ESERCITAZIONI DI ANALISI NUMERICA *

A. SOMMARIVA † E M.R. RUSSO ‡

1. Alcuni problemi della teoria dell'approssimazione.

1.1. Costanti di Lebesgue. Dato un set di punti distinti $X = \{x_0, \ldots, x_n\}$ nell'intervallo chiuso e limitato $[a, b]$ si definisce *costante di Lebesgue* la quantità

$$\Lambda_n := \max_{x \in [a,b]} \sum_{i=0}^{n} |L_i(x)|$$

con al solito L_i *i-simo* polinomio di Lagrange (relativamente al set $x_0, \ldots, x_n$). Sia ora $\phi_0, \ldots, \phi_n$ una base per lo spazio $\mathbb{P}_n$ dei polinomi di grado n. Sia $V_n(X)$ la matrice di Vandermonde del set di punti X rispetto alla base $\phi_0, \ldots, \phi_n$. Sia $Y \subset [a, b]$ un sottoinsieme di discreto con cardinalità finita. Se `VX` e `VY` sono rispettivamente $V_n(X)$ e $V_n(Y)$ allora, se l'insieme Y è sufficientemente *fitto*, una buona stima della costante di Lebesgue è data da

```
leb_const=norm(VX'\VY',1);
```

D'altra parte un esempio di base è

```
function V = chebvand(deg,mymesh,intv)

% computes the Chebyshev-Vandermonde matrix at a 1d mesh
% in Chebyshev basis of a rectangle containing
% the mesh points

% INPUT:

% deg = polynomial degree;
% mymesh = 1-columns array of mesh points coordinates;
% intv = interval that contains the mesh.

%OUTPUT:

% V = Chebyshev-Vandermonde matrix

%FUNCTION BODY

if nargin < 3
    intv=[-1 1];
end

% CHEBYSHEV BASIS. SCALED.
a=intv(1); b=intv(2);
xx=(2*mymesh-a-b)/(b-a); % SCALING.
```

*Ultima revisione: 1 marzo 2011

†DIPARTIMENTO DI MATEMATICA PURA ED APPLICATA, UNIVERSITÀ DEGLI STUDI DI PADOVA, VIA TRIESTE 63, 35121 PADOVA, ITALIA (ALVISE@MATH.UNIPD.IT)

‡DIPARTIMENTO DI MATEMATICA PURA ED APPLICATA, UNIVERSITÀ DEGLI STUDI DI PADOVA, VIA TRIESTE 63, 35121 PADOVA, ITALIA (MRRUSSO@MATH.UNIPD.IT)

```
for index = 1:(deg+1)
    V(:,index)=cos((index-1)*acos(xx)); % VANDERMONDE.
end
```

Si domanda:

1. definiti per $m = n + 1$ i nodi equispaziati nell'intervallo $[-1, 1]$ che determinano l'insieme X e i nodi test
$$Y = \{-1 + hk\},\ k = 0, \ldots, 1000, h = 1/500$$
calcolare approssimativamente la costante di Lebesgue per $n = 3, 4, 5, \ldots, 10$.
2. definiti per $m = n + 1$ i nodi di Chebyshev
$$X = \{\cos \frac{(2k + 1)\pi}{2m}\},\ k = 0, \ldots, m - 1$$
e i nodi test
$$Y = \{-1 + hk\},\ k = 0, \ldots, 1000, h = 1/500$$
calcolare approssimativamente la costante di Lebesgue per $n = 3, 4, 5, \ldots, 10$.

1.2. Miglior approssimante polinomiale e interpolante polinomiale nei nodi di Chebyshev. Il metodo di Remez permette di calcolare il polinomio di miglior approssimazione di grado n di una funzione continua f in un intervallo chiuso e limitato $[a, b]$. Esistono varie implementazioni di tale algoritmo. Utilizzando la versione descritta in

```
R. Pachon e L.N. Trefethen,
"Barycentric-Remez algorithms for best polynomial approximation
 in the chebfun system",
BIT Numer Math (2009) 49, p. 721741,
```

è possibile calcolare, fissato un grado n il polinomio di miglior approssimazione $p^* \in \mathbb{P}_1 0$.

Vediamo alcuni esempi. Si consideri la funzione $f(x) = \sin(\exp(x))$, $x \in [-1, 1]$ citata in tale pubblicazione a pagina 734. Se il sistema `chebfun` è stato installato nella versione Matlab (per il download si veda la homepage degli autori), il codice

```
% CHEBFUN DI UNA FUNZIONE.
f=chebfun('sin(exp(x))');

% POLINOMIO DI MIGLIOR APPROSSIMAZIONE p (DI GRADO 10)
% ED ERRORE COMPIUTO err.
[p,err]=remez(f,10);

% POLINOMIO p DESCRITTO NELLA BASE MONOMIALE
% pp(1)*x^N + pp(2) * x^{N-1} + ...
pp=poly(p);

% PUNTI TEST.
xx=(-1:0.001:1)';

% VALUTAZIONE DEL POLINOMIO NEI PUNTI TEST.
pxx=polyval(pp,xx);
```

```
% VALUTAZIONE DELLA FUNZIONE NEI PUNTI TEST.
fxx=feval(f,xx);

% ERRORE COMPIUTO.
err2=norm(fxx-pxx,inf);

pp'
err2
```

Si vede che l'errore ottenuto (nella norma 2 nel continuo, cioè $\int_a^b |f(x)|^2 \, dx$) è approssimativamente $1.7862e-06$ e che il polinomio p_{10}^* di miglior approssimazione ha coefficienti rispetto alla base monomiale (il primo è quello di grado massimo!)

```
 0.003196437149154
 0.017324847917382
 0.033716160802325
 0.027736791074028
-0.020560308792805
-0.139174865843104
-0.321930004523161
-0.420608521955273
-0.150685513280442
 0.540295397586175
 0.841472656602631
```

