```python
import numpy as np
import numba as nb
from mpmath import mp  # per tenir més decimals
mp.dps = 50


#@nb.jit

# PADÉ

def pade4all(ordre, coeff_mat, s):
    """
    ordre: profunditat seleccionada
    coeff_mat: matriu o vector de coeficients
    s: valor en el qual s'avalua la sèrie , sovint s=1
    """

    if coeff_mat.ndim > 1:  # nombre de columnes
        nbus = coeff_mat.shape[1]
    else:
        nbus = coeff_mat.ndim

    voltatges = np.zeros(nbus, dtype=complex)  # resultats finals

    if ordre % 2 != 0:
        nn = int(ordre / 2)
        L = nn
        M = nn
        for d in range(nbus):
            if nbus > 1:
                rhs = coeff_mat[L + 1:L + M + 1, d]  # vector de la dreta , conegut
            else:
                rhs = coeff_mat[L + 1:L + M + 1]

            C = np.zeros((M, M), dtype=complex)  # matriu del sistema
            for i in range(M):
                k = i + 1
                if nbus > 1:
                    C[i, :] = coeff_mat[L - M + k:L + k, d]
                else:
                    C[i, :] = coeff_mat[L - M + k:L + k]

            b = np.zeros(rhs.shape[0] + 1, dtype=complex)  # denominador
            x = np.linalg.solve(C, -rhs)
            b[0] = 1
            b[1:] = x[::-1]

            a = np.zeros(L + 1, dtype=complex)  # numerador
            if nbus > 1:
                a[0] = coeff_mat[0, d]
            else:
                a[0] = coeff_mat[0]

            for i in range(L):  # completar numerador
                val = complex(0)
                k = i + 1
                for j in range(k + 1):
                    if nbus > 1:
                        val += coeff_mat[k - j, d] * b[j]
                    else:
                        val += coeff_mat[k - j] * b[j]
                a[i + 1] = val

            p = complex(0)
            q = complex(0)

            for i in range(len(a)):  # avaluar numerador i denominador
```

```
            p += a[i] * s ** i
        for i in range(len(b)):
            q += b[i] * s ** i

        voltatges[d] = p / q

        ppb = np.poly1d(b)  # convertir a polinomi
        ppa = np.poly1d(a)
        ppbr = ppb.r  # pols
        ppar = ppa.r  # zeros
else:
    nn = int(ordre / 2)
    L = nn
    M = nn - 1
    for d in range(nbus):
        if nbus > 1:
            rhs = coeff_mat[M + 2: 2 * M + 2, d]  # vector de la dreta , conegut
        else:
            rhs = coeff_mat[M + 2: 2 * M + 2]

        C = np.zeros((M, M), dtype=complex)  # matriu del sistema
        for i in range(M):
            k = i + 1
            if nbus > 1:
                C[i, :] = coeff_mat[L - M + k:L + k, d]
            else:
                C[i, :] = coeff_mat[L - M + k:L + k]

        b = np.zeros(rhs.shape[0] + 1, dtype=complex)  # denominador
        x = np.linalg.solve(C, -rhs)
        b[0] = 1
        b[1:] = x[::-1]

        a = np.zeros(L + 1, dtype=complex)  # numerador
        if nbus > 1:
            a[0] = coeff_mat[0, d]
        else:
            a[0] = coeff_mat[0]

        for i in range(1, L):  # completar numerador
            val = complex(0)
            for j in range(i + 1):
                if nbus > 1:
                    val += coeff_mat[i - j, d] * b[j]
                else:
                    val += coeff_mat[i - j] * b[j]
