10 Prima Lezione A

Uno dei problemi che spesso capita di dover affrontare in fisica è di dover diagonalizzare una matrice cioè trovare λ e $\mathbf{v}$ tale che:

$$A\mathbf{v} = \lambda\mathbf{v} \quad . \tag{13}$$

A volte siamo interessati a tutti gli autovalori, a volte solo ai più grandi o a i più piccoli, dipende un po' dai casi, quindi vogliamo un po' far vedere qualche metodo per calcolare la decomposizione spettrale di una data matrice.

10.1 Metodo delle potenze

Il più semplice metodo da poter implementare è il metodo delle potenze, il quale è un algoritmo iterativo la cui regola di iterazione è:

$$\mathbf{v}_{k+1} = \frac{A\mathbf{v}_k}{||A\mathbf{v}_k||} = \frac{A^k\mathbf{v}_0}{||A^k\mathbf{v}_0||} \quad . \tag{14}$$

Dove $\mathbf{v}_0$ è un certo vettore iniziale che scegliamo a caso. L'idea è abbastanza semplice, se A è diagonalizzabile allora possiamo decomporre un vettore generico nella base degli autovettori di A:

$$\mathbf{v}_0 = \alpha_1 v_1 + \cdots + \alpha_n v_n \quad , \tag{15}$$

Quindi applicando la potenza ennesima di A a v_0 si ottiene:

$$\begin{aligned} A^k\mathbf{v}_0 &= \alpha_1 A^k v_1 + \cdots + \alpha_n A^k v_n && (16)\\ &= \alpha_1\lambda_1^k v_1 + \cdots + \alpha_n\lambda_n^k v_n && (17)\\ &= \lambda_1^k\left(\alpha_1 v_1 + \cdots + \alpha_n\left(\frac{\lambda_n}{\lambda_1}\right)^k v_n\right) \quad . && (18) \end{aligned}$$

Quindi se gli autovalori sono tutti diversi e sono ordinati in ordine decrescente $\lambda_1 > \cdots > \lambda_n$ allora tutti i termini dentro la parentesi tendono a zero per k che va all'infinito in quanto minori di uno. Questo metodo converge all'autovalore maggiore della matrice. Se volessimo trovarli tutti possiamo sempre usare questo metodo ma integrarlo con un metodo di ortogonalizzazione ad esempio Gram–Schmidt che chiamiamo ad ogni passo. Come criteri di convergenza possiamo mettere o la distanza tra iterazione successive dell'autovettore oppure la radice del valore assoluto della differenza tra iterazione successive dell'autovalore. Usiamo la radice perché la convergenza è quadratica negli autovalori:

$$\begin{aligned} R_{1,2} &= ||\mathbf{v}_{k+1} \pm \mathbf{v}_k|| \\ R_3 &= \sqrt{|\lambda_{k+1} - \lambda_k|} \quad , \end{aligned} \tag{19}$$