Si desidera

1. usando i comandi `polyfit` e `polyval`, valutare nei punti definiti dalla mesh $x = -1 : 0.001 : 1$ il polinomio di grado 10 che interpola la funzione

$$f(x) = \sin(\exp(x))$$

   nei nodi di Chebyshev

$$\cos\frac{(2k+1)\pi}{2m}, \ k = 0, \ldots, m-1.$$

   Se vogliamo calcolare il polinomio interpolante di grado 10, che valore deve assumere m?
2. calcolare una stima dell'errore in norma infinito compiuto da p nell'approssimare f, come

$$\max_{x=-1:0.001:1} |f(x) - p(x)|$$

3. plottare il polinomio di miglior approssimazione p_{10}^* nei punti definiti dalla mesh $x = -1 : 0.001 : 1$;
4. plottare in scala semilogaritmica, nei punti della mesh $x = -1 : 0.001 : 1$, la funzione $|f - p_{10}^*|$;

2. Quadratura numerica.

2.1. Alcune formule di quadratura.

2.1.1. Esempi formule composte: trapezi e Cavalieri-Simpson. Tra i metodi comunemente utilizzati per approssimare una funzione continua f in un intervallo chiuso e limitato $[a, b]$, uno dei più semplici è la formula composta dei trapezi e di Cavalieri-Simpson. Si suddivida l'intervallo (chiuso e limitato) $[a, b]$ in N subintervalli $T_j = [x_j, x_{j+1}]$ tali che $x_j = a + jh$ con $h = (b - a)/N$. Dalle proprietà dell'integrale

$$\int_a^b f(x)\,dx = \sum_{j=0}^{N-1} \int_{x_j}^{x_{j+1}} f(x)\,dx \approx \sum_{j=0}^{N-1} S(f, x_j, x_{j+1}) \tag{2.1}$$

dove S è una delle regole di quadratura finora esposte (ad esempio $S_3(f)$). Le formule descritte in (2.1) sono dette *composte*. Due casi particolari sono

1. *formula composta dei trapezi*

$$S_1^{(c)} := h\left[\frac{f(x_0)}{2} + f(x_1) + \ldots + f(x_{N-1}) + \frac{f(x_N)}{2}\right] \tag{2.2}$$

il cui errore è

$$E_1^{(c)}(f) := I(f) - S_1^{(c)}(f) = \frac{-(b-a)}{12} h^2\, f^{(2)}(\xi),\;\; h = \frac{(b-a)}{N} \tag{2.3}$$

per qualche $\xi \in (a, b)$;

2. *formula composta di Cavalieri-Simpson* fissati il numero N di subintervalli e i punti $x_k = a + kh/2$ dove $h = \frac{b-a}{N}$ sia

$$I(f) \approx S_3^{(c)}(f) := \frac{h}{6}\left[f(x_0) + 2\sum_{r=1}^{N-1} f(x_{2r}) + 4\sum_{s=0}^{N-1} f(x_{2s+1}) + f(x_{2N})\right]; \tag{2.4}$$

il cui errore è

$$E_3^{(c)}(f) := I(f) - S_3^{(c)}(f) = \frac{-(b-a)}{180}\left(\frac{h}{2}\right)^4 f^{(4)}(\xi) \tag{2.5}$$

per qualche $\xi \in (a, b)$.

2.1.2. Formule gaussiane. Ricordiamo ora che una formula

$$\int_a^b f(x)w(x)dx \approx \sum_{i=1}^{M} w_i f(x_i)$$

ha grado di precisione *almeno* N se e solo se è esatta per tutti i polinomi f di grado inferiore o uguale a N. Ha inoltre grado di precisione N se e solo se è esatta per ogni polinomio di grado N ed esiste un polinomio di grado $N + 1$ per cui non lo sia. Nelle formule interpolatorie di Newton-Cotes (come ad esempio la regola del Trapezio o di Cavalieri-Simpson che sono alla base delle formule composte precedentemente citate) i nodi $x_1, \ldots, x_n$ sono equispaziati e il grado di precisione δ è generalmente uguale almeno a $n - 1$ ma in alcuni casi, come per la regola di Cavalieri-Simpson, uguale al numero di nodi n. Vediamo ora formule che a parità di nodi hanno grado di precisione maggiore di n.

Sia $w : (a, b) \to \mathbb{R}$ (non necessariamente limitato) è una funzione peso, cioè tale che

1. w è nonnegativa in (a, b);
2. w è integrabile in $[a, b]$;
3. esista e sia finito

$$\int_a^b |x|^n w(x)\, dx$$

per ogni $n \in \mathbb{N}$;
4. se

$$\int_a^b g(x) w(x)\, dx = 0$$

per una qualche funzione nonnegativa g allora $g \equiv 0$ in (a, b).

Tra gli esempi più noti ricordiamo

1. *Legendre*: $w(x) \equiv 1$ in $[a, b]$ limitato;
2. *Jacobi*: $w(x) = (1 - x)^{\alpha} (1 + x)^{\beta}$ in $(-1, 1)$ per $\alpha, \beta \geq -1$;
3. *Chebyshev*: $w(x) = \frac{1}{\sqrt{1-x^2}}$ in $(-1, 1)$;
4. *Laguerre*: $w(x) = \exp(-x)$ in $[0, \infty)$;
5. *Hermite*: $w(x) = \exp(-x^2)$ in $(-\infty, \infty)$;

Si dimostra che

TEOREMA 2.1. *Per ogni $n \geq 1$ esistono e sono unici dei nodi $x_1, \ldots, x_n$ e pesi $w_1, \ldots, w_n$ per cui il grado di precisione della formula di quadratura*

$$\int_a^b g(x)\, w(x)\, dx \approx \sum_{i=1}^{n} w_i g(x_i)$$

sia almeno $2n - 1$. I nodi sono gli zeri del polinomio ortogonale di grado n,

$$\phi_n(x) = A_n \cdot (x - x_1) \cdot \ldots \cdot (x - x_n)$$

e i corrispettivi pesi sono

$$w_i = \int_a^b L_i(x) w(x) dx,\ i = 1, \ldots, n.$$

In generale il calcolo dei nodi e dei pesi non è cosa banale e rimandiamo questo dettaglio alla sezione successiva.