            a[i] = val

        val = complex(0)
        for j in range(L):
            if nbus > 1:
                val += coeff_mat[M - j + 1, d] * b[j]
            else:
                val += coeff_mat[M - j + 1] * b[j]
        a[L] = val

        p = complex(0)
        q = complex(0)

        for i in range(len(a)):  # avaluar numerador i denominador
            p += a[i] * s ** i
        for i in range(len(b)):
            q += b[i] * s ** i

        voltatges[d] = p / q

        ppb = np.poly1d(b)  # convertir a polinomi
```

```python
        ppa = np.poly1d(a)
        ppbr = ppb.r  # pols
        ppar = ppa.r  # zeros

    return voltatges


# THÉVENIN

@nb.jit
def thevenin(U, X):
    """
    U: vector de coeficients de tensió
    X: vector de coeficients de la tensió inversa conjugada
    """

    complex_type = nb.complex128
    n = len(U)

    r_3 = np. zeros(n, complex_type)
    r_2 = np. zeros(n, complex_type)
    r_1 = np. zeros(n, complex_type)
    r_0 = np. zeros(n, complex_type)

    T_03 = np. zeros(n, complex_type)
    T_02 = np. zeros(n, complex_type)
    T_01 = np. zeros(n, complex_type)
    T_00 = np. zeros(n, complex_type)
    T_13 = np. zeros(n, complex_type)
    T_12 = np. zeros(n, complex_type)
    T_11 = np. zeros(n, complex_type)
    T_10 = np. zeros(n, complex_type)
    T_23 = np. zeros(n, complex_type)
    T_22 = np. zeros(n, complex_type)
    T_21 = np. zeros(n, complex_type)
    T_20 = np. zeros(n, complex_type)


    r_0[0] = -1  # inicialització de residus
    r_1[0:n - 1] = U[1:n] / U[0]
    r_2[0:n - 2] = U[2:n] / U[0] - U[1] * np.conj(U[0]) / U[0] * X[1:n - 1]

    T_00[0] = -1  # inicializació de polinomis
    T_01[0] = -1
    T_02[0] = -1
    T_10[0] = 0
    T_11[0] = 1 / U[0]
    T_12[0] = 1 / U[0]
    T_20[0] = 0
    T_21[0] = 0
    T_22[0] = -U[1] * np.conj(U[0]) / U[0]

    for l in range(n):  # càlculs successius
        a = (r_2[0] * r_1[0]) / (- r_0[1] * r_1[0] + r_0[0] * r_1[1] - r_0[0] * r_2[0])
        b = -a * r_0[0] / r_1[0]
        c = 1 - b
        T_03[0] = b * T_01[0] + c * T_02[0]
        T_03[1:n] = a * T_00[0:n - 1] + b * T_01[1:n] + c * T_02[1:n]
        T_13[0] = b * T_11[0] + c * T_12[0]
        T_13[1:n] = a * T_10[0:n - 1] + b * T_11[1:n] + c * T_12[1:n]
        T_23[0] = b * T_21[0] + c * T_22[0]
        T_23[1:n] = a * T_20[0:n - 1] + b * T_21[1:n] + c * T_22[1:n]
        r_3[0:n-2] = a * r_0[2:n] + b * r_1[2:n] + c * r_2[1:n - 1]

        if l == n - 1:  # si és l'última iteració
            t_0 = T_03
            t_1 = T_13
```

```
            t_2 = T_23

        r_0[:] = r_1[:]  # actualització de residus
        r_1[:] = r_2[:]
        r_2[:] = r_3[:]

        T_00[:] = T_01[:]  # actualització de polinomis
        T_01[:] = T_02[:]
        T_02[:] = T_03[:]
        T_10[:] = T_11[:]
        T_11[:] = T_12[:]
        T_12[:] = T_13[:]
        T_20[:] = T_21[:]
        T_21[:] = T_22[:]
        T_22[:] = T_23[:]

        r_3 = np.zeros(n, complex_type)
        T_03 = np.zeros(n, complex_type)
        T_13 = np.zeros(n, complex_type)
        T_23 = np.zeros(n, complex_type)

    usw = -np.sum(t_0) / np.sum(t_1)
    sth = -np.sum(t_2) / np.sum(t_1)

    sigma_bo = sth / (usw * np.conj(usw))

