Linear Algebra with $A$ and $A^T$

Aleksandar Donev

May 2001

1 Module Network_Matrix_Operations

This module contains routines for multiplying a flow vector $x$ with the node-arc incidence matrix of a graph $b = Ax$, where $b$ is the excess flow at each node. This is the routine NodesExcessFlows. Also included is a routine ArcsVoltages that will multiply a potential vector $\lambda$ with $A^T$ to calculate the voltages (potential drops) across the network arcs $t$, $t = A^T\lambda$. There is a routine for multiplying a vector with a Laplacian matrix of the form $AMA^T$ MultiplyADA (the name is kind of not-in-tune, but it is makes it clear that it can be viewed as a purely mathematical multiplication with a Laplacian matrix), and a routine for extracting only the diagonal of the Laplacian, Diag(AMA$^T$). All routines accept an optional mask vector arcs_mask which selects a subset of the arcs to be used in the operation.

This module is not fully finished yet. This module contains both array and loop statements. The array syntax is easier to read, and some compilers may like it better. Still, I think the loop constructs are preferred with most of today's Fortran processors. For the serial versions given here, it shouldn't really matter much. However, these routines are all easily parallelizable using halos, and are therefore separated from the rest of the intrinsically serial routines in Graph_Algorithms.

"WEAVE.f90" 1.0.0.1 ≡

MODULE Network_Matrix_Operations
  USE Precision
  USE Error_Handling
  USE System_Monitors
  IMPLICIT NONE
  PUBLIC :: NodesExcessFlows, ArcsVoltages, MultiplyADA, DiagonalADA
PRIVATE
CONTAINS
  (NodesExcessFlows 1.1.0.1)
  (ArcsVoltages 1.2.0.1)
  (MultiplyADA 1.3)
  (DiagonalADA 1.4)
END MODULE Network_Matrix_Operations
1.1 Muliplication with $A$

This routine calculates the product $b = Ax$ for some flow vector $x$. Here the array statement fails to convey the fact that the the loops are to be merged:

$$\langle \text{NodesExcessFlows 1.1.0.1} \rangle \equiv$$

```
SUBROUTINE NodesExcessFlows (heads_tails, arcs_mask, node_offset, arcs_flows, excess_flows, initialize)  // $b = Ax$
INTEGER (KIND = i_wp), INTENT (IN) :: node_offset  // Node numbering lower bound
INTEGER (KIND = i_wp), DIMENSION (:,:), INTENT (IN) :: heads_tails  // Heads-tails array for $G$
LOGICAL (KIND = L_wp), DIMENSION (:,:), INTENT (IN), OPTIONAL :: arcs_mask
REAL (KIND = r_wp), DIMENSION (:,:), INTENT (IN) :: arcs_flows  // $x$
REAL (KIND = r_wp), DIMENSION (-node_offset :), INTENT (OUT) :: excess_flows  // $b$
LOGICAL, INTENT (IN), OPTIONAL :: initialize

INTEGER :: head, tail, arc, node
LOGICAL :: first_time

first_time = .T.  // A virgin run by default
IF (PRESENT(initialize))
  first_time = initialize
IF (first_time) excess_flows = 0.0_r_wp
IF (PRESENT(arcs_mask)) THEN
  (MaskedLoopSyntaxA 1.1.1.2)
ELSE
  (LoopSyntaxA 1.1.1.1)
END IF

END SUBROUTINE NodesExcessFlows
```

This code is used in section 1.0.0.1.

1.1.1 Loops

Here are both the array and loop based forms:

$$\langle \text{LoopSyntaxA 1.1.1.1} \rangle \equiv$$

```
DO arc = _BOUND(heads_tails, i_wp, DIM = 2), _UBOUND(heads_tails, i_wp, DIM = 2)
  head = heads_tails1, arc
tail = heads_tails2, arc
  excess_flows_head = excess_flows_head + arcs_flows_arc
  excess_flows_tail = excess_flows_tail - arcs_flows_arc
```

2
This code is used in section 1.1.0.1.

\[ \langle \text{MaskedLoopSyntax}_A \ 1.1.1.2 \rangle \equiv \]
\begin{verbatim}
DO arc = _JBOUND(heads_tails, i_up, DIM = 2), _JBOUND(heads_tails, i_up, DIM = 2)
   IF (arcs_mask[arc]) THEN
      head = heads_tails[1, arc]
      tail = heads_tails[2, arc]
      excess_flows_head = excess_flows_head + arcs_flows[arc]
      excess_flows_tail = excess_flows_tail - arcs_flows[arc]
   END IF
END DO
\end{verbatim}

This code is used in section 1.1.0.1.

