General remarks on the algorithm underlying fadvance

The first thing to understand is that the unknown for which the equations are written is not the voltage value of each compartment, but rather the increase of the voltage value from the previous to the current time-step. So if V is the voltage, using the notation for the "standard" unknown $V_i^n = V(x_i, t_n)$, the actual unknown is $\Delta V_i^n = V_i^n - V_i^{n-1}$. We can compare the "standard" Backward Euler for the NEURON equations (see my other document on NEURON's algorithm)

$$c_j \frac{V_j^n - V_j^{n-1}}{\Delta t} = I_j + \frac{1}{2\pi a_j \Delta x}\left(\frac{\pi a_{j+1/2}^2}{R_{j+1/2}}\frac{V_{j+1}^n - V_j^n}{\Delta x} + \frac{\pi a_{j-1/2}^2}{R_{j-1/2}}\frac{V_{j-1}^n - V_j^n}{\Delta x}\right) \tag{1}$$

with the actual equations solved by fadvance (see Chapter 7, page 168 of the NEURON BOOK)

$$\begin{aligned} c_j \frac{\Delta V_j^n}{\Delta t} - \frac{1}{2\pi a_j \Delta x}\left(\frac{\pi a_{j+1/2}^2}{R_{j+1/2}}\frac{\Delta V_{j+1}^n - \Delta V_j^n}{\Delta x} + \frac{\pi a_{j-1/2}^2}{R_{j-1/2}}\frac{\Delta V_{j-1}^n - \Delta V_j^n}{\Delta x}\right) \\ = I_j + \frac{1}{2\pi a_j \Delta x}\left(\frac{\pi a_{j+1/2}^2}{R_{j+1/2}}\frac{V_{j+1}^{n-1} - V_j^{n-1}}{\Delta x} + \frac{\pi a_{j-1/2}^2}{R_{j-1/2}}\frac{V_{j-1}^{n-1} - V_j^{n-1}}{\Delta x}\right). \end{aligned} \tag{2}$$

In this equation, the unknowns are $\Delta V_j^n, \Delta V_{j+1}^n$ and ΔV_{j+1}^n. The second equation is a bit uglier, but Hines states that it can be reused for different methods (for example, for Backward Euler and Crank-Nicholson, but also CVODE ad daspk).

There are a few vectors in memory:

- `VEC_A`;
- `VEC_B`;
- `VEC_D`;
- `VEC_RHS`;
- `VEC_V`.

Each thread accesses its own part of the memory, asynchronously from the other threads, because anyway each thread accesses its own dedicated portion of the vectors, without interacting with the others. This statement should be checked for the end points!
These vectors contain:

`VEC_A` the upper triangular of the matrix representing equation (2).

`VEC_B` the lower triangular of the matrix representing equation (2).

`VEC_D` the diagonal of the matrix representing equation (2).

`VEC_RHS` the right hand side of equation (2).

`VEC_V` the voltage values at the center of the compartments.

Moreover, there are two index arrays: the array of indexes that belong to a certain thread, and the array of parent nodes. Figure 1 gives an example of the numbering process, which follows these rules

1. there is one root node per cell (neuron);
2. as we go farther from the root, the numbering increases;
3. the ID of a branching point is alway smaller than its children.

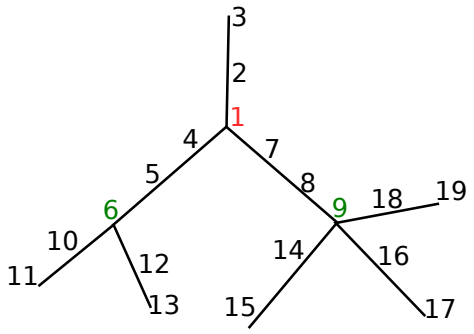

Figure 1: Example of tree structured neuron.

Supposing this whole neuron belonged to a thread, the indexes and parents would be:

index	parent index
1	-
2	1
3	2
4	1
5	4
6	5
7	4
8	7
9	8
10	6
11	10
12	6
13	12
14	9
15	14
16	9
17	16
18	9
19	18

In the case where multiple cells (neurons) are assigned to one thread, we then have multiple root nodes (one per each cell). However the numbering is global, and all the roots are assigned the smallest possible indexes, with the intent of keeping them at the beginning of the matrix.
What fadvance does is loop through all the nodes in the neuron assigned to a certain thread, and populate vectors `VEC_A,VEC_B,VEC_D,VEC_RHS`, using the values from the previous time step contained in vector `VEC_V`. An important detail is that the resistance values can be used for computing the right hand side as well, so it is possible to see them also in the expression of the right hand side! Therefore `VEC_A` plays the double role of matrix entries but also vector of physical parameters, which can be confusing at first!

First the right hand side is populated with function call `setup_tree_matrix_minimal(nth)`, which in turn calls `nrn_rhs(_nt);`.

