Network_Data_Structures
  IMPLICIT NONE
  INTEGER, INTENT(IN) :: node
  REAL(KIND=r_dp) :: supply_demand

  IF(node > n_nodes) THEN
    supply_demand = supplies_demands(n_nodes-node)
  ELSE
    supply_demand = supplies_demands_node
  END IF
  WRITE(*, *) "Called RHS:", node, supply_demand
END FUNCTION RHS

FUNCTION XLOWER(arc) RESULT(lb)  // Lower flow bound
  USE Precision
  IMPLICIT NONE

  lb = ...  // Lower bound for flow through arc
END FUNCTION XLOWER
\end{verbatim}
\begin{verbatim}
INTEGER, INTENT (IN) :: arc
REAL (KIND = r_dp) :: lb

lb = -1.0_r_dp / EPSILON(1.0_r_dp)  // A very large negative number
END FUNCTION LOWER

FUNCTION UPPER(ARC) RESULT(ub)  // Upper bound
USE Precision
IMPLICIT NONE
INTEGER, INTENT (IN) :: arc
REAL (KIND = r_dp) :: ub

ub = 1.0_r_dp / EPSILON(1.0_r_dp)  // A very large negative number
END FUNCTION UPPER

SUBROUTINE RANGE (IEL, ELDIM, INTDIM, W1, W2, MODE)  // Dummy routine
USE Precision
USE Error_Handling
IMPLICIT NONE
INTEGER, INTENT (IN) :: IEL, ELDIM, MODE
INTEGER, INTENT (OUT) :: INTDIM
REAL (KIND = r_dp), DIMENSION (*) :: INTDIM (INOUT) :: W1, W2

IF (MODE == 0) THEN
  INTDIM = ELDIM
ELSE
  CallCriticalError("RANGE called by LSNNO with non-zero MODE", caller = "RANGE")
END IF
END SUBROUTINE RANGE
\end{verbatim}

This code is used in section 1.0.1.

1.5 Computing the Node Potentials and Verifying Optimality

The routine CalculateLSNNO_Potentials will compute the potentials on the nodes \( \lambda \) using the spanning tree algorithms in the module Graph_Algorithms, and store these in the array nodes_potentials. Please note that is uses the spanning-tree structures set up in Network_Spanning_Trees. It accepts the same subroutine Elemental_Costs that was passed to CallLSNNO, which it uses to calculate the potential drops on the (basic) arcs of a spanning tree (chosen here to be the tree with the smallest possible voltages) via \( t_{i,j} = f_{i,j} (x_{i,j}) \). From these it then calculates the potential of all nodes by calling PropagateNodesPotentials, via \( \lambda = B^{-T} t \).

The solution returned by LSNNO is optimal if and only if the potential-flow characteristics are satisfied on all arcs (including non-tree ones), and the flow is feasible. The flow that LSNNO returns is feasible by construction (to within precision), so this routine tries to verify the first condition. If the optional argument potential_mismatch is present, this routine will compare \( t \) with the potential drops obtained from the predicted potentials \( \lambda \) via \( \tilde{t} = A^T \lambda \). The percentage error mismatches between \( t \) and \( \tilde{t} \) in the 2- and infinity-norm are returned in potential_mismatch one after the other. Basically, both of these numbers have to much smaller than 1 to tolerate the solution produced.

Further feasibility validation of the the solution is obtained by computing the excess-flow at the nodes via \( \tilde{b} = Az \). The percentage mismatches between the requested supply-demand \( b \) and \( \tilde{b} \) are returned in the
optional \textit{flow\_mismatch}, if requested (i.e. if \texttt{PRESENT}). These numbers should also be small if the solution is to be trusted, and since LSNNO uses a feasible method these will usually be at the level of the numerical tolerance even if the solution is not optimal (but I still included this check for extra security). Note that the routines in \texttt{Network\_Matrix\_Operations} are used to calculate the products with $A$ and $A^T$.

\begin{verbatim}
\{CalculateLSNNOPotentials 1.5.1\} \equiv

\textbf{SUBROUTINE} CalculateLSNNOPotentials (ElementalCosts, potential\_mismatch, flow\_mismatch)
\textbf{USE} Network\_Matrix\_Operations
\textbf{USE} Graph\_Algorithms
\textbf{IMPLICIT} NONE

 ElementalCostsInterface     // Elemental costs functions
REAL (KIND = r_wp), DIMENSION (2), INTENT (OUT), OPTIONAL :: potential\_mismatch,
flow\_mismatch     // 2- and Infinity-Norm of the mismatch of the predicted potentials and
flows. These basically measure the mismatch between the predicted solution and an optimal
one. All norms should be small at a true optimum.

\textbf{TYPE} (Network\_SC\_Cost) :: dummy\_cost\_function     // Dummy variable used in ElementalCost
REAL (KIND = r_wp), DIMENSION (:), ALLOCATABLE :: temp\_buffer
INTEGER (KIND = i_wp) :: nnode, arc
INTEGER :: alloc\_status
REAL (KIND = r_wp) :: dummy1

\textbf{CALL} ElementalCosts(cost\_function = dummy\_cost\_function, arcs\_flows = arcs\_flows,
arcs\_voltages = arcs\_voltages, arcs\_resistances = dummy, arcs\_costs = dummy,
arcs\_indices = (\! n\_special\_arcs, n\_arcs /), arguments\_status = "FCID",
tolerance = EPSILON(1.0,wp))
// $t_{i,j} = f(i,j)(x(i,j))$
\textbf{CALL} CreateSpanningTree(tree\_type = min\_cost\_tree, arcs\_weights = ABS(arcs\_voltages),
weights\_distribution = "Random")
// Choose the small voltages in the tree to avoid high-cost arcs and numerical problems


