```
#FORMULACIÓ ORIGINAL

# LLIBRERIES
import numpy as np
from mpmath import mp  # per tenir més decimals
mp.dps = 50
import pandas as pd
import matplotlib.pyplot as plt
from scipy.sparse import csc_matrix, coo_matrix
from scipy.sparse import lil_matrix, diags, hstack, vstack
from scipy.sparse.linalg import spsolve, factorized

np.set_printoptions(linewidth=2000, edgeitems=1000, suppress=True)
pd.set_option('display.max_rows', 5000)
pd.set_option('display.max_columns', 1000)
pd.set_option('display.width', 2000)
pd.set_option("display.precision", 5)
# FI LLIBRERIES

# DEFINICIÓ D'OBJECTES INICIALS
df_top = pd.read_excel('IEEE14.xlsx', sheet_name='Topologia')  # dades de la topologia
df_bus = pd.read_excel('IEEE14.xlsx', sheet_name='Busos')  # dades dels busos

n = df_bus.shape[0]  # nombre de busos, inclou l'oscil·lant
nl = df_top.shape[0]  # nombre de línies

A = np.zeros((n, nl), dtype=int)  # matriu d'incidència, formada per 1, -1 i 0
L = np.zeros((nl, nl), dtype=complex)  # matriu amb les branques
np.fill_diagonal(L, [1 / (df_top.iloc[i, 2] + df_top.iloc[i, 3] * 1j) for i in range(nl)])
A[df_top.iloc[range(nl), 0], range(nl)] = 1
A[df_top.iloc[range(nl), 1], range(nl)] = -1

Yseries = np.dot(np.dot(A, L), np.transpose(A))  # matriu de les branques sèrie, es reduirà
Yseries_slack = np.zeros((n, n), dtype=complex)
Yseries_slack[:, :] = Yseries[:, :]  # també contindrà les admitàncies amb el bus oscil·lant

Ytap = np.zeros((n, n), dtype=complex)  # agrupa les asimetries
for i in range(nl):  # emplenar matriu quan hi ha trafo de relació variable
    tap = df_top.iloc[i, 5]
    ang_tap = df_top.iloc[i, 6]
    tap = abs(tap) * np.cos(ang_tap) + abs(tap) * np.sin(ang_tap) * 1j
    if tap != 1 or ang_tap != 0:
        Ytap[df_top.iloc[i, 0], df_top.iloc[i, 0]] += 1 / (df_top.iloc[i, 2] + df_top.iloc[i, 3] * 1j) / \
                                        (tap * np.conj(tap)) - 1 / (
                                            df_top.iloc[i, 2] + df_top.iloc[i, 3] * 1j)
        Ytap[df_top.iloc[i, 1], df_top.iloc[i, 1]] += 1 / (df_top.iloc[i, 2] + df_top.iloc[i, 3] * 1j) \
                                        - 1 / (df_top.iloc[i, 2] + df_top.iloc[i, 3] * 1j)
        Ytap[df_top.iloc[i, 0], df_top.iloc[i, 1]] += - 1 / (df_top.iloc[i, 2] + df_top.iloc[i, 3] * 1j) / \
                                        (np.conj(tap)) + 1 / (
                                            df_top.iloc[i, 2] + df_top.iloc[i, 3] * 1j)
        Ytap[df_top.iloc[i, 1], df_top.iloc[i, 0]] += - 1 / (df_top.iloc[i, 2] + df_top.iloc[i, 3] * 1j) / \
                                        (tap) + 1 / (
                                            df_top.iloc[i, 2] + df_top.iloc[i, 3] * 1j)

vec_Pi = np.zeros(n, dtype=float)  # dades de potència activa
vec_Qi = np.zeros(n, dtype=float)  # dades de potència reactiva
vec_Vi = np.zeros(n, dtype=float)  # dades de tensió
vec_Wi = np.zeros(n, dtype=float)  # tensió al quadrat

pq = []  # índexs dels busos PQ
pv = []  # índexs dels busos PV
sl = []  # índexs dels busos oscil·lants
vec_Pi[:] = np.nan_to_num(df_bus.iloc[:, 1])  # emplenar el vector de números
vec_Qi[:] = np.nan_to_num(df_bus.iloc[:, 2])
vec_Vi[:] = np.nan_to_num(df_bus.iloc[:, 3])
V_sl = []  # tensions dels oscil·lants
```

```
for i in range(n):  # cerca per a guardar els índexs
    if df_bus.iloc[i, 5] == 'PQ':
        pq.append(i)
    elif df_bus.iloc[i, 5] == 'PV':
        pv.append(i)
    elif df_bus.iloc[i, 5] == 'Slack':
        sl.append(i)
        V_sl.append(df_bus.iloc[i, 3] * (np.cos(df_bus.iloc[i, 4]) + np.sin(df_bus.iloc[i, 4]) * 1j))

pq = np.array(pq)  # índexs en forma de vector
pv = np.array(pv)
sl = np.array(sl)
npq = len(pq)  # nombre de busos PQ
npv = len(pv)  # nombre de busos PV
nsl = len(sl)  # nombre de busos oscil·lants
npqpv = npq + npv  # nombre de busos incògnita

pqpv_x = np.sort(np.r_[pq, pv])  # ordenar els vectors amb incògnites
pqpv = []
[pqpv.append(int(pqpv_x[i])) for i in range(len(pqpv_x))]  # convertir els índexs a enters

pq_x = pq  # guardar els índexs originals
pv_x = pv

vec_P = vec_Pi[pqpv]  # agafar la part del vector necessària
vec_Q = vec_Qi[pqpv]
vec_V = vec_Vi[pqpv]