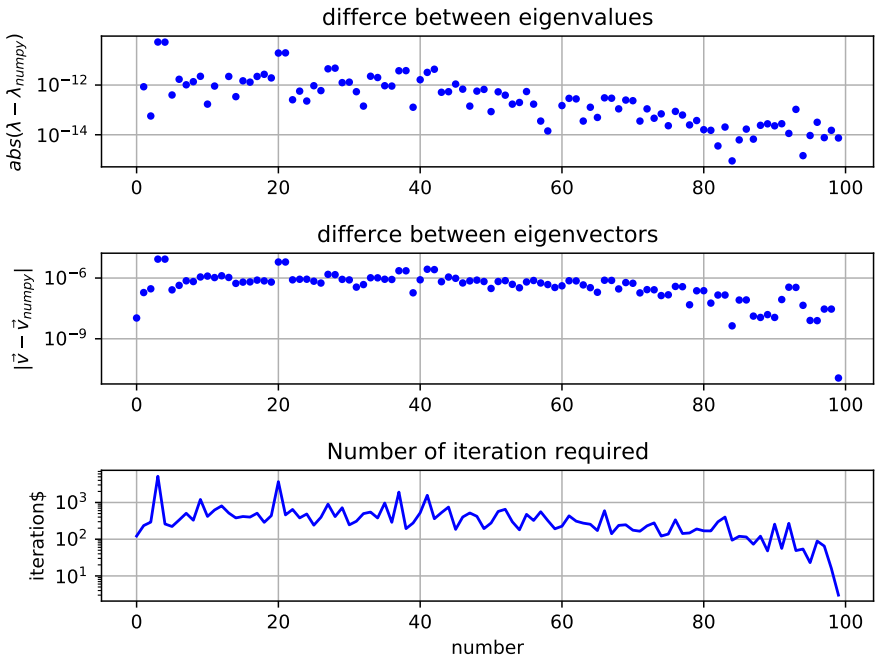
dove il $\pm$ deriva dal fatto che sia $+\mathbf{v}$ che $-\mathbf{v}$ sono autovettori. Quindi quando una di queste quantità è minore di una certa tolleranza che scegliamo noi l'algoritmo termina.
Capita sovente però che magari non si sia interessati a tutti gli autovalori, magari solo ai più grandi o a i più piccoli. Per i più grandi possiamo applicare l'algoritmo così come lo abbiamo descritto. Ma se fossimo interessati ai più piccoli? Per questo secondo basta sostituire A con la sua inversa in quanto gli autovalori di A^{-1} sono il reciproco degli autovalori di A quindi il più grande autovalore di A^{-1} sarà il più piccolo di A, proprio come volevamo; questo si chiama metodo delle potenze inverso. In genere questo in questo metodo non si inverte direttamente A; si usa invece una routine per risolvere un sistema. Qui è stato deciso invece di invertire direttamente la matrice di partenza senza farci troppi problemi. Vediamo ora il codice:

```
import numpy as np

def eig(M, k=None, tol=1e-10, magnitude='small'):
    '''
    Compute the eigenvalue decomposition
    of the symmetric matrix A using power iteration
    or inverse iteration.
    Inverse iteration is the same of power iteration
    but we use M^-1 instead of M so the eigenvalues
    are the reciprocal.

    Parameters
    ----------
    M : 2darray
        N x N matrix, symmetric
    k : None or int, if None k=N
```

```
        number of eigenvariates and eigenvectors to find,
        if k<N then k eigenvectors corresponding to the k
        largest eigenvalues will be found
    tol : float, optional default 1e-10
        required tollerance
    magnitude : string, optional, default small
        if magnitude == 'small' the smallest eigenvalues
        and thei relative eigenvectors will be computed
        if magnitude == 'big' the biggest eigenvalues
        and thei relative eigenvectors will be computed

    Return
    ------
    eigval : 1darray
        array of eigenvalues
    eigvec : 2darray
        kxk matrix, the column eigvec[:, i] is the
        ormalized eigenvector corresponding to the
        eigenvalue eigval[i]
    counts : 1darray
        how many iteration are made for each eigenvector
    '''

    if magnitude == 'small':
        A = np.copy(np.linalg.inv(M))
    if magnitude == 'big':
        A = np.copy(M)

    N = A.shape[0]
    if k is None:
        k = N

    eigvec = []  # will contain the eignvectors
    eigval = []  # will contain the eignvalues
    counts = []  # will contain the numbero of iteration of each eigenvalue

    for _ in range(k):

        v_p = np.random.randn(N) #initial vector
        v_p = v_p / np.sqrt(sum(v_p**2))
        l_v = np.random.random()
        Iter= 0

        while True:
            l_o = l_v
            v_o = v_p                           # update vector
            v_p = np.dot(A, v_p)                # compute new vector
            v_p /= np.sqrt(sum(v_p**2))         # normalization

            # Orthogonalization respect
            # all eigenvectors find previously
            for i in range(len(eigvec)):
                v_p = v_p - np.dot(eigvec[i], v_p) * eigvec[i]

            #eigenvalue of v_p, A @ v_p = l_v * v_p
            #multiplying by the transposed => (A @ v_p) @ v_p.T = l_v
            #using v_p @ v_p.T = 1
            l_v = np.dot(np.dot(A, v_p), v_p)

            R1 = np.sqrt(sum((v_p - v_o)**2))
            R2 = np.sqrt(sum((v_o + v_p)**2))
            R3 = np.sqrt(abs(l_v - l_o))      # In eigenvalues the convergence is quadratic

            Iter += 1
            if R1 < tol or R2 < tol or R3 < tol:
                break

        eigvec.append(v_p)
        eigval.append(l_v)
        counts.append(Iter)

    if magnitude == 'small':
        eigval = 1/np.array(eigval)
        eigvec = np.array(eigvec).T
    if magnitude == 'big':
        eigvec = np.array(eigvec).T
```

```
        eigval = np.array(eigval)

    return eigvec, eigval, counts
```

