COMPLETE FORMULATION OF THE HOLOMORPHIC EMBEDDING METHOD

Josep Fanals Batllori

u1946589@campus.udg.edu

Universitat de Girona

Contents

Introduction

This document attempts to present a coherent working formulation and implementation of the Holomorphic Embedding (HE from now on) applied to electrical power systems. The PQ buses had already been solved in the first development stage of the HE (A. Trias, 2012). However, PV buses where not included there. When it is about PV buses, the work of B. Schmidt (2015) is certainly not optimal: it has been showed that the coefficients that form the MacLaurin series do not converge. Using continuation algorithms it is still possible to come up with the correct solution. Despite that, one encounters degradation problems in all solutions. Several authors, such as A. Trias (2018) or S. Rao et al. (2016) have opted for a formulation that obtains the reactive power and voltages at each step.

The approach this document explains does not calculate the reactive power at the very first step. Instead, even thouth the voltages are obtained at each and every single step, the reactive power is calculated a step later in respect to voltages. Furthermore, these are initialized at 1 pu, which according to the theory behind the method, becomes the correct starting point.

Following the structure of the document, first of all one encounters a brief explanation of the equations that define both PQ and PV buses. Finally it is all packed toghether in the section 'Full formulation'. I would like to mention that it has been tested in Python (with a code pretty badly written, I know). That code has also been included.

PQ bus' equations

Considering the first Kirchhoff's Law at a PQ bus, it is quite straightforward to obtain equation 1 for the node i.

$$\sum_{j=1} Y_{ij}U_j(s) = I_i(s) + s\frac{S_i^*}{U_i^*(s^*)} + sU_i(s)Y_{sh,i} \tag{1}$$

where

Y_{ij}: element i, j of the admittances matrix. This matrix does not contain shunt admittances, only series elements which can be connected to the slack bus.

$U_j(s)$: j bus voltage unknown.

$I_i(s)$: current injected from the slack bus to the PQ bus.

S_i^*: complex conjugate of the bus i apparent power.

$Y_{sh,i}$: negative sum of shunt admittances connected to bus i.

Note that the right-hand side terms are all dependent on s except for $I_i(s)$. That is key to initialitze al voltages at 1. It is also worth mentioning that the term $sU_i(s)Y_{sh,i}$ ought to be $-sU_i(s)Y_{sh,i}$. For reasons concerning simplicity, the negative sign will be introduced internally in $Y_{sh,i}$ and the notation $sU_i(s)Y_{sh,i}$ is still going to be used.

The slack bus voltage is also embedded in s following equation 2.

$$V_0(s) = 1 + s(V_0 - 1) \tag{2}$$

where V_0 is the known voltage at the slack bus (tipically with a phase of 0) and the index 0 is reserved for the slack bus. Therefore, PV and PQ indices will run from 1 to n.

Next, equation 3 comes from guessing that the bus i is connected directly to ground, in other words, its voltage is 0. That guess will be far from the truth but becomes necessary in order to be consistent with the definition of the admittances matrix Y, where there are the admittances Y_{i0} between the slack bus and the bus i.

$$I_i(s) = V_0(s) * Y_{0i} \tag{3}$$

The way to obtain the coefficients that conform the series will be detailed i the 'Full formulation' section.

PV bus' equations

Applying Kirchhoff's Law at a PV bus yields the equation 4, which resembles the already obtained equation 1.

$$\sum_{j=1} Y_{ij}U_j(s) = I_i(s) + s\frac{P_i - jQ_i(s)}{U_i^*(s^*)} + sU_i(s)Y_{sh,i} \tag{4}$$

The only difference of that last equation in comparison with equation 1 is that the reactive power is unknown. Thus, it has to be calculated. Recall that the property $S_i + S_i^* = 2P_i$ (presented by B. Schmidt (2015)) cannot be used due to obtaining non-convergent series.

In contrast with A. Tias (2018), by multiplying both P_i and $jQ_i(s)$ by s, the coefficient $Q_i[0]$ most likely will not equal 0. That should not be a concern since the solution obtained will still be consistent with the equations.