factor_carrega = 1.0  # factor de càrrega de les potències de tots els busos
vec_P = factor_carrega * vec_P
vec_Q = factor_carrega * vec_Q

Yshunts = np.zeros(n, dtype=complex)  # admitàncies en paral·lel, degudes a les capacitats
for i in range(nl):
    if df_top.iloc[i, 5] == 1:  # si la relació de transformació és unitària
        Yshunts[df_top.iloc[i, 0]] += df_top.iloc[i, 4] * 1j  # es donen en forma d'admitàncies
        Yshunts[df_top.iloc[i, 1]] += df_top.iloc[i, 4] * 1j
    else:
        Yshunts[df_top.iloc[i, 0]] += df_top.iloc[i, 4] * 1j / (df_top.iloc[i, 5] ** 2)
        Yshunts[df_top.iloc[i, 1]] += df_top.iloc[i, 4] * 1j

for i in range(n):  # afegir les càrregues d'admitància constant
    Yshunts[df_bus.iloc[i, 0]] += df_bus.iloc[i, 6] * 1
    Yshunts[df_bus.iloc[i, 0]] += df_bus.iloc[i, 7] * 1j

Yshunts_slack = np.zeros(n, dtype=complex)  # inclou els busos oscil·lants, no es redueix
Yshunts_slack[:] = Yshunts[:]

df = pd.DataFrame(data=np.c_[Yshunts, vec_Pi, vec_Qi, vec_Vi], columns=['Ysh', 'P0', 'Q0', 'V0'])
print(df)  # imprimir dades inicials

Yslack = Yseries_slack[:, sl]  # les columnes pertanyents als oscil·lants
# FI DEFINICIÓ OBJECTES INICIALS

# PREPARACIÓ DE LA IMPLEMENTACIÓ
prof = 60  # nombre de coeficients de les sèries

U = np.zeros((prof, npqpv), dtype=complex)  # sèries de voltatges
U_re = np.zeros((prof, npqpv), dtype=float)  # part real de voltatges
U_im = np.zeros((prof, npqpv), dtype=float)  # part imaginària de voltatges
X = np.zeros((prof, npqpv), dtype=complex)  # tensió inversa conjugada
X_re = np.zeros((prof, npqpv), dtype=float)  # part real d'X
X_im = np.zeros((prof, npqpv), dtype=float)  # part imaginària d'X
Q = np.zeros((prof, npqpv), dtype=complex)  # sèries de potències reactives

W = vec_V * vec_V  # mòdul de les tensions al quadrat
dim = 2 * npq + 3 * npv  # nombre d'incògnites
```

```
Yseries = Yseries[np.ix_(pqpv, pqpv)]  # reduir per a deixar de banda els oscil·lants
Ytaps = Ytap[np.ix_(pqpv, pqpv)]  # reduir per a deixar de banda els oscil·lants
Ytapslack = Ytap[np.ix_(pqpv, sl)]  # columnes de la matriu d'admitàncies asimètrica per als oscil·lants

G = np.real(Yseries)  # part real de la matriu simètrica
B = np.imag(Yseries)  # part imaginària de la matriu simètrica
Yshunts = Yshunts[pqpv]  # reduir per a deixar de banda els slack
Yslack = Yslack[pqpv, :]  # enllaç amb els busos PQ i PV

nsl_compt = np.zeros(n, dtype=int)  # nombre de busos oscil·lants trobats abans d'un bus
compt = 0
for i in range(n):
    if i in sl:
        compt += 1
    nsl_compt[i] = compt
if npq > 0:
    pq_ = pq - nsl_compt[pq]
else:
    pq_ = []
if npv > 0:
    pv_ = pv - nsl_compt[pv]
else:
    pv_ = []
if nsl > 0:
    sl_ = sl - nsl_compt[sl]

pqpv_x = np.sort(np.r_[pq_, pv_])  # ordenar els nous índexs dels busos PQ i PV
pqpv_ = []
[pqpv_.append(int(pqpv_x[i])) for i in range(len(pqpv_x))]  # convertir els índexs a enters
# FI PREPARACIÓ DE LA IMPLEMENTACIÓ

# TERMES [0]
U_re[0, pqpv_] = 1  # estat de referència
U_im[0, pqpv_] = 0
U[0, pqpv_] = U_re[0, pqpv_] + U_im[0, pqpv_] * 1j
X[0, pqpv_] = 1 / np.conj(U[0, pqpv_])
X_re[0, pqpv_] = np.real(X[0, pqpv_])
X_im[0, pqpv_] = np.imag(X[0, pqpv_])
Q[0, pv_] = 0
# FI TERMES [0]

# TERMES [1]
range_pqpv = np.arange(npqpv)  # tots els busos ordenats
valor = np.zeros(npqpv, dtype=complex)

prod = np.dot((Yslack[pqpv_, :]), V_sl[:])  # intensitat que injecten els oscil·lants
prod2 = np.dot((Ytaps[pqpv_, :]), U[0, :])  # itensitat amb la matriu asimètrica

valor[pq_] = - prod[pq_] \
           + np.sum(Yslack[pq_, :], axis=1) \
           - Yshunts[pq_] * U[0, pq_] \
           + (vec_P[pq_] - vec_Q[pq_] * 1j) * X[0, pq_] \
           - prod2[pq_] \
           - np.sum(Ytapslack[pq_, :], axis=1)

valor[pv_] = - prod[pv_] \
           + np.sum(Yslack[pv_, :], axis=1) \
           - Yshunts[pv_] * U[0, pv_] \
           + vec_P[pv_] * X[0, pv_] \
           - prod2[pv_] \
           - np.sum(Ytapslack[pv_, :], axis=1)

RHS = np.r_[valor.real, valor.imag, W[pv_] - 1]  # vector de la dreta del sistema d'equacions

VRE = coo_matrix((2 * U_re[0, pv_], (np.arange(npv), pv_)), shape=(npv, npqpv)).tocsc()  # matriu dispersa COO a compr.