2.2. Implementazione delle formule di quadratura. Per quanto riguarda la formula dei trapezi, una sua implementazione in Matlab/Octave è data da

```
function [x,w]=trapezi_composta(N,a,b)

% FORMULA DEI TRAPEZI COMPOSTA.

% INPUT:
% N: NUMERO SUBINTERVALLI.
% a, b: ESTREMI DI INTEGRAZIONE.
```

```
% OUTPUT:
% x: NODI INTEGRAZIONE.
% w: PESI INTEGRAZIONE (INCLUDE IL PASSO!).

h=(b-a)/N;                % PASSO INTEGRAZIONE.
x=a:h:b; x=x';            % NODI INTEGRAZIONE.
w=ones(N+1,1);            % PESI INTEGRAZIONE.
w(1)=0.5; w(N+1)=0.5;
w=w*h;
```

che fornisce i nodi di quadratura x e i corrispettivi pesi w come due vettori colonna.

Similmente, per quanto riguarda la formula di Cavalieri-Simpson composta, i nodi di quadratura x e i corrispettivi pesi w si ottengono dalla routine

```
function [x,w]=simpson_composta(N,a,b)

% FORMULA DI SIMPSON COMPOSTA.

% INPUT:
% N: NUMERO SUBINTERVALLI.
% a, b: ESTREMI DI INTEGRAZIONE.

% OUTPUT:
% x: INTEGRAZIONE.
% w: PESI INTEGRAZIONE (INCLUDE IL PASSO!).

h=(b-a)/N;                % AMPIEZZA INTERVALLO.
x=a:(h/2):b; x=x';        % NODI INTEGRAZIONE.

w=ones(2*N+1,1);              % PESI INTEGRAZIONE.
w(3:2:2*N-1,1)=2*ones(length(3:2:2*N-1),1);
w(2:2:2*N,1)=4*ones(length(2:2:2*N),1);
w=w*h/6;
```

Osserviamo che

- in Matlab si può definire una funzione (matematica) f come *inline* e che può essere valutata con il comando *feval* (se necessario aiutarsi con l'help di Matlab/Octave), ad esempio

```
>> f=inline('sin(x)'); % DEFINIZIONE COME INLINE DI UNA FUNZIONE.
>> feval(f,0)              % VALUTAZIONE DELLA FUNZIONE.
ans =
     0
>>
```

  (si noti che la funzione è passata come stringa, visti gli apici);
- una volta noti il vettore (colonna) x dei nodi e w dei pesi di integrazione, se la funzione f è richiamata da un m-file *f.m*, basta

```
fx=feval(f,x);                   % VALUT. FUNZIONE.
I=w'*fx;                   % VALORE INTEGRALE.
```

per calcolare il risultato fornito dalla formula di quadratura composta.

Per quanto concerne le formule gaussiane, esponiamo di seguito una routine eseguita da D. Laurie e W. Gautschi, che fissato un numero naturale positivo N calcola una formula gaussiana rispetto alla funzione peso di Jacobi

$$w(x) = (1-x)^{\alpha}(1+x)^{\beta},\ x \in (-1, 1).$$

Osserviamo che per $\alpha = \beta = 0$, la funzione peso coincide con $w \equiv 1$.

```
function [x,w]=gauss_jacobi(N,alpha,beta)

% GAUSS-JACOBI RULE ON [-1,1] (OR (-1,1) DEPENDING ON alpha, beta).
% N: FOR GAUSSIAN RULES IT IS THE NUMBER OF QUAD. POINTS.
% alpha, beta ARE THE GAUSS-JACOBI EXPONENTS.
% x, w ARE COLUMN VECTORS OF NODES AND WEIGHTS.
%      THE LENGTH OF x AND w IS "N" IF gl=0, "N+2" IF "gl=1".

if nargin < 2
    alpha=0; beta=0;
end

ab=r_jacobi(N,alpha,beta);
xw=gauss(N,ab);

x=xw(:,1);
w=xw(:,2);

%----------------------------------------------------------------------------
% ADDITIONAL FUNCTIONS BY D.LAURIE AND W.GAUTSCHI.
%----------------------------------------------------------------------------

function ab=r_jacobi(N,a,b)

nu=(b-a)/(a+b+2);
mu=2^(a+b+1)*gamma(a+1)*gamma(b+1)/gamma(a+b+2);
if N==1
 ab=[nu mu]; return
end

N=N-1;
n=1:N;
nab=2*n+a+b;
nuadd=(b^2-a^2)*ones(1,N)./(nab.*(nab+2));
A=[nu nuadd];
n=2:N;
nab=nab(n);
B1=4*(a+1)*(b+1)/((a+b+2)^2*(a+b+3));
B=4*(n+a).*(n+b).*n.*(n+a+b)./((nab.^2).*(nab+1).*(nab-1));
abadd=[mu; B1; B'];
ab=[A' abadd];


function xw=gauss(N,ab)
N0=size(ab,1); if N0<N, error('input array ab too short'), end
```

```
J=zeros(N);
for n=1:N, J(n,n)=ab(n,1); end
for n=2:N
  J(n,n-1)=sqrt(ab(n,2));
  J(n-1,n)=J(n,n-1);
end
[V,D]=eig(J);
[D,I]=sort(diag(D));
V=V(:,I);
xw=[D ab(1,2)*V(1,:)'.^2];
```

2.3. Esempio. Vediamo di seguito un esempio su come calcolare con i codici visti un integrale di una funzione (che scegliamo regolare).