\[ \langle \text{ArraySyntax}_A \ 1.1.1.3 \rangle \equiv \]
\begin{verbatim}
excess_flows(heads_tails[1, :]) = excess_flows(heads_tails[1, :]) + arcs_flows
excess_flows(heads_tails[2, :]) = excess_flows(heads_tails[2, :]) - arcs_flows
\end{verbatim}

### 1.2 Multiplication with $A^T$

This routine will multiply a potential vector $\lambda$ with $A^T$ to give a tension vector $t$. In this case I like the array syntax better:

\[ \langle \text{ArcsVoltages} \ 1.2.0.1 \rangle \equiv \]
\begin{verbatim}
SUBROUTINE ArcsVoltages(heads_tails, node_offset, arcs_mask, nodes_potentials, arcs_voltages)
   // t = $A^T\lambda$
   INTEGER (KIND = i_up), INTENT (IN) :: node_offset    // Node numbering lower bound
   INTEGER (KIND = i_up), DIMENSION (:), INTENT (IN) :: heads_tails    // Heads-tails array for G
   LOGICAL (KIND = l_up), DIMENSION (:), INTENT (IN), OPTIONAL :: arcs_mask
   REAL (KIND = r_up), DIMENSION (-node_offset :), INTENT (IN) :: nodes_potentials    // $\lambda$
   REAL (KIND = r_up), DIMENSION (:), INTENT (OUT) :: arcs_voltages    // t
   INTEGER :: head, tail, arc, node
   IF (PRESENT(arcs_mask)) THEN
      \( \langle \text{MaskedArraySyntax}_A \ 1.2.1.3 \rangle \)
   ELSE
      \( \langle \text{ArraySyntax}_A \ 1.2.1.2 \rangle \)
   END IF
\end{verbatim}
END SUBROUTINE ArcsVoltages

This code is used in section 1.0.0.1.

1.2.1 Loops

Here are both the array and loop based forms:

\[ \langle \text{LoopSyntax}_A \ 1.2.1.1 \rangle \equiv \]
\[
\text{DO } \text{arc} = \_\text{BOUND}(\text{heads\_tails}_1, i\_wp, \text{DIM} = 2), \_\text{BOUND}(\text{heads\_tails}_2, i\_wp, \text{DIM} = 2) \\
\hspace{1cm} \text{head} = \text{heads\_tails}_{1, \text{arc}} \\
\hspace{1cm} \text{tail} = \text{heads\_tails}_{2, \text{arc}} \\
\hspace{1cm} \text{arcs\_voltages}_{\text{arc}} = \text{nodes\_potentials}_{\text{head}} - \text{nodes\_potentials}_{\text{tail}} \\
\text{END DO} \]

\[ \langle \text{ArraySyntax}_A \ 1.2.1.2 \rangle \equiv \]
\[
\text{arcs\_voltages} = \text{nodes\_potentials}(\text{heads\_tails}_1, :) - \text{nodes\_potentials}(\text{heads\_tails}_2, :) \]

This code is used in section 1.2.0.1.

\[ \langle \text{MaskedArraySyntax}_A \ 1.2.1.3 \rangle \equiv \]
\[
\text{WHERE}(\text{arcs\_mask}) \\
\hspace{1cm} \text{arcs\_voltages} = \text{nodes\_potentials}(\text{heads\_tails}_1, :) - \text{nodes\_potentials}(\text{heads\_tails}_2, :) \]

This code is used in section 1.2.0.1.

1.3 Multiplication with $ADA^T$

This routine performs the matrix-vector multiplication at the core of the Dual Newton Method for the Newton system $y = (ADA^T)x$, where $y$ is the left-hand side vector $\text{nodes\_flows}$ and $x$ is $\text{nodes\_potentials}$. The matrix $D$ is assumed to be diagonal and is given via the arc-length vector $D$ in $\text{arcs\_conductances}$.

\[ \langle \text{Multiply}_A^\text{ADA} \ 1.3 \rangle \equiv \]
\[
\text{SUBROUTINE Multiply}_A^\text{ADA}(\text{heads\_tails}, \text{node\_offset}, \text{arcs\_conductances}, \text{arcs\_mask}, \text{nodes\_flows}, \text{nodes\_potentials}, \text{initialize}) \hspace{1cm} // t = A^T \lambda \\
\hspace{1cm} \text{INTEGER} (\text{KIND} = i\_wp), \text{INTENT} (\text{IN}) : \text{node\_offset} \hspace{1cm} // \text{Node numbering lower bound} \]
INTEGER (KIND = i_wp), DIMENSION (:), INTENT (IN) :: heads_tails // Heads-tails array for G
REAL (KIND = r_wp), DIMENSION (:), INTENT (IN) :: arcs_conductances // $D_a$ a diagonal matrix
LOGICAL (KIND = L_wp), DIMENSION (:), INTENT (IN), OPTIONAL :: arcs_mask
REAL (KIND = r_wp), DIMENSION (-node_offset :), INTENT (IN) :: nodes_potentials // $x$
REAL (KIND = r_wp), DIMENSION (-node_offset :), INTENT (OUT) :: nodes_flows // $y$
LOGICAL, INTENT (IN), OPTIONAL :: initialize