VIM = coo_matrix((2 * U_im[0, pv_], (np.arange(npv), pv_)), shape=(npv, npqpv)).tocsc()
XIM = coo_matrix((-X_im[0, pv_], (pv_, np.arange(npv))), shape=(npqpv, npv)).tocsc()
```

```
XRE = coo_matrix((X_re[0, pv_], (pv_, np.arange(npv))), shape=(npqpv, npv)).tocsc()
BUIT = csc_matrix((npv, npv))  # matriu dispera comprimida

MATx = vstack((hstack((G, -B, XIM)),
               hstack((B, G, XRE)),
               hstack((VRE, VIM, BUIT))), format='csc')

MAT_LU = factorized(MATx.tocsc())  # matriu factoritzada, només cal fer-ho una vegada
LHS = MAT_LU(RHS)  # vector amb les solucions

U_re[1, :] = LHS[:npqpv]  # actualització de les incògnites
U_im[1, :] = LHS[npqpv:2 * npqpv]
Q[1, pv_] = LHS[2 * npqpv:]

U[1, :] = U_re[1, :] + U_im[1, :] * 1j
X[1, :] = (-X[0, :] * np.conj(U[1, :])) / np.conj(U[0, :])
X_re[1, :] = X[1, :].real
X_im[1, :] = X[1, :].imag
# FI TERMES [1]


def convqx(q, x, i, cc):  # convolució entre Q i X
    suma = 0
    for k in range(cc):
        suma += q[k, i] * x[cc - k, i]
    return suma


def convv(u, i, cc):  # convolució entre U i U conjugada
    suma = 0
    for k in range(1, cc):
        suma += u[k, i] * np.conj(u[cc - k, i])
    return np.real(suma)


def convx(u, x, i, cc):  # convolució entre U i X
    suma = 0
    for k in range(1, cc + 1):
        suma += np.conj(u[k, i]) * x[cc - k, i]
    return suma


# TERMES [2]
prod2 = np.dot((Ytaps[pqpv_, :]), U[1, :])  # intensitat amb l'asimètrica pels PQ i PV
prod3 = np.dot((Ytapslack[pqpv_, :]), V_sl[:])  # intensitat amb l'asimètrica pels oscil·lants
c = 2  # profunditat actual

valor[pq_] = - Yshunts[pq_] * U[c - 1, pq_] \
           + (vec_P[pq_] - vec_Q[pq_] * 1j) * X[c - 1, pq_] \
           - prod2[pq_] \
           - np.sum(Ytapslack[pq_, :], axis=1) * (-1) \
           - prod3[pq_]

valor[pv_] = - Yshunts[pv_] * U[c - 1, pv_] \
           + vec_P[pv_] * X[c - 1, pv_] \
           - 1j * convqx(Q, X, pv_, c) \
           - prod2[pv_] \
           - np.sum(Ytapslack[pv_, :], axis=1) * (-1) \
           - prod3[pv_]

RHS = np.r_[valor.real, valor.imag, -convv(U, pv_, c)]  # vector de la dreta del sistema d'equacions

LHS = MAT_LU(RHS)  # vector amb les solucions

U_re[c, :] = LHS[:npqpv]  # actualització d'incògnites
U_im[c, :] = LHS[npqpv:2 * npqpv]
Q[c, pv_] = LHS[2 * npqpv:]
```

```
U[c, :] = U_re[c, :] + U_im[c, :] * 1j
X[c, :] = - convx(U, X, range_pqpv, c) / np.conj(U[0, :])
X_re[c, :] = X[c, :].real
X_im[c, :] = X[c, :].imag
# FI TERMES [2]

# TERMES [c>=3]
for c in range(3, prof):  # c és la profunditat actual
    prod2 = np.dot((Ytaps[pqpv_, :]), U[c - 1, :])  # intensitat amb l'asimètrica dels PQ i PV

    valor[pq_] = - Yshunts[pq_] * U[c - 1, pq_] \
                 + (vec_P[pq_] - vec_Q[pq_] * 1j) * X[c - 1, pq_] \
                 - prod2[pq_]

    valor[pv_] = - Yshunts[pv_] * U[c - 1, pv_] \
                 + vec_P[pv_] * X[c - 1, pv_] \
                 - 1j * convqx(Q, X, pv_, c) \
                 - prod2[pv_]

    RHS = np.r_[valor.real, valor.imag, -convv(U, pv_, c)]  # vector de la dreta del sistema d'equacions

    LHS = MAT_LU(RHS)  # vector amb les solucions

    U_re[c, :] = LHS[:npqpv]  # actualització de les incògnites
    U_im[c, :] = LHS[npqpv:2 * npqpv]
    Q[c, pv_] = LHS[2 * npqpv:]

    U[c, :] = U_re[c, :] + U_im[c, :] * 1j
    X[c, :] = - convx(U, X, range_pqpv, c) / np.conj(U[0, :])
    X_re[c, :] = X[c, :].real
    X_im[c, :] = X[c, :].imag
# FI TERMES [c>=3]

# RESULTATS
Pfi = np.zeros(n, dtype=complex)  # potència activa final
Qfi = np.zeros(n, dtype=complex)  # potència reactiva final
U_sum = np.zeros(n, dtype=complex)  # tensió a partir la suma de coeficients
U_pa = np.zeros(n, dtype=complex)  # tensió amb aproximants de Padé
U_th = np.zeros(n, dtype=complex)  # tensió amb aproximants de Thévenin
U_ait = np.zeros(n, dtype=complex)  # tensió amb Delta d'Aitken