Dapprima vediamo qual'è il risultato corretto

```
>> % CALCOLO SIMBOLICO IN MATLAB.
>> f=inline('1/(1+x.^2)'); % DEF. FUNZIONE.
>> syms x
>> int(1./(1+x.^2),-1,1) % INTEGRALE DEF. DA -1 A 1.

ans =

pi/2
```

e di seguito eseguiamo alcuni test numerici

```
>> f=inline('1/(1+x.^2)'); % DEF. FUNZIONE.
>> % CALCOLO INTEGRALE FORMULA TRAPEZI COMPOSTA
>> [x,w]=trapezi_composta(5,-1,1);
>> length(x)

ans =

     6

>> fx=feval(f,x);
>> I_tpz=w'*fx

I_tpz =

     1.557466063348416e+00

>> err_tpz=abs(I_tpz-pi/2)

err_tpz =

     1.333026344648047e-02

>> % CALCOLO INTEGRALE FORMULA COMPOSTA SIMPSON.
>> [x,w]=simpson_composta(5,-1,1);
>> fx=feval(f,x);
>> I_simpson=w'*fx;
>> format long e
```

```
>> pi/2

ans =

     1.570796326794897e+00

>> I_simpson

     1.570795388091188e+00

>> err_simpson=abs(I_simpson-pi/2)

err_simpson =

     9.387037087638106e-07

>> length(x)  % PUNTI FORMULA COMPOSTA

ans =

    11

>> % FORMULE GAUSSIANE.
>> [x,w]=gauss_jacobi(5,0,0); % NODI [-1,1]!!!
>> fx=feval(f,x);
>> length(x)

ans =

     5

>> I_gauss=w'*fx

I_gauss =

     1.571171171171171e+00

>> err_gauss=abs(I_gauss-pi/2)

err_gauss =

     3.748443762745524e-04

>> % A PARITA' DI NODI TRAPEZI COMPOSTA OFFRIVA
>> % UN ERRORE DI 1.333026344648047e-02.
>>
>> [x,w]=gauss_jacobi(11,0,0); % NODI [-1,1]!!!
>> fx=feval(f,x);
>> I_gauss=w'*fx

I_gauss =

     1.570796336515167e+00
```

```
>> err_gauss=abs(I_gauss-pi/2)

err_gauss =

     9.720270366386785e-09

>> % A PARITA' DI NODI SIMPSON COMPOSTA OFFRIVA
>> % UN ERRORE DI 9.387037087638106e-07.
```

Si noti che in effetti il prodotto scalare `w'*fx` equivale alla valutazione della formula di quadratura

```
>> [x,w]=trapezi_composta(5,-1,1);
>> fx=feval(f,x);
>> format long e
>> I=w'*fx

I =

     1.557466063348416e+00

>> II=sum(w.*fx)

II =

     1.557466063348416e+00

>>
```

2.4. Esercizi. Quale esercizio, ricordato di scrivere le funzioni matematiche come *inline* in forma vettoriale, si approssimino (dopo averli calcolati esplicitamente o via calcolo simbolico) con le formule composte dei trapezi e di Cavalieri-Simpson come pure con le formule gaussiane (scegliere tramite α e β la funzione peso più opportuna), per $N = 5, 10, 15, 20, 25, 30$ i seguenti integrali

1. del polinomio

$$\int_{-1}^{1} x^{20}\, dx;$$

ricordando le funzioni peso di Jacobi, qual'è la funzione peso più appropriata? Il fatto che la funzione integranda sia un polinomio, spiega alcuni risultati ottenuti dalla formula gaussiana utilizzata?

2. della funzione periodica

$$\int_{-1}^{1} \sin\left(\pi \left(x+1\right)\right) dx;$$

ricordando le funzioni peso di Jacobi, qual'è la funzione peso più appropriata? (Facoltativo e richiede pazienza) Il fatto che la funzione integranda sia un polinomio, spiega i risultati ottenuti dalla formula composta dei trapezi? A tale scopo aiutarsi con il teorema di Eulero Mac-Laurin (cercare online, via motore di ricerca).

3. della funzione regolare

$$\int_{-1}^{1} \exp(x)\, dx;$$

ricordando le funzioni peso di Jacobi, qual'è la funzione peso più appropriata? Sono buone le stime dell'errore fornite dalle formule composte (2.3), (2.5)?

4. della funzione *holderiana* e poco regolare

$$\int_{-1}^{1} (1 - x^2)^{3/2}\, dx;$$

ricordando le funzioni peso di Jacobi, qual'è la funzione peso più appropriata? Sono scadenti i risultati delle formule composte rispetto a quelle gaussiane? Qual'e' la distribuzione dei nodi gaussiani prescelti nell'intervallo $[-1, 1]$? Suggerimento: Nel caso delle formule gaussiane, si osservi che se $f = g \cdot w$ allora

$$\int_a^b f(x)\, dx = \int_a^b g(x) w(x)\, dx \approx \sum_i w_i g(x_i).$$

3. Metodi iterativi. Sia A una matrice reale avente n righe ed n colonne, b un vettore colonna avente n righe e si supponga di voler risolvere il sistema lineare $Ax = b$. Come noto, se il determinante della matrice è diverso da 0 (cioè la matrice A è non singolare) allora il problema $Ax = b$ ha una ed una sola soluzione.
Ricordiamo che in Matlab/Octave la soluzione può essere calcolata con il metodo LU, utilizzando il comando \ e il comando `lu`. Un esempio:

```
>> A=[1 2 4; 2 4 16; 3 9 81];
>> b=ones(3,1);
>> x=A\b
x =
   -2.9167
    2.2083
   -0.1250
>> norm(A*x-b)
ans =  9.9301e-16
>> det(A)
ans = -24.000
>>[L,U]=lu(A);
x=(U\(L\b)))
x =
   -2.9167
    2.2083
   -0.1250
```