Function `nrn_rhs()` is charged with computing the relevant values corresponding to the term

$$\frac{a_*^2}{2a_*R_*\Delta x}\frac{V_*^{n-1}-V_*^{n-1}}{\Delta x}. \tag{3}$$

In the right hand side of equation (2) this term appears at least twice: on for the couple $j, j+1$ and another for the couple $j, j-1$. However, in the code the `VEC_RHS(i)` is only modified once! This is because node i computes value (3) for the couple composed by itself and its parent node and stores it in `VEC_RHS(i)-= VEC_B(i)*dv`, then computes the value for the "inversed" couple, composed by the parent node and itself, and stores it in the parent

node's position `VEC_RHS(_nt->_v_parent_index[i])+= VEC_A(i)*dv`. This automatically takes care of the connectivity between nodes! Referencing Figure 1, node 4 computes the term `VEC_RHS(4)-=`$\frac{a_2^n}{2R_{4,2}a_4(\Delta x)^2}(V_2^{n-1} - V_4^{n-1})$, and then computes `VEC_RHS(2)+=`$\frac{a_4^n}{2R_{4,2}a_2(\Delta x)^2}(V_2^{n-1} - V_4^{n-1})$ (I did not check the signs...). This corresponds to a specific pattern: each node assigns to himself the part corresponding to `VEC_B`, and to his parent the part corresponding to `VEC_A`.

Function `nrn_lhs()`. After, the vector `VEC_D` is populated with two contributions: the axial currents from the term

$$\frac{1}{2\pi a_j \Delta x}\left(\frac{\pi a_{j+1/2}^2}{R_{j+1/2}}\frac{\Delta V_{j+1}^n - \Delta V_j^n}{\Delta x} + \frac{\pi a_{j-1/2}^2}{R_{j-1/2}}\frac{\Delta V_{j-1}^n - \Delta V_j^n}{\Delta x}\right) \tag{4}$$

with the call

```
VEC_D(i)-= VEC_B(i);
VEC_D(_nt->_v_parent_index[i])-= VEC_A(i);
```

and the capacitance current from the term

$$c_j \frac{\Delta V_j^n}{\Delta t} \tag{5}$$

with the call

```
nrn_cap_jacob(_nt, _nt->tml->ml).
```

It should be noted that the population of the axial current term follows the same pattern as before (each node assigns to himself the part corresponding to `VEC_B`, and to his parent the part corresponding to `VEC_A`) while the capacitance current is assigned directly.
Finally, the right hand side is populated with the values coming from the mechanism (this is also complicated, but not necessary to understand the linear algebra).

Function `nrn_solve_minimal()`. Another important concept is the fact that the lower and upper parts of the matrix, i.e. `VEC_B,VEC_A` are never recomputed after the first initialization, because the resistivity parameters do not depend upon time. This means that the only computations that we perform are to populate the right hand side.
Finally, the last concept to grasp is that the solution of the equations is stored in the right hand side vector. This means that fadvance calls `nrn_solve_minimal(nth)`, which is composed of the two subroutines that I talked about in my presentation (triangularization and backward substitution), but they only modify `VEC_D` and `VEC_RHS`! At the end of the backward substitution subroutine, `VEC_RHS` contains the solution to (2).

Function `update()`. Now, the solution contained in `VEC_RHS` is not the actual voltage values, but the increments defined by $\Delta V_i^n = V_i^n - V_i^{n-1}$. This means that a call to an update function must be made by `update(nth)` in order to recover the voltage values, by the simple equation $V_i^n = \Delta V_i^n + V_i^{n-1}$.

```
rright = 0.;
for (j = 0; j < sec->nnode-1; j++) {
        nd = sec->pnode[j];
        ...
        diam = p->param[0];
        ...
        NODEAREA(nd) = PI*diam*dx;
        UPDATE_VEC_AREA(nd);
    rleft = 1e-2*ra*(dx/2)/(PI*diam*diam/4.); /*left half segment
        Megohms*/
    NODERINV(nd) = 1./(rleft + rright); /*uS*/
    rright = rleft;
}
/* last segment has 0 length. area is 1e2 in dimensionless units */
NODEAREA(sec->pnode[j]) = 1.e2;
UPDATE_VEC_AREA(sec->pnode[j]);
NODERINV(sec->pnode[j]) = 1./rright;
sec->recalc_area_ = 0;
diam_changed = 1;
```

Figure 2: Piece of code computing the area and inverse of resistance for each node.