U_shanks = np.zeros(n, dtype=complex)  # tensió amb transformacions de Shanks
U_rho = np.zeros(n, dtype=complex)  # tensió amb Rho de Wynn
U_eps = np.zeros(n, dtype=complex)  # tensió amb Èpsilon de Wynn
U_theta = np.zeros(n, dtype=complex)  # tensió amb Theta de Brezinski
U_eta = np.zeros(n, dtype=complex)  # tensió amb Eta de Bauer
Q_eps = np.zeros(n, dtype=complex)
Q_ait = np.zeros(n, dtype=complex)
Q_rho = np.zeros(n, dtype=complex)
Q_theta = np.zeros(n, dtype=complex)
Q_eta = np.zeros(n, dtype=complex)
Q_sum = np.zeros(n, dtype=complex)
Q_shanks = np.zeros(n, dtype=complex)
Sig_re = np.zeros(n, dtype=complex)  # part real de sigma
Sig_im = np.zeros(n, dtype=complex)  # part imaginària de sigma

Ybus = Yseries_slack + diags(Yshunts_slack) + Ytap  # matriu d'admitàncies total

from Funcions import pade4all, thevenin, Sigma, aitken, shanks, rho, epsilon, theta, eta  # importar funcions

# SUMA
U_sum[pqpv] = np.sum(U[:, pqpv_], axis=0)
U_sum[sl] = V_sl
if npq > 0:
    Q_sum[pq] = vec_Q[pq_]
if npv > 0:
    Q_sum[pv] = np.sum(Q[:, pv_], axis=0)
```

```
# FI SUMA

# PADÉ
Upa = pade4all(prof, U[:, :], 1)
Qpa = pade4all(prof, Q[:, pv_], 1)
U_pa[sl] = V_sl
U_pa[pqpv] = Upa
Pfi[pqpv] = vec_P[pqpv_]
if npq > 0:
    Qfi[pq] = vec_Q[pq_]
if npv > 0:
    Qfi[pv] = Qpa
Pfi[sl] = np.nan
Qfi[sl] = np.nan
# FI PADÉ

limit = 6  # límit per a no utilitzar tots els coeficients
if limit > prof:
    limit = prof - 1

# SIGMA
Ux1 = np.copy(U)
Sig_re[pqpv] = np.real(Sigma(Ux1, X, prof - 1, V_sl))
Sig_im[pqpv] = np.imag(Sigma(Ux1, X, prof - 1, V_sl))
Sig_re[sl] = np.nan
Sig_im[sl] = np.nan

arrel = np.zeros(n, dtype=float)  # discriminant
arrel[sl] = np.nan
arrel[pqpv] = 0.25 + np.abs(Sig_re[pqpv]) - np.abs(Sig_im[pqpv]) ** 2
# FI SIGMA

# THÉVENIN
Ux1 = np.copy(U)
for i in pq:  # només pels busos PQ
    U_th[i] = thevenin(Ux1[:limit, i - nsl_compt[i]], X[:limit, i - nsl_compt[i]])
# FI THÉVENIN

# RECURRENTS
Ux = np.copy(U)
Qx = np.copy(Q)

for i in range(npqpv):
    U_ait[i] = aitken(Ux[:, i], limit)
    U_shanks[i] = shanks(Ux[:, i], limit)
    U_rho[i] = rho(Ux[:, i], limit)
    U_eps[i] = epsilon(Ux[:, i], limit)
    U_theta[i] = theta(Ux[:, i], limit)

    if i in pq_:
        Q_ait[i + nsl_compt[i]] = vec_Q[i]
        Q_shanks[i + nsl_compt[i]] = vec_Q[i]
        Q_rho[i + nsl_compt[i]] = vec_Q[i]
        Q_eps[i + nsl_compt[i]] = vec_Q[i]
        Q_theta[i + nsl_compt[i]] = vec_Q[i]

    elif i in pv_:
        Q_ait[i + nsl_compt[i]] = aitken(Qx[:, i], limit)
        Q_shanks[i + nsl_compt[i]] = shanks(Qx[:, i], limit)
        Q_rho[i + nsl_compt[i]] = rho(Qx[:, i], limit)
        Q_eps[i + nsl_compt[i]] = epsilon(Qx[:, i], limit)
        Q_theta[i + nsl_compt[i]] = theta(Qx[:, i], limit)

U_ait[pqpv] = U_ait[pqpv_]
U_ait[sl] = V_sl
Q_ait[sl] = np.nan
U_shanks[pqpv] = U_shanks[pqpv_]
```

```
U_shanks[sl] = V_sl
Q_shanks[sl] = np.nan
U_rho[pqpv] = U_rho[pqpv_]
U_rho[sl] = V_sl
Q_rho[sl] = np.nan
U_eps[pqpv] = U_eps[pqpv_]
U_eps[sl] = V_sl
Q_eps[sl] = np.nan
U_theta[pqpv] = U_theta[pqpv_]
U_theta[sl] = V_sl
Q_theta[sl] = np.nan
# FI RECURRENTS

# ERRORS
S_out_sum = np.asarray(U_sum) * np.conj(np.asarray(np.dot(Ybus, U_sum)))
S_in_sum = (Pfi[:] + 1j * Q_sum[:])
error_sum = S_in_sum - S_out_sum

S_out = np.asarray(U_pa) * np.conj(np.asarray(np.dot(Ybus, U_pa)))
S_in = (Pfi[:] + 1j * Qfi[:])
error = S_in - S_out  # error final de potències amb Padé

S_out_ait = np.asarray(U_ait) * np.conj(np.asarray(np.dot(Ybus, U_ait)))
S_in_ait = (Pfi[:] + 1j * Q_ait[:])
error_ait = S_in_ait - S_out_ait

S_out_shanks = np.asarray(U_shanks) * np.conj(np.asarray(np.dot(Ybus, U_shanks)))
S_in_shanks = (Pfi[:] + 1j * Q_shanks[:])
error_shanks = S_in_shanks - S_out_shanks

S_out_rho = np.asarray(U_rho) * np.conj(np.asarray(np.dot(Ybus, U_rho)))
S_in_rho = (Pfi[:] + 1j * Q_rho[:])
error_rho = S_in_rho - S_out_rho

S_out_eps = np.asarray(U_eps) * np.conj(np.asarray(np.dot(Ybus, U_eps)))