INTEGER :: head, tail, arc, node, n_arcs
REAL (KIND = r_wp) :: arc_flow // A temporary
LOGICAL :: first_time

first_time = .T. // A virgin run by default
IF (PRESENT(initialize))
  first_time = initialize
IF (first_time) nodes_flows = 0.0_r_wp
IF (PRESENT(arcs_mask)) THEN
  (MaskedLoopSyntaxADA 1.3.1.2)
ELSE
  (LoopSyntaxADA 1.3.1.1)
END IF

n_arcs = _SIZE(heads_tails, i_wp, DIM = 2)
n_reads = n_reads + 7 * n_arcs // Count the number of memory reads–expensive
n_writes = n_writes + 2 * n_arcs // Number of memory writes–most expensive
n_flops = n_flops + 4 * n_arcs // The number of flops

END SUBROUTINE Multiply_ADA

This code is used in section 1.0.0.1.

1.3.1 Loops

Here are both the array and loop based forms:

(LoopSyntaxADA 1.3.1.1) ≡

  DO arc = _BOUND(heads_tails, i_wp, DIM = 2), _UBOUND(heads_tails, i_wp, DIM = 2)
    head = heads_tails1, arc
    tail = heads_tails2, arc
    arc_flow = arcs_conductances arc * (nodes_potentials head − nodes_potentials tail)
    nodes_flows head = nodes_flows head + arc_flow
    nodes_flows tail = nodes_flows tail − arc_flow
  END DO

This code is used in section 1.3.
\( \text{MaskedLoopSyntax}_{AD} \text{At 1.3.2} \equiv \)

\[
\text{DO } \text{arc} = \text{LB} \text{OUND}(\text{heads}\_\text{tails}, \ i\_\text{wp}, \ \text{DIM} = 2), \ \text{UB} \text{OUND}(\text{heads}\_\text{tails}, \ i\_\text{wp}, \ \text{DIM} = 2) \\
\text{IF (arc\_mask}_{\text{arc})} \text{THEN} \\
\quad \text{head} = \text{heads}\_\text{tails}_{1, \ \text{arc}} \\
\quad \text{tail} = \text{heads}\_\text{tails}_{2, \ \text{arc}} \\
\quad \text{arc\_flow} = \text{arc\_conductances}_{\text{arc}} \times (\text{nodes\_potentials}_{\text{head}} - \text{nodes\_potentials}_{\text{tail}}) \\
\quad \text{nodes\_flows}_{\text{head}} = \text{nodes\_flows}_{\text{head}} + \text{arc\_flow} \\
\quad \text{nodes\_flows}_{\text{tail}} = \text{nodes\_flows}_{\text{tail}} - \text{arc\_flow} \\
\text{END IF} \\
\text{END DO}
\]

This code is used in section 1.3.

### 1.4 The Diagonal of \( \text{ADA}^T \)

In preconditioning the dual Newton system of equations it is often needed to know the diagonal of the above Hessian matrix, called \( \text{nodes\_conductances} \), which can easily be calculated as follows:

\( \text{Diagonal}_{\text{ADA} \text{At 1.4}} \equiv \)

\[
\text{SUBROUTINE } \text{Diagonal}_{\text{ADA} \text{At}}(\text{heads}\_\text{tails}, \ \text{node}\_\text{offset}, \ \text{arc}\_\text{conductances}, \ \text{nodes}\_\text{conductances}, \ \text{nodes}\_\text{resistances}, \ \text{arc}\_\text{mask}, \ \text{initialize}) \\
\text{INTEGER (KIND = i\_wp), IN} : \text{node}\_\text{offset} \quad / / \text{Node numbering lower bound} \\
\text{INTEGER (KIND = i\_wp), DIMENSION (:), IN} : \text{heads}\_\text{tails} \quad / / \text{Heads-tails array for } G \\
\text{REAL (KIND = r\_wp), DIMENSION (:), IN} : \text{arc}\_\text{conductances} \quad / / \text{D—a diagonal matrix} \\
\text{REAL (KIND = r\_wp), DIMENSION (—node\_offset :), OUT, OPTIONAL } : \text{nodes}\_\text{conductances}, \ \text{nodes}\_\text{resistances} \quad / / \text{at least one must be present} \\
\text{LOGICAL (KIND = l\_wp), DIMENSION (:), IN, OPTIONAL } : \text{arc}\_\text{mask} \\
\text{LOGICAL, IN, OPTIONAL } : \text{initialize} \\
\text{INTEGER : head, tail, arc, node, n\_arscs} \\
\text{LOGICAL : first\_time} \\
\text{first\_time} = T \quad / / \text{A virgin run by default} \\
\text{IF (PRESENT(initialize))} \\
\quad \text{first\_time} = \text{initialize} \\
\text{IF (first\_time) nodes\_conductances = 0.0_{\_wp} } \\
\text{IF (PRESENT(arc\_mask))} \text{THEN} \\
\quad \text{Syntax}_{\text{D}_{\text{ADA} \text{At}}(_{\text{MaskedLoop})}} \\
\text{ELSE} \\
\quad \text{Syntax}_{\text{D}_{\text{ADA} \text{At}}(_{\text{Loop})}} \\
\text{END IF} \\
\text{END SUBROUTINE } \text{Diagonal}_{\text{ADA} \text{At}}
\]