PV buses are also defined by the voltage magnitude. That information can be converted into equation 5.

$$U_i(s)U_i^*(s^*) = 1 + s(W_i - 1) \tag{5}$$

where $W_i = |U_i|^2$. The main motivation behind this embedding is that it is needed to initialize all voltages at 1. Note that equation 2 also satisfies that need. Consequently, there will be no current flowing throught series impedances in the first step. That approach is way different form the one presented by B. Schmidht (2015) where the author claims that a made up term to initialize the voltages is needed. Their first coefficients are in most cases distant from 1.

Apart from that, it is easily observed that equations 2 and 3 will be common for PQ and PV buses. Equation 1 is meant to be applied to PQ buses while equations 4 and 5 will be only used for PV buses.

Full formulation

This section aims to detail the steps necessary to calculate the coeficients of the series, for both voltages and also unknown reactive powers.

First step

First, it is necessary to establish $s = 0$. From equations 1 and 4, and a bit of manipulation when it comes to express the injected current from the slack bus, the equation 6 can be deduced.

$$\begin{bmatrix} Y_{11} & \cdots & Y_{1n} \\ \vdots & \ddots & \vdots \\ Y_{n1} & \cdots & Y_{nn} \end{bmatrix} \begin{bmatrix} U_1[0] \\ \vdots \\ U_n[0] \end{bmatrix} = \begin{bmatrix} Y_{01} \\ \vdots \\ Y_{0n} \end{bmatrix} \tag{6}$$

It can be shown that applying equation 6, according by the above definitions, all terms $U_i[0] = 1$. That has been forced in the code in order to have no imaginary parts (despite being extremely small) in the first terms. Despite that, there is no problem in using 6 because of the fact that practically speaking in both cases the results are the same.

In equations 1 and 4 one realizes that there may be difficulty in dealing with $U_i^*(s^*)$ in the denominator. To simplify that, let's define a new variable following equation 7.

$$X_i(s) = \frac{1}{U_i^*(s^*)} \tag{7}$$

As a result of that, the coefficients of order 0 simply become $1/U_i^*[0]$. Therefore, $X_i[0] = 1$.

So far the so-called 'First step' has been solved. That is, the first terms of all variables one could find at order 0 are known. The procedure will be detailed for order 1 ('Second step') and generalized for orders bigger than 1 ('c step').

Second step

From now on $s = 1$. In that step it is necessary to solve for all $U_i[1]$, and for all $Q_i[0]$ concerning PV buses. Things get a little bit trickier here since equations 1 and 4 to be separated into real and imaginary parts because the unknown reactive powers have to be obtained. So, for every PQ bus there will be 2 unknowns (real and imaginary part of $U_i[1]$, which will be denoted as $U_i^{re}[1]$ and $U_i^{im}[1]$ respectively) and 2 equations, and form each PV bus there will be 3 unknowns ($U_i^{re}[1]$, $U_i^{im}[1]$ and $Q_i[0]$) and 3 equations. Therefore, the system has as many unkowns as equations and is expected to have a determined solution.

The equation 8 is the one concerning PQ buses.

$$\begin{aligned}\sum_{j=1} G_{ij}U_j^{re}[1] - B_{ij}U_j^{im}[1] = RHS_{PQ}[i,1] = \mathrm{Re}((V_0-1)Y_{0i} + S_i^* X_i[0] + U_i[0]Y_{sh,i})\\ \sum_{j=1} B_{ij}U_j^{re}[1] + G_{ij}U_j^{im}[1] = RHS_{PQ}[i,2] = \mathrm{Im}((V_0-1)Y_{0i} + S_i^* X_i[0] + U_i[0]Y_{sh,i})\end{aligned} \tag{8}$$

where Re and Im are functions that grab the real and the imaginary part respectively and Y_{ij} can be split into $G_{ij} + jB_{ij}$ (do not confuse the index j with the imaginary unit $j = \sqrt{-1}$).

