Introduction

In contrast with the last document, this one offers a formulation which is easily implemented as a piece of code. From what I have found, there is no need to use for loops per se during the calculation of the matrix as well as the RHS and LHS vectors.

An effort has been made in order to use sparse matrices thanks to `csc_matrix` and `lil_matrix`.

LHS and RHS formulation

Finding the voltages of order [0] is quite straightforward and it hasn't changed at all in comparison to the last document. Thus, I will detail how to operate from order [1] up to $[n_{coeff}]$.

The vectors LHS and RHS have been rearranged by just moving elements up or down as convenient. First, one encounters the real parts of the voltages stacked up, then the imaginary parts and finally, the unknown reactive powers. The indices follow the natural sequence, ignoring if they fall in the PV or PQ bus classification. In other words, considering the slack bus receives the index 0, the linear system $LHS = MAT.RHS$ becomes:

$$\begin{bmatrix} U_1^{re}[c] \\ \vdots \\ U_i^{re}[c] \\ \vdots \\ U_n^{re}[c] \\ U_1^{im}[c] \\ \vdots \\ U_i^{im}[c] \\ \vdots \\ U_n^{im}[c] \\ Q_i[c-1] \\ Q_n[c-1] \end{bmatrix} = MAT \begin{bmatrix} \mathrm{Re}(RHS[c,1]) \\ \vdots \\ \mathrm{Re}(RHS[c,i]) \\ \vdots \\ \mathrm{Re}(RHS[c,n]) \\ \mathrm{Im}(RHS[c,1]) \\ \vdots \\ \mathrm{Im}(RHS[c,i]) \\ \vdots \\ \mathrm{Im}(RHS[c,n]) \\ RHS_Q[c,i] \\ RHS_Q[c,n] \end{bmatrix} \tag{1}$$

where

$c = 1, ..., n_{coeff}$: index that represents the current depth.

$U_i^{re}[c]$: real part of the c coefficient corresponding to bus i.

$U_i^{im}[c]$: imaginary part of the c coefficient corresponding to bus i.

$Q_i[c-1]$: $c-1$ coefficient of the unknown reactive power from bus i.

$RHS[c,i]$: right hand side element of i bus at depth c.

$RHS_Q[c,i]$: right hand side element for the last PV bus equation of i bus at depth c.

Note that the last elements containing the unknown reactive powers only appear for PV buses. Otherwise, they wouldn't be unknowns. So following the above formulation, buses i and n are PV buses.

Vector RHS depends on value of c while M is strictly constant. When $c = 1$, for PQ buses we get equation 2.

$$RHS[1,i] = (V_0 - 1)Y_{0i} + S_i^* X_i[0] + U_i[0]Y_{sh,i} \tag{2}$$

For PV buses we have to use equation 3.

$$RHS[1,i] = (V_0 - 1)Y_{0i} + P_i X_i[0] + U_i[0]Y_{sh,i} \tag{3}$$

And the final elements of the RHS are calculated with equation 4.

$$RHS_Q[1,i] = W_i - 1 \tag{4}$$

For orders $c > 1$, equations 2, 3 and 4 end up being equations 5, 6 and 7 respectively.

$$RHS[c,i] = (S_i^* X_i[c-1] + U_i[c-1]Y_{sh,i}) \tag{5}$$

$$RHS[c,i] = -j\sum_{k=1}^{c-1} X_i[k]Q_i[c-1-k] + P_i X_i[c-1] + U_i[c-1]Y_{sh,i} \tag{6}$$