S_in_eps = (Pfi[:] + 1j * Q_eps[:])
error_eps = S_in_eps - S_out_eps

S_out_theta = np.asarray(U_theta) * np.conj(np.asarray(np.dot(Ybus, U_theta)))
S_in_theta = (Pfi[:] + 1j * Q_theta[:])
error_theta = S_in_theta - S_out_theta
# FI ERRORS

df = pd.DataFrame(
    np.c_[np.abs(U_sum), np.angle(U_sum), np.abs(U_pa), np.angle(U_pa), np.real(Sig_re), np.real(Sig_im),
          np.abs(error[0, :])], columns=['|V| sum', 'A. sum', '|V| Padé', 'A. Padé', 'Sigma re', 'Sigma im', 'S error'])
print(df)

#print('Error màxim amb suma: ', max(abs(np.r_[error_sum[0, pqpv]])))
print('Error màxim amb Padé: ', max(abs(np.r_[error[0, pqpv]])))
#print('Error màxim amb Delta d\'Aitken: ', max(abs(np.r_[error_ait[0, pqpv]])))
#print('Error màxim amb transformacions de Shanks: ', max(abs(np.r_[error_shanks[0, pqpv]])))
#print('Error màxim amb Rho de Wynn: ', max(abs(np.r_[error_rho[0, pqpv]])))
#print('Error màxim amb Èpsilon de Wynn: ', max(abs(np.r_[error_eps[0, pqpv]])))
#print('Error màxim amb Theta de Brezinski: ', max(abs(np.r_[error_theta[0, pqpv]])))



# --------------------------- PADÉ-WEIERSTRASS (P-W)
s0 = [0.5, 1]  # vector de les s parcials
ng = len(s0)  # nombre de graons menys 1

s0p = []  # producte de les (1-s0)
s0p.append(1)
Vs0p = []  # producte dels V(s0)
Vs0p.append(1)
for i in range(1, ng):
    s0p.append(s0p[i - 1] * (1 - s0[i - 1]))
```

```
Vw = V_sl[0]  # voltatge del bus oscil·lant, només se n'admet 1
Vs = np.zeros((ng, 2), dtype=complex)  # auxiliar per a trobar Vs0
Vs0 = np.zeros(ng, dtype=complex)  # tensions del bus oscil·lant a cada graó
Vs[:, 0] = 1
Vs[0, 1] = s0p[0] * (Vw - 1)
Vs0[0] = Vs[0, 0] + s0[0] * Vs[0, 1]
Vs0p.append(Vs0p[0] * Vs0[0])
for i in range(1, ng):
    Vs[i, 1] = s0p[i] * (Vw - 1) / Vs0p[i]
    Vs0[i] = Vs[i, 0] + s0[i] * Vs[i, 1]
    Vs0p.append(Vs0p[i] * Vs0[i])

prof_pw = prof  # profunditat de les sèries del P-W

Up = np.zeros((prof_pw, npqpv, ng), dtype=complex)  # tensions prima incògnita
Up_re = np.zeros((prof_pw, npqpv, ng), dtype=float)
Up_im = np.zeros((prof_pw, npqpv, ng), dtype=float)
Xp = np.zeros((prof_pw, npqpv, ng), dtype=complex)  # invers conjugat de la tensió prima
Xp_re = np.zeros((prof_pw, npqpv, ng), dtype=float)
Xp_im = np.zeros((prof_pw, npqpv, ng), dtype=float)
Qp = np.zeros((prof_pw, npqpv, ng), dtype=complex)  # potència reactiva prima incògnita
Us0 = np.zeros((n, ng), dtype=complex)  # tensions parcials
Qs0 = np.zeros((n, ng), dtype=complex)  # potències reactives parcials

gamma_x = 0  # variable que agrupa els termes que no depenen de la s^(r)
Yahat = np.copy(Ytap)  # admitàncies on les files no sumen 0
Ybhat = np.copy(Yseries_slack)  # admitàncies on les files sumen 0

for kg in range(ng - 1):  # calcular cada graó
    Us0[sl, kg] = Vs0[kg]  # completar columna del bus oscil·lant
    if kg == 0:  # si és el primer graó, utilitzar els objectes del MIH bàsic
        Us0[pqpv, kg] = pade4all(prof - 1, U[:, pqpv_], s0[kg])
        if npv > 0:
            Qs0[pv, kg] = pade4all(prof - 1, Q[:, pv_], s0[kg])
    else:  # si no és el primer graó, calcular la solució parcial anterior
        Us0[pqpv, kg] = pade4all(prof_pw - 1, Up[:, pqpv_, kg - 1], s0[kg])
        if npv > 0:
            Qs0[pv, kg] = pade4all(prof_pw - 1, Qp[:, pv_, kg - 1], s0[kg])

    Yahat = np.copy(Ytap)
    Ybhat = np.copy(Yseries_slack)
    for i in range(n):
        if i not in sl:  # per la fila del bus oscil·lant no cal fer-ho
            for j in range(n):
                Yahat[i, j] = Yahat[i, j] * np.prod(Us0[j, :kg + 1], axis=0) * np.prod(np.conj(Us0[i, :kg + 1]), axis=0)
                Ybhat[i, j] = Ybhat[i, j] * np.prod(Us0[j, :kg + 1], axis=0) * np.prod(np.conj(Us0[i, :kg + 1]), axis=0)

    gamma_x += s0[kg] * s0p[kg]  # actualització

    Ybtilde = np.copy(Ybhat)  # matriu evolucionada
    if npq > 0:
        Ybtilde[pq, pq] += gamma_x * Yshunts_slack[pq] * np.prod(abs(Us0[pq, :kg + 1]) ** 2, axis=1) \