The equation 9 describes the expressions needed for PV buses.

$$\begin{aligned}\sum_{j=1}(G_{ij}U_j^{re}[1] - B_{ij}U_j^{im}[1]) - X_i^{im}[0]Q_i[0] = \mathrm{Re}((V_0-1)Y_{0i} + P_i X_i[0] + U_i[0]Y_{sh,i})\\ \sum_{j=1}(B_{ij}U_j^{re}[1] + G_{ij}U_j^{im}[1]) + X_i^{re}[0]Q_i[0] = \mathrm{Im}((V_0-1)Y_{0i} + P_i X_i[0] + U_i[0]Y_{sh,i})\\ 2U_i^{re}[0]U_i^{re}[1]2U_i^{im}[0]U_i^{im}[1] = W_i - 1\end{aligned} \tag{9}$$

Next, one only needs to solve the linear system expressed in equation 10. The approach followed in the code does not sort first PQ buses and then the PV ones; they are contained in the matrix sorted by their index. It could be easily modified to sort them by type.

$$M = \begin{bmatrix} G_{11} & -B_{11} & \cdots & G_{1i} & -B_{1i} & 0 & \cdots & G_{1n} & -B_{1n} \\ B_{11} & G_{11} & \cdots & B_{1i} & G_{1i} & 0 & \cdots & B_{1n} & G_{1n} \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots \\ G_{i1} & -B_{i1} & \cdots & G_{ii} & -B_{ii} & -X_i^{im}[0] & \cdots & G_{in} & -B_{in} \\ B_{i1} & G_{i1} & \cdots & B_{ii} & G_{ii} & X_i^{re}[0] & \cdots & B_{in} & G_{in} \\ 0 & 0 & \cdots & 2U_i^{re}[0] & 2U_i^{im}[0] & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots & \vdots & & \vdots & \vdots \\ G_{n1} & -B_{n1} & \cdots & G_{ni} & -B_{ni} & 0 & \cdots & G_{nn} & -B_{nn} \\ B_{n1} & G_{n1} & \cdots & B_{ni} & G_{ni} & 0 & \cdots & B_{nn} & G_{nn} \end{bmatrix} \tag{10}$$

By working with the matrix M it will only be needed to invert this once because it will remain constant for all orders. Note that following equation 10, buses 1 and n are PQ while bus i is a PV bus. The formulation can be easily extrapolated to other cases. Finally, the linear system ends up being expressed in equation 11.

$$M \begin{bmatrix} U_1^{re}[1] \\ U_1^{im}[1] \\ \vdots \\ U_i^{re}[1] \\ U_i^{im}[1] \\ Q_i[0] \\ \vdots \\ U_n^{re}[1] \\ U_n^{im}[1] \end{bmatrix} = \begin{bmatrix} \mathrm{Re}((V_0 - 1)Y_{01} + S_1^* X_1[0] + U_1[0]Y_{sh,1}) \\ \mathrm{Im}((V_0 - 1)Y_{01} + S_1^* X_1[0] + U_1[0]Y_{sh,1}) \\ \vdots \\ \mathrm{Re}((V_0 - 1)Y_{0i} + P_i X_i[0] + U_i[0]Y_{sh,i}) \\ \mathrm{Im}((V_0 - 1)Y_{0i} + P_i X_i[0] + U_i[0]Y_{sh,i}) \\ W_i - 1 \\ \vdots \\ \mathrm{Re}((V_0 - 1)Y_{0n} + S_n^* X_n[0] + U_n[0]Y_{sh,n}) \\ \mathrm{Im}((V_0 - 1)Y_{0n} + S_n^* X_n[0] + U_n[0]Y_{sh,n}) \end{bmatrix} \tag{11}$$

To end that step, all unknowns have to be updated, meaning that $U_i[1] = U_i^{re}[1] + jU_i^{im}[1]$. It will also be required to update the series $X_i(s)$ by means of the equation 12, which is based on the convolution between $U^*(s)$ and $X(s)$. That equation is valid for orders $c \geq 1$.

$$X_i[c] = \frac{-\sum_{k=1}^{c} U_i^*[k] X_i[c-k]}{U_i^*[0]} \tag{12}$$

c step

The third step is caracterized by being generalised for orders $c \geq 2$, where c arrives to the maximum depth established. That depth is arbitrary, although it would be possible to introduce some kind of stop condition. According to A. Trias (2018), under normal conditions it should range from 20 to 40.