    u = 0.5 + np.sqrt(0.25 + np.real(sigma_bo) - np. imag(sigma_bo)**2) + np.imag(sigma_bo)*1j  # branca estable
    #u = 0.5 - np.sqrt(0.25 + np.real(sigma_bo) - np.imag(sigma_bo) ** 2) + np.imag(sigma_bo) * 1j  # branca inestable

    ufinal = u * usw  # resultat final

    return ufinal


# SIGMA

def Sigma(coeff_matU, coeff_matX, ordre, V_slack):
    """
    coeff_matU: matriu de coeficients de tensió
    coeff_matX: matriu de coeficients de la tensió inversa conjugada
    ordre: profunditat seleccionada
    V_slack: tensions dels busos oscil·lants
    """

    if len(V_slack) > 1:
        print('Els valors poden no ser correctes')

    V0 = V_slack[0]  # tensió del bus oscil·lant de referència
    coeff_A = np.copy(coeff_matU)  # adaptar els coeficients per a la funció racional
    coeff_B= np.copy(coeff_matX)

    coeff_A[0, :] = 1
    for i in range(1, coeff_matU.shape[0]):
        coeff_A[i, :] = coeff_matU[i, :] - (V0 - 1) * coeff_A[i-1, :]
    coeff_B[0, :] = 1
    for i in range(1, coeff_matX.shape[0]):
        coeff_B[i, :] = coeff_matX[i, :] + (V0 - 1) * coeff_matX[i-1, :]

    nbus = coeff_matU.shape[1]
    sigmes = np.zeros(nbus, dtype=complex)

    if ordre % 2 == 0:
        M = int(ordre / 2) - 1
    else:
        M = int(ordre / 2)

    for d in range(nbus):  # emplenar objectes del sistema d'equacions
```

```
        a = coeff_A[1:2 * M + 2, d]
        b = coeff_B[0:2 * M + 1, d]
        C = np.zeros((2 * M + 1, 2 * M + 1), dtype=complex)  # matriu del sistema
        for i in range(2 * M + 1):
            if i < M:
                C[1 + i:, i] = a[:2 * M - i]
            else:
                C[i - M:, i] = - b[:3 * M - i + 1]

        lhs = np.linalg.solve(C, -a)
        sigmes[d] = np.sum(lhs[M:])/(np.sum(lhs[:M]) + 1)

    return sigmes


# DELTA D'AITKEN

@nb.jit
def aitken(U, limit):
    """
    U: vector de coeficients de tensió
    limit: profunditat seleccionada
    """



    def S(Um, k):  # funció de sumes parcials
        suma = np.sum(Um[:k + 1])
        return suma


    complex_type = nb.complex128
    Um = U[:limit]
    n = limit
    T = np.zeros(n-2, complex_type)

    for i in range(len(T)):  # emplenar el vector
        T[i] = S(Um, i + 2) - (S(Um, i + 1) - S(Um, i))**2 / ((S(Um, i + 2) - S(Um, i + 1)) - (S(Um, i + 1) - S(Um, i)))

    return T[-1]  # l'últim element , que en principi és la millor aproximació


# TRANSFORMACIONS DE SHANKS

@nb.jit
def shanks(U, limit):
    """
    U: vector de coeficients de tensió
    limit: profunditat seleccionada
    """



    def S(Um, k):  # funció de sumes parcials
        suma = np.sum(Um[:k + 1])
        return suma


    complex_type = nb.complex128
    Um = U[:limit + 1]
    n = limit
    n_trans = 3  # nombre de transformacions
    T = np.zeros((n, n_trans), complex_type)

    for lk in range(n_trans):  # emplenar la taula
        for i in range(n - 2 * lk):
            if lk == 0:
                T[i, lk] = S(Um, i + 2) - (S(Um, i + 1) - S(Um, i))**2 / ((S(Um, i + 2) - S(Um, i + 1)) - (S(Um, i + 1) - S(Um, i)))
            else:
```

```python
            T[i, lk] = T[i + 2, lk - 1] - (T[i + 2, lk - 1]-T[i + 1, lk - 1])**2 / \
                    ((T[i + 2, lk - 1]-T[i + 1, lk - 1]) - (T[i + 1, lk - 1]-T[i, lk - 1]))

    return T[n - 2 * (n_trans - 1) - 1, n_trans - 1]