Vediamo ora i risultati due test del nostro algoritmo. Cominciamo generando una matrice random grazie a numpy e per averla simmetrica consideriamone il prodotto per se stessa trasposta: "P = np.random.normal(size=[n, n]) H = np.dot(P.T, P)"; quindi diagonalizziamo H, prendiamo $n = 100$, settando "magnitude='big'". Mostriamo solo il risultato, il codice lo troverete scritto nella cartella, confrontando il risultato con la diagonalizzazione fatta da numpy:



Dal punto di vista del tempo chiaramente numpy nemmeno fa fatica: 14.240 secondi noi contro 0.001 di numpy, e ci guarda pure con aria di superiorità perché gli facciamo schifo. Comunque il risultato è soddisfacente.

10.2 Equazione di Schrödinger

Vediamo ora un test un po' più fisico dove siamo interessati agli autovalori più piccoli. Consideriamo una matrice 'a bischero' fatta del tipo:

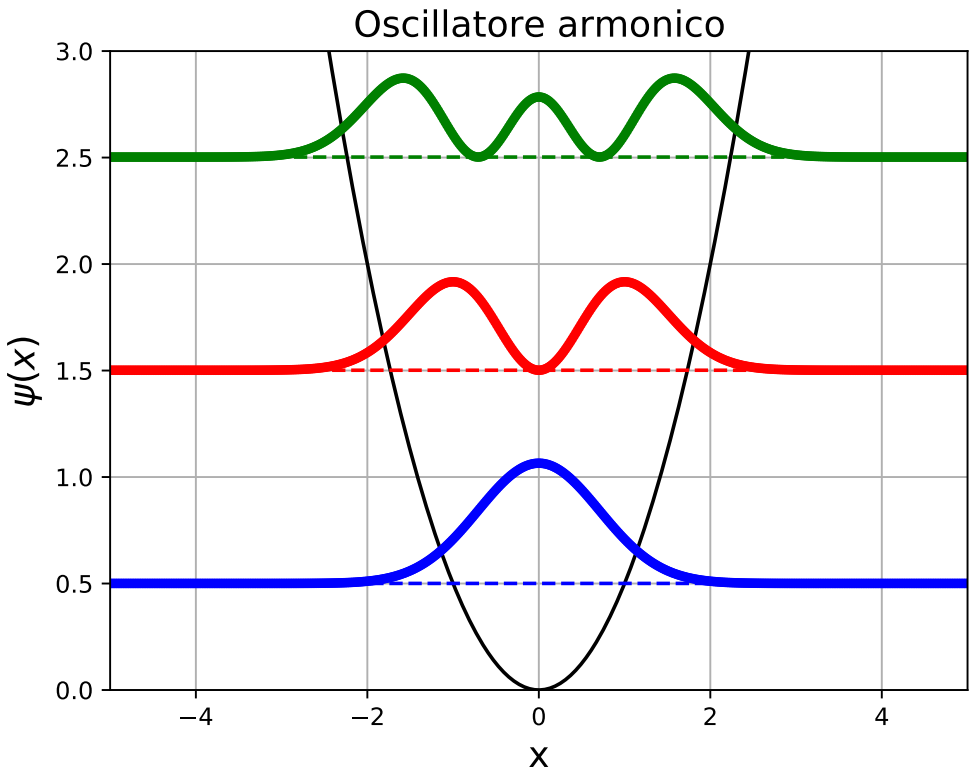
$$H = -\frac{1}{2h^2}\begin{pmatrix} -2 & 1 & 0 & \cdots & 0 \\ 1 & -2 & 1 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & 1 & -2 & 1 \\ 0 & \cdots & 0 & 1 & -2 \end{pmatrix} + \begin{pmatrix} V(x_1) & 0 & 0 & \cdots & 0 \\ 0 & V(x_2) & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & 0 & V(x_{N-1}) & 0 \\ 0 & \cdots & 0 & 0 & V(x_N) \end{pmatrix} \tag{20}$$