Anyway, in order to calculate the coefficients, the matrix M remains constant for all c. The vector on the right-hand side changes slightly. Equation 13 shows the linear system for order c.

$$M \begin{bmatrix} U_1^{re}[c] \\ U_1^{im}[c] \\ \vdots \\ U_i^{re}[c] \\ U_i^{im}[c] \\ Q_i[c-1] \\ \vdots \\ U_n^{re}[c] \\ U_n^{im}[c] \end{bmatrix} = \begin{bmatrix} \mathrm{Re}(S_1^* X_1[c-1] + U_1[c-1]Y_{sh,1}) \\ \mathrm{Im}(S_1^* X_1[c-1] + U_1[c-1]Y_{sh,1}) \\ \vdots \\ \mathrm{Re}(-j\sum_{k=1}^{i-1} X_i[k]Q_i[i-1-k] + P_i X_i[c-1] + U_i[c-1]Y_{sh,i}) \\ \mathrm{Im}(-j\sum_{k=1}^{i-1} X_i[k]Q_i[i-1-k] + P_i X_i[c-1] + U_i[c-1]Y_{sh,i}) \\ -\sum_{k=1}^{i-1} U_i[k]U_i^*[i-k] \\ \vdots \\ \mathrm{Re}(S_n^* X_n[c-1] + U_n[c-1]Y_{sh,n}) \\ \mathrm{Im}(S_n^* X_n[c-1] + U_n[c-1]Y_{sh,n}) \end{bmatrix} \tag{13}$$

In each step the unknowns also have to be updated ($U_i[1] = U_i^{re}[1] + jU_i^{im}[1]$). Once that is done and c gets to the desired depth, the algorithm is concluded.

The final voltages could be calculated with Padé approximants, although there exist several other methods. The most rudimentary one consists in adding all coefficients if they do converge. Others, based on recurrence, such as Wynn's Epsilons are proved to be a suitable option in most cases.

Python code

Important: in the following code, due to the fact that the coefficients converge rapidly, the final voltages are obtained by adding all terms. The depth is established at 30. In addition to that, admit that the admittance matrix could be build with an incidence matrix, it might have been easier. The index 0 is meant to be used for only the slack bus.