Uno dei principali problemi del metodo LU è legato all'alto costo computazionale. Se A è una generica matrice quadrata di ordine n infatti necessitano circa

$$O\left(\frac{n^3}{3} + \frac{n^2}{2}\right)$$

operazioni moltiplicative, che possono risultare eccessive nel caso di matrici di grandi dimensioni. Inoltre se calcoliamo la fattorizzazione LU di una matrice A sparsa cioè con un numero

elevato di elementi nulli, otteniamo che i fattori L ed U sono molto più 'pieni', ovvero si perde la proprietà di sparsità iniziale di A, e questo è un inconveniente sopratutto per matrici di grandi dimensioni. Si verifica il cosidetto fenomeno del *fill-in*, per cui a partire da una matrice sparsa, la U rimane triangolare superiore ma piena, risultando quindi più complicata della matrice di partenza.
Per ovviare a questo problema si usano ad esempio metodi iterativi stazionari del tipo

$$x^{(k+1)} = P\,x^{(k)} + c, \;\; k = 0, 1, \ldots$$

con P dipendente da A e c dipendente da A e b (ma non da k). A differenza dei metodi diretti (come ad esempio il metodo LU), in genere un metodo iterativo stazionario convergente calcola usualmente solo un approssimazione della soluzione x (a meno di una tolleranza prefissata). Se m è il numero di iterazioni necessarie, visto che ogni iterazione ha un costo $O(n^2)$ dovuto al prodotto matrice-vettore $P\,x^{(k)}$, ci si augura che il costo computazionale $O(m\,n^2)$ del metodo iterativo sia di gran lunga inferiore a $O(\frac{n^3}{3} + \frac{n^2}{2})$ di un metodo diretto quale LU.

3.1. I metodi stazionari. Sia $A = M - N$ con M invertibile. Di conseguenza, da $Ax = b$ abbiamo facilmente $Mx = Nx + b$ ed essendo M invertibile necessariamente $x = M^{-1}Nx + M^{-1}b$. In modo naturale, da quest'ultima uguaglianza, si definisce un *metodo iterativo stazionario* come

$$x^{(k+1)} = M^{-1}Nx^{(k)} + M^{-1}b. \tag{3.1}$$

La matrice $P = M^{-1}N$ è usualmente chiamata *matrice di iterazione* del metodo iterativo stazionario definito da M, N. Osserviamo che posto $c = M^{-1}b$, il metodo sopracitato è ovviamente stazionario essendo

$$x^{(k+1)} = Px^{(k)} + c \tag{3.2}$$

con P e c indipendenti da k.

3.2. Il metodo di Jacobi. Tale metodo è definito ponendo

$$M = D, \;\; N = E + F \tag{3.3}$$

e quindi

$$P = M^{-1}N = D^{-1}(E+F) = D^{-1}(D-D+E+F) = D^{-1}(D-A) = I - D^{-1}A \tag{3.4}$$

Si osservi che se D è non singolare allora il metodo di Jacobi, almeno in questa versione di base, non può essere utilizzato visto che in (3.4) non ha senso la scrittura D^{-1}.
Qualora sia $a_{ii} \neq 0$ per ogni $i = 1, \ldots, n$, il metodo di Jacobi può essere descritto come

$$x_i^{(k+1)} = (b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)})/a_{ii}, \;\; i = 1, \ldots, n. \tag{3.5}$$

3.3. Il metodo di Gauss-Seidel. Il metodo di Gauss-Seidel è definito quale metodo stazionario in cui

$$M = D - E, N = F \tag{3.6}$$

e quindi

$$P = M^{-1}N = (D - E)^{-1}F \tag{3.7}$$

Similmente al metodo di Jacobi, possiamo riscrivere più semplicemente anche Gauss-Seidel come

$$x_i^{(k+1)} = \left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)} \right) / a_{ii}. \tag{3.8}$$

3.4. Generalizzazioni del metodo di Jacobi e Gauss-Seidel. Quali generalizzazioni del metodo di Jacobi e Gauss-Seidel si introducono, per un opportuno parametro ω, la versione *rilassata del metodo di Jacobi*

$$x^{(k+1)} = (I - \omega D^{-1}A)x^{(k)} + \omega D^{-1}b \tag{3.9}$$

la versione *rilassata del metodo di Gauss-Seidel*

$$x^{(k+1)} = \left(\frac{D}{\omega} - E\right)^{-1} \left(\left(\frac{1}{\omega} - 1\right) D + F\right) x^{(k)} + \left(\frac{D}{\omega} - E\right)^{-1} b. \tag{3.10}$$

L'idea di fondo di questi metodi rilassati è l'introduzione del parametro ω per accelerarne la convergenza. Si osservi che i metodi di Jacobi e Gauss-Seidel si ottengono rispettivamente da (3.9) e (3.10) per la scelta $\omega = 1$.

3.5. Convergenza dei metodi iterativi. Sia $\rho(P)$ il massimo degli autovalori in modulo della matrice di iterazione $P = M^{-1}N$ (il cosidetto *raggio spettrale*).

TEOREMA 3.1. *Un metodo iterativo stazionario consistente $x^{(k+1)} = Px^{(k)} + c$ converge per ogni vettore iniziale x_0 se e solo se $\rho(P) < 1$.*

Questo teorema permette di mostrare alcuni casi in cui i metodi di Jacobi e Gauss-Seidel risultano convergenti.

3.5.1. Convergenza Jacobi.

1. A è a predominanza diagonale (per righe) se per ogni $i = 1, \dots, n$ risulta

   $$|a_{i,i}| \geq \sum_{j=1, j\neq s}^{n} |a_{i,j}|$$

   e per almeno un indice s si abbia

   $$|a_{s,s}| > \sum_{j=1, j\neq s}^{n} |a_{s,j}|.$$

   Ad esempio la matrice

   $$A = \begin{pmatrix} 4 & -4 & 0 \\ -1 & 4 & -1 \\ 0 & -4 & 4 \end{pmatrix}$$

   è a predominanza diagonale (per righe).