dove h è la spaziatura tra x_i e x_{i+1} il quale è un array in un certo range in un certo numero di punti. Probabilmente avrete notato che la matrice scritta sopra non è altro che la discretizzazione dell'equazione di Schrödinger (non stiamo qui a ricavarla, lo vedrete nei corsi di meccanica quantistica, prendetela per buona per il momento) non dipendente dal tempo :

$$H\psi = \left(-\frac{1}{2}\frac{\partial^2}{\partial x^2} + V(x) \right)\psi = E\psi \tag{21}$$

Qui chiaramente siamo interessati solo ai livelli energetici minori in quanto l'approssimazione del laplaciano che abbiamo fatto peggiora man mano che gli stati sono sempre più estesi, discretizzare così infatti significa mettersi dentro una scatola di lato fissato ma chiaramente noi vorremmo la scatola infinita, quindi per i livelli più bassi l'approssimazione è buona ($h = L/N$ nel nostro caso abbiamo preso L=20 e N=1000). Per semplicità consideriamo il caso dell'oscillatore armonico $V(x) = x^2/2$ e troviamo i dieci autovalori più bassi (analiticamente e in unità naturali $E = n + 1/2$). Grafichiamo anche poi per bellezza tre autovettori, o autofunzioni, con il caveat che ogni autovettore va diviso per $\sqrt{h}$. Come sopra il codice è già scritto e lo trovate tutto insieme, noi qui mostriamo solo i risultati:

teorico	calcolato	errore
0.5	0.50049	4.88e-04
1.5	1.50144	1.44e-03
2.5	2.50234	2.34e-03
3.5	3.50319	3.19e-03
4.5	4.50399	3.99e-03
5.5	5.50474	4.74e-03
6.5	6.50544	5.44e-03
7.5	7.50609	6.09e-03
8.5	8.50669	6.69e-03
9.5	9.50724	7.24e-03



Per un totale di tempo di esecuzione di 0.77814 secondi.

10.3 Algoritmo QR

Vediamo adesso un altro algoritmo che può essere interessante trattare. Come si può intuire dal nome questo algoritmo si basa sulla scomposizione QR di una matrice, dove Q è una matrice ortogonale $Q^T = Q^{-1}$ e R è una matrice triangolare superiore. Data quindi la nostra matrice A da diagonalizzare quello che si fa è, chiamando $A_0 = A = Q_0 R_0$:

$$A_{k+1} = R_k Q_k = Q_k^{-1} Q_k R_k Q_k = Q_k^T A_k Q_k \quad . \tag{22}$$

Vedete quindi che procedendo per trasformazioni ortogonali tutte le A_k sono simili. Avremo quindi che gli autovalori saranno gli elementi sulla diagonale della matrice A_{k+1}. Mentre per gli autovettori quello che si fa è calcolare il prodotto delle varie Q_k con loro stesse:

$$V = Q_0 Q_1 \dots Q_k \quad , \tag{23}$$

V sarà la matrice che conterrà gli autovettori di A. Tutto ciò viene eseguito un certo tot di volte che scegliamo noi da input. Per completare la spiegazione andiamo a vedere come si esegue la decomposizione QR. Fondamentalmente si tratta di applicare Gram-Schmidt alle colonne di A.