# RHO DE WYNN

@nb.jit
def rho(U, limit):
    """
    U: vector de coeficients de tensió
    limit: profunditat seleccionada
    """



    def S(Um, k):  # funció de sumes parcials
        suma = np.sum(Um[:k + 1])
        return suma


    complex_type = nb.complex128
    Um = U[:limit]
    n = limit

    mat = np.zeros((n, n + 1), complex_type)
    for i in range(n):
        mat[i, 1] = S(Um, i)  # emplenar de sumes parcials
    for j in range(2, n + 1):  # completar la resta de columnes
        for i in range(0, n + 1 - j):
            mat[i, j] = mat[i + 1, j - 2] + (j - 1) / (mat[i + 1, j - 1] - mat[i, j - 1])
    if limit % 2 == 0:
        return mat[0, n - 1]
    else:
        return mat[0, n]


# ÈPSILON DE WYNN

@nb.jit
def epsilon(U, limit):
    """
    U: vector de coeficients de tensió
    limit: profunditat seleccionada
    """



    def S(Um, k):  # funció de sumes parcials
        suma = np.sum(Um[:k + 1])
        return suma


    complex_type = nb.complex128
    Um = U[:limit]
    n = limit

    mat = np.zeros((n, n + 1), complex_type)
    for i in range(n):
        mat[i, 1] = S(Um, i)  # emplenar de sumes parcials
    for j in range(2, n + 1):  # completar la resta de columnes
        for i in range(0, n + 1 - j):
            mat[i, j] = mat[i + 1, j - 2] + 1 / (mat[i + 1, j - 1] - mat[i, j - 1])

    if limit % 2 == 0:
        return mat[0, n - 1]
    else:
        return mat[0, n]
```

```python
# THETA DE BREZINSKI

@nb.jit
def theta(U, limit):
    """
    U: vector de coeficients de tensió
    limit: profunditat seleccionada
    """



    def S(Um, k):  # funció de sumes parcials
        suma = np.sum(Um[:k + 1])
        return suma


    complex_type = nb.complex128
    n = limit
    Um = np.zeros(n, complex_type)
    Um[:] = U[:limit]

    mat = np.zeros((n, n + 1), complex_type)  # inicialització de la matriu
    for i in range(n):
        mat[i, 1] = S(Um, i)  # emplenar de sumes parcials
    for j in range(2, n + 1):  # completar la resta de columnes
        if j % 2 == 0:
            for i in range(0, n + 1 - j):
                mat[i, j] = mat[i + 1, j - 2] + 1 / (mat[i + 1, j - 1] - mat[i, j - 1])
        else:
            for i in range(0, n + 1 - j):
                mat[i, j] = mat[i + 1, j - 2] + ((mat[i + 2, j - 2] - mat[i + 1, j - 2]) * (mat[i + 2, j - 1] - mat[i + 1, j - 1])) \
                            / (mat[i + 2, j - 1] - 2 * mat[i + 1, j - 1] + mat[i, j - 1])
    if limit % 2 == 0:
        return mat[0, n - 1]
    else:
        return mat[0, n]


# ETA DE BAUER

@nb.jit
def eta(U, limit):
    """
    U: vector de coeficients de tensió
    limit: profunditat seleccionada
    """

    complex_type = nb.complex128
    n = limit
    Um = np.zeros(n, complex_type)
    Um[:] = U[:limit]

    mat = np.zeros((n, n+1), complex_type)
    mat[:, 0] = np.inf  # infinit
    mat[:, 1] = Um[:]

    for j in range(2, n + 1):  # emplenar la taula
        if j % 2 == 0:
            for i in range(0, n + 1 - j):
                mat[i, j] = 1 / (1 / mat[i + 1, j - 2] + 1 / (mat[i + 1, j - 1]) - 1 / (mat[i, j - 1]))
        else:
            for i in range(0, n + 1 - j):
                mat[i, j] = mat[i + 1, j - 2] + mat[i + 1, j - 1] - mat[i, j - 1]

    return np.sum(mat[0, 1:])
```