2. A è a predominanza diagonale in senso stretto (per righe) se per ogni $i = 1, \dots, n$ risulta

$$|a_{i,i}| > \sum_{j=1, j \neq i}^{n} |a_{i,j}|.$$

Ad esempio la matrice

$$A = \begin{pmatrix} 4 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 4 \end{pmatrix}$$

è a predominanza diagonale in senso stretto (per righe).

3. A è a predominanza diagonale per colonne (in senso stretto) se A^T è a predominanza diagonale per righe (in senso stretto).

4. A è tridiagonale se $a_{i,j} = 0$ per $|i - j| > 1$. Ad esempio la matrice

$$A = \begin{pmatrix} 4 & -1 & 0 & \dots & 0 \\ -1 & 4 & -1 & \dots & 0 \\ 0 & -1 & 4 & \dots & \dots \\ 0 & \dots & \dots & \dots & -1 \\ 0 & 0 & \dots & -1 & 4 \end{pmatrix}$$

è tridiagonale.

5. A è definita positiva se e solo se i suoi autovalori sono positivi.

6. A di ordine $n \geq 2$ è riducibile se esiste una matrice di permutazione Π e un intero k con $0 < k < n$, tale che

$$B = \Pi A \Pi^T = \begin{pmatrix} A_{1,1} & A_{1,2} \\ 0 & A_{2,2} \end{pmatrix}$$

in cui $A_{1,1} \in C^{k \times k}$, $A_{2,2} \in C^{(n-k)\times(n-k)}$. Se A non è riducibile si dice che A è irriducibile.

Il metodo di Jacobi risulta convergente in uno dei seguenti casi:

1. A è a predominanza diagonale in senso stretto;
2. A è a predominanza diagonale ed è irriducibile;
3. A è a predominanza diagonale in senso stretto per colonne;
4. A è a predominanza diagonale per colonne ed è irriducibile.

3.5.2. Convergenza di Gauss-Seidel. Il metodo di Gauss-Seidel risulta convergente in uno dei seguenti casi:

1. A è a predominanza diagonale in senso stretto.
2. Sia A una matrice simmetrica definita positiva, non singolare con elementi principali $a_{i,i} \neq 0$, allora Gauss-Seidel è convergente.

TEOREMA 3.2. *Per matrici tridiagonali (a blocchi) $A = (a_{i,j})$ con componenti diagonali non nulle, i metodi di Jacobi e Gauss-Seidel sono o entrambi convergenti o divergenti e il tasso di convergenza del metodo di Gauss-Seidel è il doppio di quello del metodo di Jacobi (il che vuol dire che* asintoticamente *sono necessarie metà iterazioni del metodo di Gauss-Seidel per ottenere la stessa precisione del metodo di Jacobi). Si può infatti dimostrare che $\rho(P_{Gs}) = \rho(P_J)^2$*

3.6. Metodi di discesa. Ad ogni sistema lineare $Ax = b$, con A matrice simmetrica definita positiva si può associare il funzionale (forma quadratica)

$$\phi(x) = \frac{1}{2}x^T Ax - b^T x, \quad x \in \mathbb{R}^n$$

il cui punto di minimo x^* è l'unica soluzione del sistema lineare, cioè $Ax^* = b$ (che esprime la condizione $\Delta\phi = 0$).

E' possibile quindi costruire dei metodi iterativi che, ad ogni passo, procedano verso il minimo della forma quadratica secondo il seguente schema:

$$x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}, \qquad k \geq 0, \alpha_k \neq 0 \tag{3.11}$$

dove $p^{(k)}$ è una direzione di discesa fissata secondo qualche criterio e α_k è un parametro di accelerazione. I metodi iterativi scritti nella forma (3.11) prendono il nome di *metodi di Richardson*. Il metodo si definisce *stazionario* nel caso in cui $\alpha_k = \alpha$ con α costante assegnata, *dinamico* nel caso in cui α_k possa cambiare ad ogni iterazione.
Vediamo di seguito alcuni di questi metodi.

3.7. Il metodo del gradiente classico. Si dimostra che il parametro α_k ottimale cosicchè $\phi(x^{(k+1)})$ sia minimo una volta scelta $p^{(k)}$ è

$$\alpha_k = \frac{(r^{(k)})^T p^{(k)}}{(p^{(k)})^T A p^{(k)}}$$

Nel metodo del gradiente si sceglie quale direzione $p^{(k)} = \text{grad}(\phi(x))|_{x=x^{(k)}}$. Ma se $r^{(k)} = b - Ax^{(k)}$, allora

$$\begin{aligned} \text{grad}(\phi(x))|_{x=x^{(k)}} &= \frac{1}{2}\text{grad}(x^T Ax)|_{x=x^{(k)}} - \text{grad}(b^T x)|_{x=x^{(k)}} \\ &= Ax^{(k)} - b = -r^{(k)} \end{aligned} \tag{3.12}$$

e quindi $p^{(k)} = r^{(k)}$ (è essenziale la direzione ma non il segno e per convincersene si calcoli la successione anche con segno opposto $p^{(k)} = -r^{(k)}$ per parametro α_k ottimale).
Di conseguenza il metodo del gradiente è definito dalla successione tipica dei metodi di Richardson non stazionari

$$x^{(k+1)} = x^{(k)} + \alpha_k r^{(k)}$$

dove

$$\alpha_k = \frac{(r^{(k)})^T p^{(k)}}{(p^{(k)})^T A p^{(k)}} = \frac{\|r^{(k)}\|_2^2}{(r^{(k)})^T A r^{(k)}}.$$

Si può dimostrare che le direzioni utilizzate dal metodo del gradiente sono ortogonali, ovvero tali che $(p^{(k+1)})^T p^{(k)} = 0$. Questa proprietà in generale non è valida per due direzioni non

consecutive (cioè , ad esempio, $(p^{(6)})^T p^{(1)} \neq 0$), e quindi il metodo del gradiente non è ancora ottimale. Nel caso del metodo del gradiente, vale la stima

$$\|e^{(k)}\|_A \leq \left(\frac{K_2(A)-1}{K_2(A)+1}\right)^k \|e^{(0)}\|_A$$

con

$$K_2(A) := \|A\|_2\|A^{-1}\|_2 = \frac{\lambda_{max}(A)}{\lambda_{min}(A)}$$

che mostra che più grande è il numero di condizionamento $K_2(A)$ più è vicino a 1 la quantità $\frac{K_2(A)-1}{K_2(A)+1}$ il che giustifica una *possibile* convergenza lenta del metodo.