Some variables and functions are defined in catalan; I will do my best to comment the code. Keep in mind that in order to work properly it needs the file 'Dades_ v1.xlsx', where the topology and the information regarding each bus is specified in different sheets. All the code is packed in the file 'HELM_ complete.py'.

```
#AUTHOR: Josep Fanals Batllori
#CONTACT: u1946589@campus.udg.edu
# --------------------------- LIBRARIES
import numpy as np
import pandas as pd
# --------------------------- END LIBRARIES
# --------------------------- INITIAL DATA: Y, SHUNTS AND Y0i
df_top = pd.read_excel('Dades_v1.xlsx', sheet_name='Topologia')  # dataframe of
    the topology
num_busos = 0 #number of buses initialized to 0
busos_coneguts = np.zeros(0, dtype=int) #vector to store the indices of the
    found buses
def trobar(element, vector): #function to check if an element is in a vector
    if element in vector:
        return True
    else:
        return False
for i in range(df_top.shape[0]):  #go through all rows
    for j in range(0, 2):  #go through 1st and 2nd column to grab the bus'
    indices
        if not trobar(df_top.iloc[i, j], busos_coneguts): #if the index is new
            num_busos += 1
            busos_coneguts = np.append(busos_coneguts, df_top.iloc[i, j])
n = num_busos
Yx = np.zeros((n, n), dtype=complex) #matrix with all series admittances, also
    slack bus
for i in range(df_top.shape[0]):  #go through all rows
    Yx[df_top.iloc[i, 0], df_top.iloc[i, 0]] = Yx[df_top.iloc[i, 0], df_top.iloc
    [i, 0]] + 1 / (
                df_top.iloc[i, 2] + df_top.iloc[i, 3] * 1j)  # diagonal element
    Yx[df_top.iloc[i, 1], df_top.iloc[i, 1]] = Yx[df_top.iloc[i, 1], df_top.iloc
    [i, 1]] + 1 / (
                df_top.iloc[i, 2] + df_top.iloc[i, 3] * 1j)  # diagonal element
    Yx[df_top.iloc[i, 0], df_top.iloc[i, 1]] = Yx[df_top.iloc[i, 0], df_top.iloc
    [i, 1]] - 1 / (
                df_top.iloc[i, 2] + df_top.iloc[i, 3] * 1j)  # off diagonal
    Yx[df_top.iloc[i, 1], df_top.iloc[i, 0]] = Yx[df_top.iloc[i, 1], df_top.iloc
    [i, 0]] - 1 / (
                df_top.iloc[i, 2] + df_top.iloc[i, 3] * 1j)  # off diagonal
Y = np.zeros((n - 1, n - 1), dtype=complex) #admittance matrix withouth slack
    bus
for i in range(n - 1):
    for j in range(n - 1):
        Y[i, j] = Yx[i + 1, j + 1] #just ignoring the first row and column
vecx_shunts = np.zeros((n, 1), dtype=complex) #vector with shunt admittances
for i in range(df_top.shape[0]):  # passar per totes les files
    vecx_shunts[df_top.iloc[i, 0], 0] = vecx_shunts[df_top.iloc[i, 0], 0] +
    df_top.iloc[i, 4] * (
        -1) * 1j  #B/2 is in column 4. The sign is changed here
    vecx_shunts[df_top.iloc[i, 1], 0] = vecx_shunts[df_top.iloc[i, 1], 0] +
    df_top.iloc[i, 4] * (
        -1) * 1j  #B/2 is in column 4. The sign is changed here
vec_shunts = np.zeros((n - 1, 1), dtype=complex) #same vector, just to adapt
for i in range(n - 1):
    vec_shunts[i, 0] = vecx_shunts[i + 1, 0]
vec_shunts = --vec_shunts  #no need to change the sign, already done
vec_Y0 = np.zeros((n - 1, 1), dtype=complex) #vector with admittances connecting
     to the slack
```

```
for i in range(df_top.shape[0]):  #go through all rows
    if df_top.iloc[i, 0] == 0:  #if slack in the first column
        vec_Y0[df_top.iloc[i, 1] - 1, 0] = 1 / (
                    df_top.iloc[i, 2] + df_top.iloc[i, 3] * 1j) #-1 so bus 1
    goes to index 0
    elif df_top.iloc[i, 1] == 0:  #if slack in the second column
        vec_Y0[df_top.iloc[i, 0] - 1, 0] = 1 / (df_top.iloc[i, 2] + df_top.iloc[
    i, 3] * 1j)
G = np.real(Y)  #real parts of Yij
B = np.imag(Y)  #imaginary parts of Yij