\textbf{n}\_nodes\_potentials_{i:1} = 0.0,wp     // Ground the root of the tree
\textbf{CALL} PropagateNodesPotentials (arc\_offset = n\_special\_arcs, node\_offset = n\_special\_nodes,
heads\_tails = heads\_tails, orientations = tree\_nodes\_orientations, parents = tree\_nodes\_parents,
level\_ordering = tree\_nodes\_ordering, arcs\_voltages = arcs\_voltages,

\textbf{n}\_nodes\_potentials = nodes\_potentials)
// This is basically $\lambda = B^{-2}t$
\textbf{IF} (\texttt{PRESENT(potential\_mismatch)}) \textbf{THEN}
\textbf{ALLOCATE} (temp\_buffer (-n\_special\_arcs : n\_arcs), stat = alloc\_status)     // $t$
\textbf{CALL} RecordAllocation(n\_elements = n\_special\_arcs + n\_arcs + 1, mold = 1.0,wp,
caller = "CalculateLSNNOPotentials", alloc\_status = alloc\_status)
\textbf{CALL} Arcs\_Voltages (heads\_tails = heads\_tails, node\_offset = n\_special\_nodes,
nodes\_potentials = nodes\_potentials, arcs\_voltages = temp\_buffer)
// $t = A^T\lambda$
potential\_mismatch = PercentMismatch (arcs\_voltages, temp\_buffer)     // Normalized error
\textbf{CALL} RecordAllocation (n\_elements = -\_SIZE(temp\_buffer, i_wp), mold = 1.0,wp)
\textbf{DEALLOCATE} (temp\_buffer)
\textbf{END IF}
\end{verbatim}
IF (PRESENT(flow_mismatch)) THEN
  ALLOCATE (temp_buffer (n_special_nodes : n_nodes), stat = alloc_status)
  CALL RecordAllocation (n_elements = n_special_nodes + n_nodes + 1, mold = 1.0_rwp,
     caller = "CalculateLSNNOPotentials", alloc_status = alloc_status)

  CALL NodesExcessFlows (heads_tails = heads_tails, node_offset = n_special_nodes,
     arcs_flows = arcs_flows, excess_flows = temp_buffer) // b = Ax
  flow_mismatch = PercentMismatch (supplies_demands, temp_buffer) // Percent errors
  CALL RecordAllocation (n_elements = -_size (temp_buffer, i_wp), mold = 1.0_rwp)
  DEALLOCATE (temp_buffer)
END IF

CONTAINS
  (PercentMismatch 1.5.2)
END SUBROUTINE CalculateLSNNOPotentials

This code is used in section 1.0.1.

This function calculates a measure of the discrepancy between two arrays A and B by computing a kind of a normalized error in the 2 and infinity (max) norm (in that order):

\[ \langle \text{PercentMismatch} \rangle \equiv \]

FUNCTION PercentMismatch (A, B) RESULT (norms)
  USE Precision
  REAL (KIND = r_wp), DIMENSION (:), INTENT (IN) :: A, B
  REAL (KIND = r_wp), DIMENSION (2) :: norms
  INTEGER (KIND = i_wp) :: element, n_elements
  REAL (KIND = i_wp) :: magnitude
  n_elements = _size (A, i_wp)
  norms = 0.0_rwp // Initialize
  magnitude = 0.0_rwp // Average magnitude of elements of A and B
  DO element = _bound (A, i_wp), _bound (B, i_wp)
    norms1 = norms1 + (A element - B element)**2 // 2-Norm
    norms2 = max (norms2, abs (A element - B element)) // Max (infinity)-Norm
    magnitude = magnitude + 0.5_rwp * (abs (A element) + abs (B element))
  END DO
  magnitude = magnitude / real (n_elements, r_wp)
  norms1 = sqrt (norms1 / real (n_elements, r_wp)) / magnitude // I decided to normalize her by
dividing with a measure of the magnitude of the elements of A and B
  norms2 = norms2 / magnitude // This is normalizes as a percent error too
END FUNCTION PercentMismatch

This code is used in section 1.5.1.
case_type type

* TYPE type
* null > null()
* private private

size(array, kind, ...) ! if else (0, 0, int(size(array), kind=_kind), int(size(array, #), kind=_kind))

maxloc(array, kind, ...) ! if else (0, 0, int(maxloc(array), kind=_kind), int(maxloc(array, #), kind=_kind))

minloc(array, kind, ...) ! if else (0, 0, int(minloc(array), kind=_kind), int(minloc(array, #), kind=_kind))

lbound(array, kind, ...) ! if else (0, 0, int(lbound(array, dim=1), kind=_kind), int(lbound(array, #), kind=_kind))

ubound(array, kind, ...) ! if else (0, 0, int(ubound(array, dim=1), kind=_kind), int(ubound(array, #), kind=_kind))

generic interface(generic name, ...)

end interface generic name

declare i_word(...)

integer :: #.

declare i_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) :: #.

full_extent(rank) ! do (dim, 2, rank) { , , }

var_sequence(variable, start, end)

var##start ! do (dim, start+1, end) { , var@dim }!

nested_loop_start(variable, array, rank, kind)

! do (dim, rank, 1, -1) { do var@dim = lbound(array, kind, dim=dim), 

ubound(array, kind, dim=dim) }

nested_loop_end(rank) ! do (dim, 1, rank) { end do }

dummy(...)

display_array(message, array)

if (size(array) <= 20) then

write(message, print_unit, "(a)" message

write(message, print_unit, "(2005.2)" array

end if