3.7.1. Facoltativo: Il metodo del gradiente coniugato. La successione delle iterazioni del gradiente coniugato è quella propria dei metodi di discesa,

$$x^{(k+1)} = x^{(k)} + \alpha_k p^{(k)}, \quad \alpha_k = \frac{(r^{(k)})^T r^{(k)}}{(p^{(k)})^T A p^{(k)}}$$

dove $p^{(0)} = r^{(0)}$ e

$$p^{(k)} = r^{(k)} + \beta_k p^{(k-1)}, \quad \beta_k = \frac{(r^{(k)})^T r^{(k)}}{(r^{(k-1)})^T r^{(k-1)}}.$$

Con questa scelta si prova che

$$(p^{(k)})^T A p^{(k-1)} = 0,$$

cioè i vettori $p^{(k)}$ e $p^{(k-1)}$ sono A-*coniugati*.
Questa proprietà è decisiva, in quanto si può dimostrare che il gradiente coniugato converge alla soluzione esatta in al più n passi, se A è di ordine n ed ammesso di riuscire a lavorare in aritmetica esatta.

3.7.2. Convergenza del gradiente coniugato. Il metodo del gradiente coniugato ha molte proprietà particolari. Ne citiamo alcune.

TEOREMA 3.3. *Se A è una matrice simmetrica e definita positiva di ordine n e $p^{(0)} = r^{(0)}$, il metodo del gradiente coniugato è convergente e fornisce in aritmetica esatta la soluzione del sistema $Ax = b$ in al massimo n iterazioni.*

Questo teorema tradisce un pò le attese, sia perchè in generale i calcoli non sono compiuti in aritmetica esatta, sia perchè in molti casi della modellistica matematica n risulta essere molto alto.

TEOREMA 3.4. *Se A è simmetrica e definita positiva,*

$$\|x\|_A = \sqrt{x^T A x}$$

e

$$e_k = x^* - x^{(k)}$$

allora

$$\|e_k\|_A \leq \left(\frac{\sqrt{K_2(A)}-1}{\sqrt{K_2(A)}+1}\right)^{2k} \|e_0\|_A. \tag{3.13}$$

Questo risultato stabilisce che la convergenza del gradiente coniugato è lenta qualora si abbiano alti numeri di condizionamento

$$K_2(A) := \|A\|_2 \|A^{-1}\|_2$$

TEOREMA 3.5. *Sia A simmetrica e definita positiva. Si supponga che ci siano esattamente $k \leq n$ autovalori distinti di A. Allora il metodo del gradiente coniugato in cui si è posto $p^{(0)} = r^{(0)} = b - Ax^{(0)}$ converge in al più k iterazioni.*

TEOREMA 3.6. *Sia A simmetrica e definita positiva. Si supponga b sia combinazione lineare di $k \leq n$ autovettori distinti di A. Allora il metodo del gradiente coniugato con la scelta $x^{(0)} = 0$ converge in al più k iterazioni.*

3.8. Precondizionamento. La maggiorazione 3.13 mette in evidenza l'importanza del numero di condizionamento per quanto concerne la rapidità di convergenza dei metodi di discesa. L'idea del precondizionamento consiste nel sostituire la risoluzione dellequazione $Ax = b$ con quella del sistema equivalente $P^{-1}Ax = P^{-1}b$, ove la matrice P simmetrica, definita positiva e invertibile, è scelta con l'obiettivo di avere $K_2(P^{-1}A) << K_2(A)$.
Naturalmente, la scelta migliore sarebbe $P^{-1} = A^{-1}$, perchè in questo caso si avrebbe $K_2(P^{-1}A) = 1$; nella pratica, si tratta, al solito, di trovare un opportuno compromesso con il costo del calcolo di $(P^{-1}A)$.
Assumendo P^{-1} simmetrica definita positiva, osserviamo che la matrice $P^{-1}A$ non è , in generale, simmetrica e quindi non possiamo applicare l'algoritmo del gradiente coniugato direttamente a $P^{-1}A$. Tuttavia, si può definire una matrice $P^{1/2}$ simmetrica e definita positiva tale che $P^{1/2}P^{1/2} = P$ (si può dimostrare che una tale matrice, detta 'radice quadrata' di P, esiste) di conseguenza, la matrice

$$P^{1/2}(P^{-1}A)P^{-1/2} = P^{-1/2}(A)P^{-1/2}$$

è simmetrica definita positiva e si può applicare il metodo del gradiente coniugato alla nuova matrice $(\tilde{A}) = P^{-1/2}(A)P^{-1/2}$, ovvero si risolve il sistema

$$P^{-1/2}(A)P^{-1/2}y = P^{-1/2}b, \quad \text{con} \quad y = P^{1/2}x$$

In pratica, non occorre disporre della matrice $P^{1/2}$ o della sua inversa. Infatti l'algoritmo che realizza il metodo del gradiente coniugato precondizionato ad ogni passo risolve un sistema lineare nella matrice P.
Esistono in letteratura diversi tipi di precondizionatore, ma la loro individuazione è basata più sull'empirismo che su argomentazioni teoriche, spesso è sufficiente prendere $P = diag(A)$.

3.9. Metodi iterativi in Matlab. Vediamo di seguito tre routines Matlab che implementano i metodi sopra descritti di Jacobi, Gauss-Seidel e del gradiente coniugato.