# --------------------------- INITIAL DATA: Y, SHUNTS AND Y0i. DONE
# --------------------------- INITIAL DATA: BUSES INFORMATION
print(num_busos)
df_bus = pd.read_excel('Dades_v1.xlsx', sheet_name='Busos')  #dataframe of the
    buses
if df_bus.shape[0] != num_busos:
    print('Error: número de busos de ''Topologia'' i de ''Busos'' no és igual')
    #check if number of buses is coherent
num_busos_PQ = 0 #initialize number of PQ buses
num_busos_PV = 0 #initialize number of PV buses
vec_busos_PQ = np.zeros([0], dtype=int) #vector to store the indices of PQ buses
vec_busos_PV = np.zeros([0], dtype=int) #vector to store the indices of PV buses
vec_P = np.zeros((n - 1, 1), dtype=float) #data of active power
vec_Q = np.zeros((n - 1, 1), dtype=float) #data of reactive power
vec_V = np.zeros((n - 1, 1), dtype=float) #data of voltage magnitude
vec_W = np.zeros((n - 1, 1), dtype=float) #voltage magnitude squared
for i in range(df_bus.shape[0]):  #find the voltage specified for the slack
    if df_bus.iloc[i, 0] == 0:
        V_slack = df_bus.iloc[i, 3]
for i in range(df_bus.shape[0]):  #store the data of both PQ and PV
    vec_P[df_bus.iloc[i, 0] - 1] = df_bus.iloc[i, 1]  #-1 to start at 0
    if df_bus.iloc[i, 4] == 'PQ':
        vec_Q[df_bus.iloc[i, 0] - 1] = df_bus.iloc[i, 2]  #-1 to start at 0
        num_busos_PQ += 1 #identify as PQ bus
        vec_busos_PQ = np.append(vec_busos_PQ, df_bus.iloc[i, 0])
    elif df_bus.iloc[i, 4] == 'PV':
        vec_V[df_bus.iloc[i, 0] - 1] = df_bus.iloc[i, 3]  #-1 to start at 0
        num_busos_PV += 1 #identify as PV bus
        vec_busos_PV = np.append(vec_busos_PV, df_bus.iloc[i, 0])
for i in range(n - 1):
    vec_W[i] = vec_V[i] * vec_V[i]
# --------------------------- INITIAL DATA: BUSES INFORMATION. DONE
# --------------------------- PREPARING IMPLEMENTATION
prof = 30  #depth
U = np.zeros((prof, n - 1), dtype=complex) #voltages
U_re = np.zeros((prof, n - 1), dtype=complex) #real part of voltages
U_im = np.zeros((prof, n - 1), dtype=complex) #imaginary part of voltages
X = np.zeros((prof, n - 1), dtype=complex) #X=1/conj(U)
X_re = np.zeros((prof, n - 1), dtype=complex) #real part of X
X_im = np.zeros((prof, n - 1), dtype=complex) #imaginary part of X
Q = np.zeros((prof, n - 1), dtype=complex)  #unknown reactive powers
# .......................CALCULATION OF TERMS [0]
vec_U = np.dot(np.linalg.inv(Y), vec_Y0)  #each element is roughly equal to 1
for i in range(n - 1):
    U[0, i] = 1  #force each element to 1. Tiny difference
for i in range(n - 1):
    X[0, i] = 1 / np.conj(U[0, i]) #could force them to be 1 directly
    U_re[0, i] = np.real(U[0, i])  #could force them to be 1 directly
    U_im[0, i] = np.imag(U[0, i])  #could force them to be 0 directly
    X_re[0, i] = np.real(X[0, i])  #could force them to be 1 directly
    X_im[0, i] = np.imag(X[0, i])  #could force them to be 0 directly
# .......................CALCULATION OF TERMS [0]. DONE
# .......................CALCULATION OF TERMS [1]
llarg = 2 * num_busos_PQ + 3 * num_busos_PV #number of unknowns
RHS = np.zeros((llarg, 1), dtype=float) #vector of the RHS data. Each element
    has to be real
k = 0 #index that will go through the rows
for i in range(n - 1): #filling the vector RHS
    if i + 1 in vec_busos_PQ:
        RHS[k] = np.real(
            (V_slack - 1) * vec_Y0[i, 0] + (vec_P[i, 0] - vec_Q[i, 0] * 1j) * X
    [0, i] + U[0, i] * vec_shunts[i, 0])
```

```
        RHS[k + 1] = np.imag(
            (V_slack - 1) * vec_Y0[i, 0] + (vec_P[i, 0] - vec_Q[i, 0] * 1j) * X
    [0, i] + U[0, i] * vec_shunts[i, 0])
        k = k + 2
    elif i + 1 in vec_busos_PV:
        RHS[k] = np.real((V_slack - 1) * vec_Y0[i, 0] + (vec_P[i, 0]) * X[0, i]
    + U[0, i] * vec_shunts[i, 0])
        RHS[k + 1] = np.imag((V_slack - 1) * vec_Y0[i, 0] + (vec_P[i, 0]) * X[0,