Population of vectors VEC_A and VEC_B

The functions `nrn_area_ri` and `connection_coef` populate the vectors A and B, for the nodes inside a section. The first function populates some support vectors, while the second one actually computes the matrix entries. The support vectors are: `NODEAREA`, `NODERINV` and `NODEA`,`NODEB`. The last two vectors contain the same values that will be inserted in `VEC_A`,`VEC_B`, whereas `NODEAREA[i]` contains the value of the area of node i and `NODERINV` contains the inverse of the resistance associated to node i. In function `nrn_area_ri`, the line `ra = nrn_ra(sec);` retrieves the R_a value corresponding to the section of interest. The subsequent line `dx = section_length(sec)/((double)(sec->nnode - 1));` computes the formula $\Delta x = \frac{L}{N-1}$ where N is the number of nodes in the section, and L is the section length.
Then the code in Figure 2 computes the areas and resistances for each node. The area is computed according to the formula

$$Area_i = \pi d_i \Delta x,$$

where d_i is the diameter at node i. On the other hand, the resistance associated to node i is computed by the sum of two contributions: half of R_{right} from the parent node, and half of R_{left} from this node, with formula

$$R_{\text{left}} = 0.01 \frac{R_a}{2} \Delta x \times \frac{4}{\pi d^2}.$$

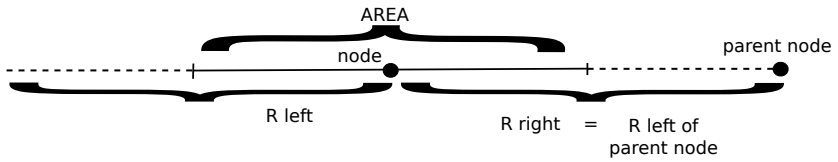

Figure 3: Position of nodes, area and resistance values.

```
nd = sec->pnode[0];
area = NODEAREA(sec->parentnode);
/* dparam[4] is rall_branch */
ClassicalNODEA(nd) = -1.e2*sec->prop->dparam[4].val * NODERINV(nd) /
    area;
for (j = 1; j < sec->nnode; j++) {
       nd = sec->pnode[j];
       area = NODEAREA(sec->pnode[j - 1]);
       ClassicalNODEA(nd) = -1.e2 * NODERINV(nd) / area;
}
/* now the effect of parent on node equation. */
ForAllSections(sec)
       for (j=0; j < sec->nnode; j++) {
              nd = sec->pnode[j];
              ClassicalNODEB(nd) = -1.e2 * NODERINV(nd) / NODEAREA(nd);
       }
```

Figure 4: Piece of code computing the matrix entries.

There is a distinction between the diameter values, which can be different for different nodes inside the same section, and the resistance values, which seem to be specific to a section. Figure 3 provides a visualization of the process of the loop in the code: for each node, the left part of the resistance is computed, while the right part is obtained from the parent computation. The auxiliary vector `NODERINV` is then filled with the values `NODERINV[i]`$= \frac{1}{R_{\text{right}}+R_{\text{left}}}$.
The function `connection_coef` is actually in charge of computing the matrix values, with the code in Figure 4.

The lower band of the tridiagonal matrix is populated with the lines

```
area = NODEAREA(sec->pnode[j - 1]);
ClassicalNODEA(nd)= -1.e2 * NODERINV(nd)/ area
```

corresponding to formula (considering node i with parent k)

$$\texttt{ClassicalNODEA(i)} = \frac{-10^4}{\pi d_k \Delta x (R_{right} + R_{left})} = \frac{-10^4}{\pi d_k \Delta x} \frac{\pi d_i^2 d_k^2}{2 R_a \Delta x (d_i^2 + d_k^2)}$$
$$= \frac{-2 \times 10^4}{2\pi a_k \Delta x} \frac{\pi a_i^2 a_k^2}{R_a \Delta x (a_i^2 + a_k^2)}$$

The upper band of the tridiagonal matrix is populated with the line
`ClassicalNODEB(nd)= -1.e2 * NODERINV(nd)/ NODEAREA(nd);`
corresponding to formula

$$\texttt{ClassicalNODEB(i)} = \frac{-10^4}{\pi d_i \Delta x (R_{right} + R_{left})} = \frac{-10^4}{\pi d_i \Delta x} \frac{\pi d_i^2 d_k^2}{2 R_a \Delta x (d_i^2 + d_k^2)} = \frac{-2 \times 10^4}{2\pi a_i \Delta x} \frac{\pi a_i^2 a_k^2}{R_a \Delta x (a_i^2 + a_k^2)}$$

These formulas can be related to equations (4), (2) by recalling that $d_i = 2a_i$ (there is a strange factor 2!). Also I am guessing that the factor 10^4 comes from measurement units.