This code is used in section 1.0.0.1.

### 1.4.1 Loops
Here are both the array and loop based forms:

```
"WEAVE.f90" 1.4.1.1 ≡
@m _MaskedLoopSyntax D ADA_t(nodes_conductances)
  do arc = J.BOUND(heads_tails, i_wp, DIM = 2), J.BOUND(heads_tails, i_wp, DIM = 2)
    if (arcs_mask arc) then
      head = heads_tails 1, arc
      tail = heads_tails 2, arc
      nodes_conductances head = nodes_conductances head + arcs_conductances arc
      nodes_conductances tail = nodes_conductances tail + arcs_conductances arc
    end if
  end do

"WEAVE.f90" 1.4.1.2 ≡
@m _LoopSyntax D ADA_t(nodes_conductances)
  do arc = J.BOUND(heads_tails, i_wp, DIM = 2), J.BOUND(heads_tails, i_wp, DIM = 2)
    head = heads_tails 1, arc
    tail = heads_tails 2, arc
    nodes_conductances head = nodes_conductances head + arcs_conductances arc
    nodes_conductances tail = nodes_conductances tail + arcs_conductances arc
  end do
@m _MaskedArraySyntax D ADA_t(nodes_conductances)
  where (arcs_mask)
    nodes_conductances (heads_tails 1, :) = nodes_conductances (heads_tails 1, :) + arcs_conductances
    nodes_conductances (heads_tails 2, :) = nodes_conductances (heads_tails 2, :) + arcs_conductances
  end where
@m _ArraySyntax D ADA_t(nodes_conductances)
  nodes_conductances (heads_tails 1, :) = nodes_conductances (heads_tails 1, :) + arcs_conductances arc
  nodes_conductances (heads_tails 2, :) = nodes_conductances (heads_tails 2, :) + arcs_conductances arc
@m _Syntax D ADA_t(_Form)
  if (PRESENT(nodes_conductances)) then
    _Form##Syntax_D ADA_t(nodes_conductances)
  else if (PRESENT(nodes_resistances)) then
    _Form##Syntax_D ADA_t(nodes_resistances)
    nodes_resistances = 1.0, wp / (nodes_resistances + EPSILON(1.0, wp))
  end if
```

7
@m CASE_TYPE TYPE
@m _TYPE TYPE
@m _NULL > NULL( )
@m _PRIVATE PRIVATE
@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 _BOUND(array, _kind,...) $IFELSE (#0, 0, INT(_BOUND(array, DIM=1), KIND=_kind),
   INT(_BOUND(array, #.), KIND=_kind))
@m _BOUND(array, _kind,...) $IFELSE (#0, 0, INT(_BOUND(array, DIM=1), KIND=_kind),
   INT(_BOUND(array, #.), KIND=_kind))
@m _GenericInterface(generic_name,...)
   INTERFACE generic_name MODULE PROCEDURE #.
@m _Declare i_word(...)
   INTEGER :: #.
@m _Declare i_wp(...)
   INTEGER (KIND = i_wp) :: #.
@m _Declare r_wp(...)
   REAL (KIND = r_wp) :: #.
@m _Declare r_sp(...)
   REAL (KIND = r_sp) :: #.
@m _Declare r_dp(...)
   REAL (KIND = r_dp) :: #.
@m _FullExtent(_rank) : DO (DIM, 2, _rank) | { : }
@m _VarSequence(_variable, _start, _end)
   _variable##_start$DO (DIM, _start + 1, _end) | { _variable@&DIM }
@m _NestedLoopStart(_variable, _array, _rank, _kind)
   $DO (_DIM, _rank, 1, -1) | { _DIM =_BOUND(_array, _kind, DIM = _DIM),
   _BOUND(_array, _kind, DIM = _DIM) }$DO (_DIM, 1, _rank) | { END DO }
@m Dummy(...)
@m _DisplayArray(message, array)
   IF (SIZE(array) <= 20) THEN
      WRITE(message, print_unit, "(A)" message
      WRITE(message, print_unit, "(2005.2)" array
   END IF

1