$$RHS_Q[c,i] = -\sum_{k=1}^{c-1} U_i[k]U_i^*[c-k] \tag{7}$$

Code for LHS and RHS

The piece of code used for creating RHS and decomposing LHS for $c = 1, ..., n_{coeff}$ is:

```
# .......................CALCULATION OF TERMS [1]
valor = np.zeros(n - 1, dtype=complex)
valor[pq - 1] = (V_slack - 1) * vec_Y0[pq - 1, 0] + (vec_P[pq - 1, 0] - vec_Q[pq
     - 1, 0] * 1j) * X[0, pq - 1] + U[
    0, pq - 1] * vec_shunts[pq - 1, 0]
valor[pv - 1] = (V_slack - 1) * vec_Y0[pv - 1, 0] + (vec_P[pv - 1, 0]) * X[0, pv
     - 1] + U[0, pv - 1] * vec_shunts[
    pv - 1, 0]
RHS = np.zeros(2*(n - 1) + npv, dtype=complex)
RHS[pq - 1] = np.real(valor[pq - 1])
RHS[pv - 1] = np.real(valor[pv - 1])
RHS[n - 1 + (pq - 1)] = np.imag(valor[pq - 1])
RHS[n - 1 + (pv - 1)] = np.imag(valor[pv - 1])
RHS[2 * (n - 1):] = vec_W[pv - 1, 0] - 1
#... HERE WE WOULD CALCULATE MATRIX MAT ...#
LHS = MAT_csc(RHS)
U_re[1, :] = LHS[:(n - 1)]
U_im[1, :] = LHS[(n - 1):2 * (n - 1)]
Q[0, pv - 1] = LHS[2 * (n - 1):]
U[1, :] = U_re[1, :] + U_im[1, :] * 1j
X[1, :] = (-X[0, :] * np.conj(U[1, :])) / np.conj(U[0, :])
X_re[1, :] = np.real(X[1, :])
X_im[1, :] = np.imag(X[1, :])
# .......................CALCULATION OF TERMS [>=2]
for c in range(2, prof):  # prof = n_coeff
    valor[pq - 1] = (vec_P[pq - 1, 0] - vec_Q[pq - 1, 0] * 1j) * X[c - 1, pq -
    1] + U[c -       1, pq - 1] * vec_shunts[pq - 1, 0]
    valor[pv - 1] = conv(X, Q, c, pv - 1, 2) * (-1) * 1j + U[c - 1, pv - 1] *
                vec_shunts[pv - 1, 0] + X[c - 1, pv - 1] * vec_P[pv - 1, 0]
```

```
RHS[pq - 1] = np.real(valor[pq - 1])
RHS[pv - 1] = np.real(valor[pv - 1])
RHS[n - 1 + (pq - 1)] = np.imag(valor[pq - 1])
RHS[n - 1 + (pv - 1)] = np.imag(valor[pv - 1])
RHS[2 * (n - 1):] = -conv(U, U, c, pv - 1, 3)
LHS = MAT_csc(RHS)
U_re[c, :] = LHS[:(n - 1)]
U_im[c, :] = LHS[(n - 1):2 * (n - 1)]
Q[c - 1, pv - 1] = LHS[2 * (n - 1):]
U[c, :] = U_re[c, :] + U_im[c, :] * 1j
X[c, range(n - 1)] = -conv(U, X, c, range(n - 1), 1) / np.conj(U[0, range(n
- 1)])
X_re[c, :] = np.real(X[c, :])
X_im[c, :] = np.imag(X[c, :])
```

MAT formulation

The matrix that relates LHS and RHS will be called MAT, as in equation 1. Considering that in that case we have considered index 1 to be represent a PQ bus and indices i and n to represent PV buses, MAT takes the form of equation 8 (having ignored dots that simbolize other buses).

$$\begin{bmatrix} [\mathbf{G}] & [-\mathbf{B}] & \begin{bmatrix} 0 & 0 \\ -X_i^{im}[0] & 0 \\ 0 & -X_n^{im}[0] \end{bmatrix} \\ [\mathbf{B}] & [\mathbf{G}] & \begin{bmatrix} 0 & 0 \\ X_i^{re}[0] & 0 \\ 0 & X_n^{re}[0] \end{bmatrix} \\ \begin{bmatrix} 0 & 2U_i^{re}[0] & 0 \\ 0 & 0 & 2U_n^{re}[0] \end{bmatrix} & \begin{bmatrix} 0 & 2U_i^{im}[0] & 0 \\ 0 & 0 & 2U_n^{im}[0] \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \end{bmatrix} \tag{8}$$

where

[**G**]: conductance matrix.

[**B**]: susceptance matrix.

The matrices containing $2U[0]$ and X are not square matrices by default. They would only be square if there were no PQ buses.

In the code below these matrices are first created by considering they are square. So we suppose there are no PQ buses during the initialitzation. Then, the rows corresponding to PQ buses are removed in the matrices with $2U[0]$ and the columns which refer to PQ buses are deleted in the matrices with X. That way we end up with the desired matrices.

In terms of efficiency I guess that it is not the optimal solution.

MAT code

I have attached the code that creates the matrix MAT. For the full code you can access the file 'HEallv2' located in the repository.

```
MAT = lil_matrix((dimensions, 2 * (n - 1) + npv), dtype=complex)
MAT[:(n - 1), :(n - 1)] = G
MAT[(n - 1):2 * (n - 1), :(n - 1)] = B
MAT[:(n - 1), (n - 1):2 * (n - 1)] = -B
MAT[(n - 1):2 * (n - 1), (n - 1):2 * (n - 1)] = G

MAT_URE = np.zeros((n - 1, n - 1), dtype=complex)
np.fill_diagonal(MAT_URE, 2*U_re[0, :])
MAT[2 * (n - 1):, :(n - 1)] = np.delete(MAT_URE, list(pq - 1), axis=0)
MAT_UIM = np.zeros((n - 1, n - 1), dtype=complex)
np.fill_diagonal(MAT_UIM, 2*U_im[0, :])
MAT[2 * (n - 1):, (n - 1):2 * (n - 1)] = np.delete(MAT_UIM, list(pq - 1), axis
    =0)
MAT_XIM = np.zeros((n - 1, n - 1), dtype=complex)
np.fill_diagonal(MAT_XIM, -X_im[0, :])
```

```
MAT[:(n - 1), 2 * (n - 1):] = np.delete(MAT_XIM, list(pq - 1), axis=1)
MAT_XRE = np.zeros((n - 1, n - 1), dtype=complex)
np.fill_diagonal(MAT_XRE, X_re[0, :])
MAT[(n-1):2 * (n - 1), 2 * (n - 1):] = np.delete(MAT_XRE, list(pq - 1), axis=1)

MAT_lil = lil_matrix(MAT)
MAT_csc = factorized(csc_matrix(MAT_lil))
LHS = MAT_csc(RHS)
```