                   - gamma_x * (Pfi[pq] - Qfi[pq] * 1j)
    if npv > 0:
        Ybtilde[pv, pv] += gamma_x * Yshunts_slack[pv] * np.prod(abs(Us0[pv, :kg + 1]) ** 2, axis=1) \
                   - gamma_x * Pfi[pv] + np.sum(Qs0[pv, :], axis=1) * 1j

    Ybtilde[:, :] += gamma_x * Yahat[:, :]  # part que no s'incrusta amb s'
    Yahat[:, :] = (1 - gamma_x) * Yahat[:, :]  # part que no s'incrusta amb s'

    # TERMES [0]
    Up[0, :, kg] = 1  # estat de referència
    Qp[0, :, kg] = 0  # estat de referència

    Up_re[0, :, kg] = np.real(Up[0, :, kg])
    Up_im[0, :, kg] = np.imag(Up[0, :, kg])
```

```
    Xp[0, :, kg] = 1 / Up[0, :, kg]
    Xp_re[0, :, kg] = np.real(Xp[0, :, kg])
    Xp_im[0, :, kg] = np.imag(Xp[0, :, kg])
    # FI TERMES [0]

    Yahatred = Yahat[np.ix_(pqpv, pqpv)]  # Ya reduïda
    Yahatw = Yahat[np.ix_(pqpv, sl)]  # Ya amb l'oscil·lant
    Ybtildered = Ybtilde[np.ix_(pqpv, pqpv)]  # Yb reduïda
    Ybtildew = Ybtilde[np.ix_(pqpv, sl)]  # Yb amb l'oscil·lant

    # TERMES [1]
    prod1 = np.dot(Ybtildew[pqpv_, 0], Vs[kg + 1, 1])  # producte de la Yb amb el bus oscil·lant
    prod2 = np.dot(Yahatred[pqpv_, :], Up[0, :, kg])  # producte de la Ya amb la tensió incògnita
    prod3 = np.dot(Yahatw[pqpv_, 0], Vs[kg + 1, 0])  # producte de la Ya amb el bus oscil·lant

    if npq > 0:
        valor[pq_] = - prod1[pq_] \
                     - prod2[pq_] \
                     - prod3[pq_] \
                     - (1 - gamma_x) * Yshunts[pq_] * Up[0, pq_, kg] * np.prod(abs(Us0[pq, :kg + 1]), axis=1) ** 2 \
                     + (1 - gamma_x) * (Pfi[pq] - Qfi[pq] * 1j) * Xp[0, pq_, kg]

    if npv > 0:
        valor[pv_] = - prod1[pv_] \
                     - prod2[pv_] \
                     - prod3[pv_] \
                     - (1 - gamma_x) * Yshunts[pv_] * Up[0, pv_, kg] * np.prod(abs(Us0[pv, :kg + 1]), axis=1) ** 2 \
                     + (1 - gamma_x) * Pfi[pv] * Xp[0, pv_, kg]

        RHS = np.r_[valor.real, valor.imag, W[pv_] / np.prod(abs(Us0[pv, :kg + 1]), axis=1) ** 2 - 1]  # vector de la dreta del
sistema d'equacions
    else:
        RHS = np.r_[valor.real, valor.imag]

    gamma = np.zeros(npqpv, dtype=complex)
    if npq > 0:
        gamma[pq_] = gamma_x * (Pfi[pq] - Qfi[pq] * 1j)  # gamma pels busos PQ
    if npv > 0:
        gamma[pv_] = gamma_x * Pfi[pv] - np.sum(Qs0[pv, :], axis=1) * 1j  # gamma pels busos PV

    Gf = np.real(Ybtildered)  # part real de la matriu Yb reduïda
    Bf = np.imag(Ybtildered)  # part imaginària de la matriu Yb reduïda

    VRE = coo_matrix((2 * Up_re[0, pv_, kg], (np.arange(npv), pv_)), shape=(npv, npqpv)).tocsc()  # en forma de blocs
    VIM = coo_matrix((2 * Up_im[0, pv_, kg], (np.arange(npv), pv_)), shape=(npv, npqpv)).tocsc()
    XIM = coo_matrix((-Xp_im[0, pv_, kg], (pv_, np.arange(npv))), shape=(npqpv, npv)).tocsc()
    XRE = coo_matrix((Xp_re[0, pv_, kg], (pv_, np.arange(npv))), shape=(npqpv, npv)).tocsc()
    BUIT = csc_matrix((npv, npv))

    M1 = np.copy(Gf)
    M2 = np.copy(-Bf)
    M3 = np.copy(Bf)
    M4 = np.copy(Gf)

    for i in range(npqpv):
        for j in range(npqpv):
            if i == j:
                M1[i, j] += np.real(2 * gamma[i])  # emplenar amb gamma
                M3[i, j] += np.imag(2 * gamma[i])

    MAT = vstack((hstack((M1, M2, XIM)),
                  hstack((M3, M4, XRE)),
                  hstack((VRE, VIM, BUIT))), format='csc')

    MAT_LU = factorized(MAT.tocsc())  # factoritzar, només cal una vegada
    LHS = MAT_LU(RHS)  # vector amb les solucions del graó
```

```
Up_re[1, :, kg] = LHS[:npqpv]  # emplenar les incògnites
Up_im[1, :, kg] = LHS[npqpv: 2 * npqpv]
Qp[1, pv_, kg] = LHS[2 * npqpv:]

Up[1, :, kg] = Up_re[1, :, kg] + Up_im[1, :, kg] * 1j
Xp[1, :, kg] = - np.conj(Up[1, :, kg]) * Xp[0, :, kg] / np.conj(Up[0, :, kg])
Xp_re[1, :, kg] = np.real(Xp[1, :, kg])