Relationship between vectors and matrix

The purpose of this section is to state the relationship between the data structures `VEC_A, VEC_B`, etc. and the matrix they represent.
An important property should be stated (see Chapter 7, page 168 of the NEURON BOOK) which stems from the fact that *"`Node[i].b` is the resistance between node i and its parent divided by the area of the node. `Node[i].a` is the same but divided by the area of the parent node"*. This means that if we consider a node i with parent j, and we look at vector `ClassicalNODEB(i)`, it contains the matrix entry corresponding to row i and column j, while `ClassicalNODEA(i)` contains the matrix entry corresponding to row j and column i! This is similar to the procedure described above for populating vectors `VEC_A, VEC_B`, where each node populates a part for himself and a part for its parent. If the entries of the matrix are denoted by M_{ij}, and we consider node i with parent k, we have:

$$\begin{aligned} \texttt{ClassicalNODEB(i)} &= M_{ik}, \\ \texttt{ClassicalNODEA(i)} &= M_{ki}, \end{aligned} \tag{6}$$

Since by the numbering we know that the parent index is always smaller than the node index, we have $k < i$, which indeed means that `ClassicalNODEB(i)` belongs to the lower triangular of the matrix, and `ClassicalNODEA(i)` belongs to the upper triangular.
Therefore, if we wish to go from the NEURON data structure to the matrix representation, we need the vectors:

- `VEC_A`
- `VEC_B`
- `VEC_D`
- `_v_parent_index`

and we can simply apply the code:

```
for(i=0;i<length_of_vectors;++i){
        Matrix[i][_v_parent_index[i]] = VEC_B[i];
        Matrix[_v_parent_index[i]][i] = VEC_A[i];
        Matrix[i][i]                  = VEC_D[i];
}
```

On the other hand, if we have the matrix and wish to obtain the vectors relevant to the NEURON representation, we need to apply the two part code:

```
//Part 1: obtain the parent indexes
for(i=0;i<Len-1;++i){
        for(int j=i+1;j<Len;++j){
                if(fabs(Matrix[i][j]) > 0){
                        _v_parent_index[[j] = i;
                }
        }
}
//Part 2: populate the vectors
for(i=0;i<Len;++i){
        VEC_A[i] = Matrix[_v_parent_index[i]][i];
        VEC_B[i] = Matrix[i][_v_parent_index[i]];
        VEC_D[i] = Matrix[i][i];
}
```

$$\begin{pmatrix} D_1 & A_2 & A_3 & 0 & 0 & A_6 & 0 & 0 & 0 \\ B_2 & D_2 & 0 & A_4 & 0 & 0 & 0 & 0 & 0 \\ B_3 & 0 & D_3 & 0 & A_5 & 0 & 0 & 0 & 0 \\ 0 & B_4 & 0 & D_4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & B_5 & 0 & D_5 & 0 & 0 & 0 & 0 \\ B_6 & 0 & 0 & 0 & 0 & D_6 & A_7 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & B_7 & D_7 & A_8 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & B_8 & D_8 & A_9 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & B_9 & D_9 \end{pmatrix}$$

```
/* triangularization of the matrix equations */
static void triang(NrnThread* _nt) {
        double p;
        int i, i2, i3;
        i2 = _nt->ncell;
        i3 = _nt->end;

        for (i = i3 - 1; i >= i2; --i) {
        p = VEC_A(i) / VEC_D(i);
        VEC_D(_nt->_v_parent_index[i]) -= p * VEC_B(i);
        VEC_RHS(_nt->_v_parent_index[i]) -= p * VEC_RHS(i);
        }
}
```

```
/* back substitution to finish solving the matrix equations */
static void bksub(NrnThread* _nt) {
	int i, i1, i2, i3;
	i1 = 0;
	i2 = i1 + _nt->ncell;
	i3 = _nt->end;
	for (i = i1; i < i2; ++i) {
		VEC_RHS(i) /= VEC_D(i);
	}
	for (i = i2; i < i3; ++i) {
		VEC_RHS(i) -= VEC_B(i) * VEC_RHS(_nt->_v_parent_index[i]);
		VEC_RHS(i) /= VEC_D(i);
	}
}
```

$$M_{kk} -= \frac{M_{ki}}{M_{ii}} M_{ik}$$

$$\begin{pmatrix}
D_1 - \alpha & 0 & 0 & 0 & 0 & A_6 & 0 & 0 & 0 \\
B_2 & D_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
B_3 & 0 & D_3 - \frac{A_5}{D_5} B_5 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & B_4 & 0 & D_4 - \frac{A_2}{D_2} B_2 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & B_5 & 0 & D_5 & 0 & 0 & 0 & 0 \\
B_6 & 0 & 0 & 0 & 0 & D_6 - \frac{A_7}{D_7} B_7 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & B_7 & D_7 - \frac{A_8}{D_8} B_8 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & B_8 & D_8 - \frac{A_8}{D_8} B_8 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & B_9 & D_9
\end{pmatrix}$$

$$\alpha = \frac{A_6}{D_6} B_6 - \frac{A_1}{D_1} B_1 - \frac{A_2}{D_2} B_2$$