     i] + U[0, i] * vec_shunts[i, 0])
        RHS[k + 2] = vec_W[i, 0] - 1
        k = k + 3
mat = np.zeros((llarg, 2 * (n - 1) + num_busos_PV), dtype=complex) #constant
    matrix
k = 0 #index that will go through the rows
l = 0 #index that will go through the columns
for i in range(n - 1): #fill the matrix
    if i + 1 in vec_busos_PQ:
        l = 0
        for j in range(n - 1):
            if j+1 not in vec_busos_PV:
                mat[k, l] = G[i, j]
                mat[k + 1, l] = B[i, j]
                mat[k, l + 1] = -B[i, j]
                mat[k + 1, l + 1] = G[i, j]
                l = l + 2 #2 columns done
            if j+1 in vec_busos_PV:
                mat[k, l] = G[i, j]
                mat[k + 1, l] = B[i, j]
                mat[k, l + 1] = -B[i, j]
                mat[k + 1, l + 1] = G[i, j]
                mat[k, l + 2] = 0
                mat[k + 1, l + 2] = 0
                l=l+3 #3 columns done
        k = k + 2 #2 rows done
    elif i + 1 in vec_busos_PV:
        l = 0
        for j in range(n - 1):
            if j+1 not in vec_busos_PV:
                mat[k, l] = G[i, j]
                mat[k + 1, l] = B[i, j]
                mat[k, l + 1] = -B[i, j]
                mat[k + 1, l + 1] = G[i, j]
                mat[k + 2, l ] = 0
                mat[k + 2, l + 1] = 0
                l=l+2 #2 columns done
            if j+1 in vec_busos_PV:
                if j==i:
                    mat[k, l] = G[i, j]
                    mat[k + 1, l] = B[i, j]
                    mat[k + 2, l] = 2 * U_re[0, i]
                    mat[k, l + 1] = -B[i, j]
                    mat[k + 1, l + 1] = G[i, j]
                    mat[k + 2, l + 1] = 2 * U_im[0, i]
                    mat[k, l + 2] = -X_im[0, i]
                    mat[k + 1, l + 2] = X_re[0, i]
                    mat[k + 2, l + 2] = 0
                    l = l + 3 #3 columns done
                elif j!=i:
                    mat[k, l] = G[i, j]
                    mat[k + 1, l] = B[i, j]
                    mat[k + 2, l] = 0
                    mat[k, l + 1] = -B[i, j]
                    mat[k + 1, l + 1] = G[i, j]
                    mat[k + 2, l + 1] = 0
                    mat[k, l + 2] = 0
                    mat[k + 1, l + 2] = 0
                    mat[k + 2, l + 2] = 0
                    l = l + 3
        k = k + 3 #3 rows done
dfx=pd.DataFrame(mat)
dfx.to_excel('Resultats3.xlsx', index=False, header=False) #to check the matrix
LHS = np.dot(np.linalg.inv(mat), RHS) #although mat only has to be inverted once
```

```
k = 0
for i in range(n - 1):  #fill unknowns
    if i + 1 in vec_busos_PQ:
        U_re[1, i] = LHS[k, 0]
        U_im[1, i] = LHS[k + 1, 0]
        k = k + 2
    elif i + 1 in vec_busos_PV:
        U_re[1, i] = LHS[k, 0]
        U_im[1, i] = LHS[k + 1, 0]
        Q[0, i] = LHS[k + 2, 0]
        k = k + 3
for i in range(n - 1):  #complete the matrices U and X
    U[1, i] = U_re[1, i] + U_im[1, i] * 1j
    X[1, i] = (-X[0, i] * np.conj(U[1, i])) / np.conj(U[0, i])
    X_re[1, i] = np.real(X[1, i])
    X_im[1, i] = np.imag(X[1, i])
# .......................CALCULATION OF TERMS [1]. DONE
# .......................CALCULATION OF TERMS [>=2]
def convX(U,X,c,i): #convolution between U^* and X
    suma=0
    for k in range(1,c+1): #c+1 perquè arribi fins a c
        suma=suma+np.conj(U[k,i])*X[c-k,i]
    return suma
def sumaPV1(X,Q,c,i): #convolution between X and Q
    suma=0
    for k in range(1,c):
        suma=suma+X[k,i]*Q[c-1-k,i]
    return suma
def sumaPV3(U,c,i): #convolution between U and U
    suma=0
    for k in range(1,c):
        suma=suma+U[k,i]*np.conj(U[c-k,i])
    return suma
for c in range(2,prof): #c defines the current depth
