```python
def eigen_jacobi_step(a,eps):
    t = a.copy()
    ## Look up the maximal element
    maxim = 0; lig = 0; col = 0;
    for i in np.arange(1,n):
        for j in np.arange(0,i):
            if (np.abs(a[i,j])>maxim):
                lig = i; col = j;
                maxim = np.abs(a[i,j]);

    if (maxim>eps): ## Apply the rotation
        cotang = (a[col,col]-a[lig,lig])/(2*a[lig,col]);
        if (cotang > 0):
            tang = -cotang+np.sqrt(cotang*cotang+1);
        else:
            tang = -cotang-np.sqrt(cotang*cotang+1);
        cosi = 1/np.sqrt(1+tang*tang);
        sinu = tang*cosi;
        for i in np.arange(0,n): ## Transform the matrix
            if (i<>lig) & (i<>col):
                t[lig,i] = cosi * a[lig,i] - sinu * a[col,i];
                t[i,lig] = t[lig,i];
                t[col,i] = sinu * a[lig,i] + cosi * a[col,i];
                t[i,col] = t[col,i];
            t[lig,lig] = a[lig,lig] - tang * a[lig,col];
            t[col,col] = a[col,col] + tang * a[lig,col];
            t[lig,col] = 0;
            t[col,lig] = 0;
        return(t,False)
    else:
        return(t,True)

def eigen_jacobi(a,maxsteps,eps):
    n = a.shape[0];
    d = a.copy();
    stp = 0; b = False;

    while (stp < maxsteps) and (not(b)) and
      (np.linalg.norm(d - np.diag(np.diag(d))) > eps):
        (d,b) = eigen_jacobi_step(d, eps)
        stp = stp+1

    return(d,stp)
```

Algorithme 3.1: Méthode de Jacobi (a est une matrice carrée de dimension $n \times n$, *maxsteps* est une borne sur le nombre de pas de calculs réalisés, et *eps* est une borne maximale sur les éléments extra-digaonaux permettant d'arrêter le calcul prématurément.)

```python
def eigen_tridiag(a):
    n = a.shape[0];
    t = a.copy()

    for i in np.arange(0,n-2):
        # Computation of the new vector
        sub = t[i+1:n, i+1:n];
        vec = t[i+1:n, i];
        nor = np.linalg.norm(vec,2);
        nvc = np.zeros([n-i-1,1]);
        nvc[0] = nor;
        # Create Householder matrix
        if (np.abs(vec[0])<>nor):
            hou = qr.hh_from_vects(vec,nvc);
            t[i+1:n,i] = nvc;
            t[i,i+1:n] = nvc.T;
            # Optimisation
            p = sub*hou;
            c = (hou.T*p)[0,0];
            for j in np.arange(i+1,n):
                for k in np.arange(i+1,n):
                    t[j,k] = t[j,k] \
                    - 2*p[j-i-1,0]*hou[k-i-1,0] \
                    - 2*p[k-i-1,0]*hou[j-i-1,0] \
                    + 4*c*hou[k-i-1,0]*hou[j-i-1,0];
    return t
```

Algorithme 3.2: Méthode de Householder pour obtenir une matrice tridiagonale (a est une matrice carrée de dimension $n \times n$.)

```
def givens_sign(t,x):
    n = t.shape[0]
    p0 = 1;
    p1 = t[0,0] - x;
    c = 1 if (p1<0) else 0;

    for i in np.arange(1,n):
        p2 = p1 * (t[i,i] - x) - (t[i,i-1])**2 * p0
        if (p1*p2<0):
            c = c+1;
        p0 = p1;
        p1 = p2;
    return c

def givens_nth_eig(t,k,eps, Nmax):
    a = -1.; b = 1.; cpt = 0;

    while ((givens_sign(t,a))>=k):
        a = a * 2;
    while ((givens_sign(t,b))<k):
        b = b * 2;
    while ((b-a>eps) and (cpt < Nmax)):
        cpt = cpt + 1
        c = (a+b)/2;
        if ((givens_sign(t,c))>=k):
            b = c;
        else:
            a = c;
    return c
```

Algorithme 3.3: Méthode de localisation de Givens (t est une matrice carrée tridiagonale de dimension $n \times n$, k est le numéro de le valeur propre recherchée, et *eps* la précision de la recherche. La fonction auxiliaire `givens_sign` calcule le nombre de changements de signe dans la suite de polynômes.)

```
def eigen_qr(a,step):

    t = a.copy();
    for i in np.arange(1,step):
        q,r = np.linalg.qr(t);
        t = r*q;

    return t
```

Algorithme 3.4: Méthode QR – VERSION RAPIDE !!! (t est une matrice carrée de dimension $n \times n$, *step* est le nombre d'étapes demandé dans la méthode QR.)