$$A = [\mathbf{a}_1 | \mathbf{a}_2 | \dots | \mathbf{a}_N] \tag{24}$$

$$
\begin{aligned}
\mathbf{v}_1 &= \mathbf{a}_1, & \mathbf{e}_1 &= \frac{\mathbf{v}_1}{\|\mathbf{v}_1\|} \\
\mathbf{v}_2 &= \mathbf{a}_2 - (\mathbf{a}_2 \cdot \mathbf{e}_1)\mathbf{e}_1, & \mathbf{e}_2 &= \frac{\mathbf{v}_2}{\|\mathbf{v}_2\|} \\
&\vdots & &\vdots \\
\mathbf{v}_k &= \mathbf{a}_k - \sum_{j=1}^{k-1} (\mathbf{a}_k \cdot \mathbf{e}_j)\mathbf{e}_j, & \mathbf{e}_k &= \frac{\mathbf{u}_k}{\|\mathbf{u}_k\|}.
\end{aligned}
$$

Dunque possiamo arrivare a dire che:

$$
A = [\mathbf{e}_1|\mathbf{e}_2|\dots|\mathbf{e}_N] \begin{pmatrix} \mathbf{a}_1 \cdot \mathbf{e}_1 & \mathbf{a}_2 \cdot \mathbf{e}_1 & \cdots & \mathbf{a}_N \cdot \mathbf{e}_1 \\ 0 & \mathbf{a}_2 \cdot \mathbf{e}_2 & \cdots & \mathbf{a}_N \cdot \mathbf{e}_2 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \mathbf{a}_N \cdot \mathbf{e}_N \end{pmatrix} = QR \quad . \tag{25}
$$

Questo metodo, a differenza del precedente, con il quale potevamo scegliere quanti autovalori e autovettori farci calcolare, restituisce tutti gli autovalori e gli autovettori della nostra matrice. Siamo in Python, probabilmente per matrici grandi ci vorrà tanto. Passiamo quindi adesso al codice:

```
#=====================================================================
# Function to compute QR decomposition
#=====================================================================

def QR_decomp(A):
    '''
    Comupute QR decomposition of a matrix A

    Parameters
    ----------
    A : 2darray
        N x N matrix

    Returns
    -------
    Q, R : 2darray
        A = Q @ R
    '''

    # we give different names because in principle the QR
    # decomposition also applies to non-square matrices
    n, m = A.shape

    Q = np.zeros((n, n)) # Initialize matrix Q
    R = np.zeros((n, m)) # Initialize matrix R
    v = np.zeros((n, n)) # Initialize matrix v: for Gram-Schmidt

    u[:, 0] = A[:, 0]
    Q[:, 0] = v[:, 0] / np.sqrt(sum(v[:, 0]**2))

    for i in range(1, n):

        v[:, i] = A[:, i]
        for j in range(i):
            v[:, i] -= (A[:, i] @ Q[:, j]) * Q[:, j]

        Q[:, i] = v[:, i] / np.sqrt(sum(v[:, i]**2))


    for i in range(n):
        for j in range(i, m):
            R[i, j] = A[:, j] @ Q[:, i]

    # uncomment for scipy comparison
    #D = np.diag(np.sign(np.diag(Q)))
    #Q[:, :] = Q @ D
    #R[:, :] = D @ R

    return Q, R

```

```
#=====================================================================
# Function to solve eigensystem via QR iterartion
#=====================================================================

def QR_eig(A, maxiter=100):
    '''
    Find the eigenvalues of A using QR iteration.

    Parameters
    ----------
    A : 2darray
        N x N matrix
    maxiter : int, optional, default 100
        number of iterations to do

    Returns
    -------
    eigval : 1darray
        array of eigenvalues
    eigvec : 2darray
        N x N matrix, the column eigvec[:, i] is the
        ormalized eigenvector corresponding to the
        eigenvalue eigval[i]
    '''
    A_new, A_old = [np.copy(A)]*2

    eigvec = np.eye(A.shape[0])

    for i in range(maxiter):

        A_old[:, :] = A_new
        Q, R = QR_decomp(A_old)

        A_new[:, :] = R @ Q
        eigvec = eigvec @ Q

    eigval = np.diag(A_new)

    return eigval, eigvec
```

Facendo la prova con la stessa matrice a bischero di prima infatti otteniamo un tempo di 717.6 secondi, usando le iterazioni di default. I risultati sono gli stessi che quelli mostrati in precedenza. Attenzione ora però all'ordine degli autovalori, dovrebbero essere ordinati dal più grande al più piccolo.