    RHS = np.zeros((llarg, 1), dtype=complex) #is real but a warning appears if
    it is not defined as complex
    k = 0
    for i in range(n - 1): #fill the vector RHS
        if i + 1 in vec_busos_PQ:
            RHS[k] = np.real((vec_P[i, 0] - vec_Q[i, 0] * 1j) * X[c-1, i] + U[c
    -1, i] * vec_shunts[i, 0])
            RHS[k + 1] = np.imag((vec_P[i, 0] - vec_Q[i, 0] * 1j) * X[c-1, i] +
    U[c-1, i] * vec_shunts[i, 0])
            k = k + 2
        elif i + 1 in vec_busos_PV:
            RHS[k] = np.real(sumaPV1(X,Q,c,i)*(-1)*1j+U[c-1,i]*vec_shunts[i,0]+X
    [c-1,i]*vec_P[i,0]) #afegit això últim!!
            RHS[k+1]=np.imag(sumaPV1(X,Q,c,i)*(-1)*1j+U[c-1,i]*vec_shunts[i,0]+X
    [c-1,i]*vec_P[i,0]) #afegit això últim!!
            RHS[k+2]=-sumaPV3(U,c,i)
            k = k + 3
    LHS = np.dot(np.linalg.inv(mat), RHS) #no need to invert another time the
    matrix mat!
    k = 0
    for i in range(n - 1):  #grab the unknowns
        if i + 1 in vec_busos_PQ:
            U_re[c, i] = LHS[k, 0]
            U_im[c, i] = LHS[k + 1, 0]
            k = k + 2
        elif i + 1 in vec_busos_PV:
            U_re[c, i] = LHS[k, 0]
            U_im[c, i] = LHS[k + 1, 0]
            Q[c-1, i] = LHS[k + 2, 0]
            k = k + 3
    for i in range(n - 1):  #complete the matrices
        U[c, i] = U_re[c, i] + U_im[c, i] * 1j
        X[c,i]=-convX(U,X,c,i)/ np.conj(U[0, i])
        X_re[c, i] = np.real(X[c, i])
        X_im[c, i] = np.imag(X[c, i])
matAdf=pd.DataFrame(U)
matAdf.to_excel('Resultats.xlsx', index=False, header=False) #to check the
    voltages
```

```
# .......................CALCULATION OF TERMS [>=2]. DONE
# --------------------------- CHECK DATA
U_final=np.zeros((n-1,1),dtype=complex) #final voltages
for j in range(n-1):
    suma=0
    for i in range(prof):
        suma=suma+U[i,j]
    U_final[j,0]=suma
print(U_final)
print(abs(U_final)) #absolute value
I_serie=np.dot(Y,U_final) #current flowing through series elements
I_inj_slack=np.zeros((n-1,1),dtype=complex) #current injected by the slack
for i in range(n-1):
    I_inj_slack[i,0]=vec_Y0[i,0]*V_slack
I_shunt=np.zeros((n-1,1),dtype=complex) #current through shunts
for i in range(n-1):
    I_shunt[i,0]=-U_final[i]*vec_shunts[i] #change the sign again
I_generada=I_serie-I_inj_slack+I_shunt #current leaving the bus
I_gen2=np.zeros((n-1,1),dtype=complex) #current entering the bus
for i in range(n-1):
    if i + 1 in vec_busos_PQ:
        I_gen2[i,0]=(vec_P[i,0]-vec_Q[i,0]*1j)/np.conj(U_final[i,0])
    elif i + 1 in vec_busos_PV:
        I_gen2[i,0]=(vec_P[i,0]-sum(Q[:,i]*1j))/np.conj(U_final[i,0])
print(I_gen2-I_generada) #balance of current. Should be almost 0
Ydf=pd.DataFrame(Q) #to check the unknown reactive power
Ydf.to_excel('Resultats2.xlsx', index=False, header=False)
```

References

A. Trias, "The Holomorphic Embedding Load Flow method," 2012 IEEE Power and Energy Society General Meeting, San Diego, CA, 2012, pp. 1-8.

S. Rao, Y. Feng, D. J. Tylavsky and M. K. Subramanian, "The Holomorphic Embedding Method Applied to the Power-Flow Problem," in IEEE Transactions on Power Systems, vol. 31, no. 5, pp. 3816-3828, Sept. 2016.

A. Trias. HELM: The Holomorphic Embedding Load-Flow Method. Foundations and Implementations. Foundations and Trends in Electrical Energy Systems, vol. 3, no. 3-4, pp. 140-370, 2018.

S. Rao. Exploration of a Scalable Holomorphic Embedding Method Formulation for Power System Analysis Applications. PhD Dissertation. Arizona State University. 2017.

B. Schmidt. Implementation and Evaluation of the Holomorphic Embedding Load Flow Method. Master Thesis. Technical University of Munich. 2015.

M. K. Subramanian. Application of Holomorphic Embedding to the Power-Flow Problem. Master Thesis. Arizona State University. 2014