Xp_im[1, :, kg] = np.imag(Xp[1, :, kg])
# FI TERMES [1]


def convxv(xp, up, i, cc, kkg):  # convolució entre X i U primes
    suma = 0
    for k in range(1, cc):
        suma = suma + xp[k, i, kkg] * np.conj(up[cc - k, i, kkg])
    return suma


def convqx(qp, xp, i, cc, kkg):  # convolució entre Q i X primes
    suma = 0
    for k in range(1, cc):
        suma = suma + qp[k, i, kkg] * xp[cc - k, i, kkg]
    return suma


def convu(up, i, cc, kkg):  # convolució entre U prima i U prima conjugada
    suma = 0
    for k in range(1, cc):
        suma = suma + up[k, i, kkg] * np.conj(up[cc - k, i, kkg])
    return suma


def convxx(u, x, i, cc, kkg):  # convolució entre U i X primes, pel càlcul de les Xp
    suma = 0
    for k in range(1, cc + 1):
        suma += np.conj(u[k, i, kkg]) * x[cc - k, i, kkg]
    return suma

# TERMES [2]
prod2 = np.dot(Yahatred[pqpv_, :], Up[1, :, kg])  # producte entre Ya reduïda i tensions dels PQ i PV
prod3 = np.dot(Yahatw[pqpv_, 0], Vs[kg + 1, 1])  # producte entre Ya de l'oscil·lant i tensió d'aquest

if npq > 0:
    valor[pq_] = - prod2[pq_] \
                 - prod3[pq_] \
                 - (1 - gamma_x) * Yshunts[pq_] * Up[1, pq_, kg] * np.prod(abs(Us0[pq, :kg + 1]), axis=1) ** 2 \
                 + (1 - gamma_x) * (Pfi[pq] - Qfi[pq] * 1j) * Xp[1, pq_, kg] \
                 + gamma_x * (Pfi[pq] - Qfi[pq] * 1j) * (- convxv(Xp, Up, pq_, 2, kg))

if npv > 0:
    valor[pv_] = - prod2[pv_] \
                 - prod3[pv_] \
                 - (1 - gamma_x) * Yshunts[pv_] * Up[1, pv_, kg] * np.prod(abs(Us0[pv, :kg + 1]), axis=1) ** 2 \
                 + (1 - gamma_x) * Pfi[pv] * Xp[1, pv_, kg] \
                 - convqx(Qp, Xp, pv_, 2, kg) * 1j \
                 + gamma[pv_] * (- convxv(Xp, Up, pv_, 2, kg))
    RHS = np.r_[valor.real, valor.imag, np.real(-convu(Up, pv_, 2, kg))]  # vector de la dreta del sistema d'equacions
else:
    RHS = np.r_[valor.real, valor.imag]

LHS = MAT_LU(RHS)  # vector amb les solucions del graó

Up_re[2, :, kg] = LHS[:npqpv]  # actualització de les incògnites
Up_im[2, :, kg] = LHS[npqpv: 2 * npqpv]
Qp[2, pv_, kg] = LHS[2 * npqpv:]

Up[2, :, kg] = Up_re[2, :, kg] + Up_im[2, :, kg] * 1j
```

```
    Xp[2, :, kg] = - convxx(Up, Xp, range_pqpv, 2, kg) / np.conj(Up[0, :, kg])
    Xp_re[2, :, kg] = np.real(Xp[2, :, kg])
    Xp_im[2, :, kg] = np.imag(Xp[2, :, kg])
    # FI TERMES [2]

    # TERMES [c>=3]
    for c in range(3, prof_pw):
        prod2 = np.dot(Yahatred[pqpv_, :], Up[c - 1, :, kg])  # producte entre Ya reduïda i tensions dels PQ i PV

        if npq > 0:
            valor[pq_] = - prod2[pq_] \
                    - (1 - gamma_x) * Yshunts[pq_] * Up[c - 1, pq_, kg] * np.prod(abs(Us0[pq, :kg + 1]),
                                                                                  axis=1) ** 2 \
                    + (1 - gamma_x) * (Pfi[pq] - Qfi[pq] * 1j) * Xp[c - 1, pq_, kg] \
                    + gamma_x * (Pfi[pq] - Qfi[pq] * 1j) * (- convxv(Xp, Up, pq_, c, kg))
        if npv > 0:
            valor[pv_] = - prod2[pv_] \
                    - (1 - gamma_x) * Yshunts[pv_] * Up[c - 1, pv_, kg] * np.prod(abs(Us0[pv, :kg + 1]),
                                                                                  axis=1) ** 2 \
                    + (1 - gamma_x) * Pfi[pv] * Xp[c - 1, pv_, kg] \
                    - convqx(Qp, Xp, pv_, c, kg) * 1j \
                    + gamma[pv_] * (- convxv(Xp, Up, pv_, c, kg))
            RHS = np.r_[valor.real, valor.imag, np.real(-convu(Up, pv_, c, kg))]  # vector de la dreta del sistema d'equacions
        else:
            RHS = np.r_[valor.real, valor.imag]

        LHS = MAT_LU(RHS)  # vector amb les solucions del graó

        Up_re[c, :, kg] = LHS[:npqpv]  # actualització de les incògnites
        Up_im[c, :, kg] = LHS[npqpv: 2 * npqpv]
        Qp[c, pv_, kg] = LHS[2 * npqpv:]