10.4 Lanczos

Un'ultima cosa carina da spiegare è l'iterazione di Lanczos, caso particolare della più generale iterazione di Arnoldi; ma per i casi di interesse fisico le ipotesi sono sempre ragionevoli: la matrice da diagonalizzare deve essere hermitiana. L'idea è di trovare due matrici per cui valga:

$$A = Q^\dagger H Q. \tag{26}$$

H sarà un amatrice tridiagonale simmetrica reale e sarà quella che andremo a diagonalizzare. Essa sarà una matrice più piccola rispetto ad A, la dimensione è un parametro che scegliamo noi, quindi verrà più leggero il conto, ma non siamo in grado di recuperare tutti gli autovalori, e autovettori di A. L'unico modo per farlo è imporre che H abbia la stessa dimensione di A. Q invece sarà una certa matrice con colonne ortonormali; se poi T ha la stessa dimensione di A allora Q sarà unitaria. Tornando a noi diagonalizzando H avremo un certo numero di autovalori, che sono anche autovalori di A, non necessariamente tutti sequenziali. Per gli autovettori invece, se vale $H\mathbf{y} = \lambda\mathbf{y}$ allora Qy è autovalore di A. Vediamo subito il codice:

```
def lanczos(A, n):
    '''
    Lanczos iteration for a matrix A

    Parameter
    ---------
    A : 2darray
        Hermitian N x N matrix
    n : int
        dimension of Krylov subspace
        e.g. dimension of H

    Return
```

```
    ------
    Q, H : 2darray
        A = Q.T H Q
    '''
    m     = A.shape[0]
    Q     = np.zeros((m, n+1))
    alpha = np.zeros(n)
    beta  = np.zeros(n)
    b     = np.random.randn(m)

    Q[:,0] = b/np.sqrt(sum(b**2))

    for i in range(n):
        v        = np.dot(A, Q[:,i])
        alpha[i] = np.dot(Q[:,i], v)

        if i == 0:
            v = v - alpha[i] * Q[:, i]
        else :
            v = v - beta[i-1] * Q[:, i-1] - alpha[i] * Q[:, i]

        beta[i]  = np.sqrt(sum(v**2))
        Q[:,i+1] = v / beta[i]

    H = Q.T @ A @ Q

    return Q, H
```

Calcolata H la passiamo quindi a una delle funzioni sopra scritte e vediamo che succede. Per riuscire a prendere dei buoni risultati per i primi livelli eccitati usiamo come $n = 300$, quindi l'algoritmo QR ora sarà un po' più spiccio. Vediamo cosa esce sempre per il caso dell'oscillatore armonico.

teorico	QR	errore	potenze	errore
0.5	0.50068	6.80e-04	0.50068	6.80e-04
1.5	1.50146	1.46e-03	1.50145	1.45e-03
2.5	2.50321	3.21e-03	2.50257	2.57e-03
3.5	3.59515	9.51e-02	3.59459	9.46e-02
4.5	4.61829	1.18e-01	4.61261	1.13e-01
5.5	6.60135	1.10e+00	6.59104	1.09e+00
6.5	7.76801	1.27e+00	7.74208	1.24e+00
7.5	10.35934	2.86e+00	10.36662	2.87e+00
8.5	12.02403	3.52e+00	12.04071	3.54e+00
9.5	13.48530	3.99e+00	13.49227	3.99e+00

Vedete quindi come per i primi 5 livelli vada tutto bene, poi gli altri autovalori sono diversi o un po' sbagliati: ad esempio 10.35 è un po' lontano mentre 13.49 è già un migliore risultato. Come dicevamo prima vedete poi che ci sono degli autovalori che sono spariti in quanto H è solo 300×300. Dal punto di vista dei tempi QR impiega circa 48 secondi, mentre il metodo delle potenze circa 0.1 secondi.