        Up[c, :, kg] = Up_re[c, :, kg] + Up_im[c, :, kg] * 1j
        Xp[c, :, kg] = - convxx(Up, Xp, range_pqpv, c, kg) / np.conj(Up[0, :, kg])
        Xp_re[c, :, kg] = np.real(Xp[c, :, kg])
        Xp_im[c, :, kg] = np.imag(Xp[c, :, kg])
    # FI TERMES [c>=3]

Upfinal = np.zeros(n, dtype=complex)  # tensió prima final amb Padé
Xpfinal = np.zeros(n, dtype=complex)  # X prima final amb Padé
Qpfinal = np.zeros(n, dtype=complex)  # potència reactiva prima final amb Padé

Upfinal[pqpv] = pade4all(prof_pw - 1, Up[:, pqpv_, ng - 2], 1)
Upfinal[sl] = V_sl[0]
Xpfinal[pqpv] = pade4all(prof_pw - 1, Xp[:, pqpv_, ng - 2], 1)
Xpfinal[sl] = 1 / np.conj(V_sl[0])

if npv > 0:
    Qpfinal[pv] = pade4all(prof_pw - 1, Qp[:, pv_, ng - 2], 1)
    Qpfinal[sl] = np.nan

Ufinalx = Upfinal[pqpv] * np.prod(Us0[pqpv, :ng - 1], axis=1)  # tensió final
if npv > 0:
    Qfinalx = Qpfinal[pv] + np.sum(Qs0[pv, :], axis=1)
    Qfi[pv] = Qfinalx  # potència reactiva definitiva
Ufinal = np.zeros(n, dtype=complex)  # tensió definitiva per a tots els busos
Ufinal[sl] = V_sl[0]
Ufinal[pqpv] = Ufinalx[:]

S_out = np.asarray(Ufinal) * np.conj(np.asarray(np.dot(Ybus, Ufinal)))  # potència que surt dels busos
S_in = (Pfi[:] + 1j * Qfi[:])  # potència que entra als busos

errorx = S_in - S_out  # error de potències
err = max(abs(np.r_[errorx[0, pqpv]]))  # màxim error de potències amb P-W
print('Error P-W amb Padé: ', abs(err))
```

```
# TEST DE CONVERGÈNCIA DELS APROXIMANTS DE PADÉ
col = ng - 2  # columna de la qual mirem la convergència
Us0x = np.copy(Us0)

if col == 0:  # si les tensions anteriors són del MIH bàsic
    Us0[pqpv, col] -= pade4all(prof - 3, U[:, pqpv_], s0[col])  # la diferència entre ara i amb 2 coeficients menys
else:  # si les tensions anteriors són del P-W
    Us0[pqpv, col] -= pade4all(prof_pw - 3, Up[:, pqpv_, col - 1], s0[col])

tol = 1e-10  # tolerància arbitrària
falla = False
ik = 1
while falla is False and ik < npqpv:  # es comprova la tolerància
    if abs(Us0[ik, col]) > tol and ik not in sl:
        falla = True
    ik += 1

if falla is True:
    print('Incorrecte, massa error')
else:
    print('Correcte, poc error')



"""
# ALTRES:
# .....VISUALITZACIÓ DE LA MATRIU .....
from pylab import *
#MATx és la del sistema del MIH, MAT la del P-W
Amm = abs(MATx.todense())  # passar a densa
figure(1)
f = plt.figure()
imshow(Amm, interpolation='nearest', cmap=plt.get_cmap('gist_heat'))
#plt.gray()  # en escala de grisos
#plt.show()
plt.spy(Amm)  # en blanc i negre
plt.show()

f.savefig("figura.pdf", bbox_inches='tight')

Bmm = coo_matrix(MATx)  # passar a dispersa
density = Bmm.getnnz() / np.prod(Bmm.shape) * 100  # convertir a percentual
print('Densitat: ' + str(density) + ' %')
"""

# .....DOMB-SYKES .....

bb = np. zeros((prof, npqpv), dtype=complex)
for j in range(npqpv):
    for i in range(3, len(U)-1):
        #bb[i, j] = np. abs(np.sqrt((U[i+1, j] * U[i-1, j] - U[i, j] ** 2) / (U[i, j] * U[i-2, j] - U[i-1, j] ** 2)))
        bb[i, j] = (U[i, j]) / (U[i-1, j])
        #print(bb[i, j])

vec_1n = np. zeros(prof)
for i in range(3, prof):
    #vec_1n[i] = 1 / i
    vec_1n[i] = i

bus = 1  # gràfic Domb-Sykes d'aquest bus

plt.plot(vec_1n[3:len(U)-1], abs(bb[3:len(U)-1, bus]), 'ro ', markersize=2)
plt.show()

print(1/max(abs(bb[-2, :])))
print(1/min(abs(bb[-2, :])))
#print(1/(abs(bb[-2, 28])))
```

```
# print(bb[3:len(U) - 2, 28])
# n_ord = abs(bb[len(U) - 2, 28]) - vec_1n[len(U) - 2] * (abs(bb[len(U) - 2, 28]) - abs(bb[len(U) - 3, 28])) / (vec_1n[len(U) - 2] -
vec_1n[len(U) - 3])
# print('radi: ' + str(1 / n_ord))


# .....GRÀFIC SIGMA .....
a = []
b = []
c = []

x = np.linspace(-0.25, 1, 1000)
y = np.sqrt(0.25 + x)
a.append(x)
b.append(y)
c.append(-y)

plt.plot(np.real(Sig_re), np.real(Sig_im), 'ro', markersize=2)
plt.plot(x, y)
plt.plot(x, -y)
plt.ylabel('Sigma im')
plt.xlabel('Sigma re')
plt.title('Gràfic Sigma')
plt.show()

# .....EXTRA.....
```