3.9.1. Implementazione del metodo di Jacobi. In questa sottosezione suggeriamo un codice Matlab che implementa il metodo di Jacobi. In input, la matrice `A` e il vettore `b` sono relativi al sistema $Ax = b$ da risolvere, `x` è il vettore colonna iniziale da cui far partire le iterazioni, `max_it` il numero massimo di iterazioni da compiere e `tol` la tolleranza richiesta da un certo criterio di arresto. Delle variabili in output, `x` è l'approssimazione della soluzione, `error` l'errore in norma 2 compiuto, `iter` il numero di iterazioni compiute, `flag` una variabile che a seconda sia 0 o 1 l'algoritmo è uscito correttamente oppure per il numero massimo di iterazioni richieste.

```
function [x, error, iter, flag]  = jacobi(A, x, b, max_it, tol)

%  -- Iterative template routine --
%     Univ. of Tennessee and Oak Ridge National Laboratory
%     October 1, 1993
%     Details of this algorithm are described in "Templates for the
%     Solution of Linear Systems: Building Blocks for Iterative
%     Methods", Barrett, Berry, Chan, Demmel, Donato, Dongarra,
%     Eijkhout, Pozo, Romine, and van der Vorst, SIAM Publications,
%     1993. (ftp netlib2.cs.utk.edu; cd linalg; get templates.ps).
%
% [x, error, iter, flag]  = jacobi(A, x, b, max_it, tol)
%
% jacobi.m solves the linear system Ax=b using the Jacobi Method.
%
% input   A        REAL matrix
%         x        REAL initial guess vector
%         b        REAL right hand side vector
%         max_it   INTEGER maximum number of iterations
%         tol      REAL error tolerance
%
% output  x        REAL solution vector
%         error    REAL error norm
%         iter     INTEGER number of iterations performed
%         flag     INTEGER: 0 = solution found to tolerance
%                           1 = no convergence given max_it

  iter = 0;                                        % initialization
  flag = 0;

  bnrm2 = norm( b );
  if  ( bnrm2 == 0.0 ), bnrm2 = 1.0; end

  r = b - A*x;
  error = norm( r ) / bnrm2;
  if ( error < tol ) return, end

  [m,n]=size(A);
  [ M, N ] = split( A , b, 1.0, 1 );       % matrix splitting

  for iter = 1:max_it,                             % begin iteration

     x_1 = x;
     x   = M \ (N*x + b);                          % update approximation

     error = norm( x - x_1 ) / norm( x );          % compute error
     if ( error <= tol ), break, end               % check convergence

  end

  if ( error > tol ) flag = 1; end                 % no convergence
```

Il codice di `jacobi` utilizza una funzione `split` che serve per calcolare le matrici M, N che definiscono l'iterazione del metodo di Jacobi:

```
function [ M, N, b ] = split( A, b, w, flag )
%
% function [ M, N, b ] = split_matrix( A, b, w, flag )
%
% split.m sets up the matrix splitting for the stationary
% iterative methods: jacobi and sor (gauss-seidel when w = 1.0 )
%
% input    A        DOUBLE PRECISION matrix
%          b        DOUBLE PRECISION right hand side vector (for SOR)
%          w        DOUBLE PRECISION relaxation scalar
%          flag     INTEGER flag for method: 1 = jacobi
%                                            2 = sor
%
% output   M        DOUBLE PRECISION matrix
%          N        DOUBLE PRECISION matrix such that A = M - N
%          b        DOUBLE PRECISION rhs vector ( altered for SOR )

  [m,n] = size( A );

  if ( flag == 1 ),                    % jacobi splitting

     M = diag(diag(A));
     N = diag(diag(A)) - A;

  elseif ( flag == 2 ),                % sor/gauss-seidel splitting

     b = w * b;
     M =  w * tril( A, -1 ) + diag(diag( A ));
     N = -w * triu( A,  1 ) + ( 1.0 - w ) * diag(diag( A ));

  end;

% END split.m
```

La routine `jacobi` è scritta da esperti di algebra lineare e si interrompe quando la norma 2 dello step relativo

$$\frac{\|x^{(k+1)} - x^{(k)}\|_2}{\|x^{(k+1)}\|_2}$$

è inferiore ad una tolleranza `tol` prefissata oppure un numero massimo di iterazioni `max_it` è raggiunto. Ricordiamo che se $v = (v_i)_{i=1,\dots,n}$ è un elemento di $\mathbb{R}^n$ allora

$$\|v\|_2 = \sqrt{\sum_{i=1}^{n} v_i^2}.$$

3.9.2. Implementazione del metodo di Gauss-Seidel. In questa sottosezione suggeriamo un codice Matlab che implementa una generalizzazione del metodo di Gauss-Seidel nota come SOR. Il metodo di Gauss-Seidel corrisponde alla scelta di $w = 1$. In input, la matrice `A` e il vettore `b` sono relativi al sistema $Ax = b$ da risolvere, `x` è il vettore colonna iniziale da cui

far partire le iterazioni, max_it il numero massimo di iterazioni da compiere e tol la tolleranza richiesta da un certo criterio di arresto. Delle variabili in output, x è l'approssimazione della soluzione, error l'errore in norma 2 compiuto, iter il numero di iterazioni compiute, flag una variabile che a seconda sia 0 o 1 l'algoritmo è uscito correttamente oppure per il numero massimo di iterazioni richieste.

```
function [x, error, iter, flag]  = sor(A, x, b, w, max_it, tol)

%  -- Iterative template routine --
%     Univ. of Tennessee and Oak Ridge National Laboratory
%     October 1, 1993
%     Details of this algorithm are described in "Templates for the
%     Solution of Linear Systems: Building Blocks for Iterative
%     Methods", Barrett, Berry, Chan, Demmel, Donato, Dongarra,
%     Eijkhout, Pozo, Romine, and van der Vorst, SIAM Publications,
%     1993. (ftp netlib2.cs.utk.edu; cd linalg; get templates.ps).
%
% [x, error, iter, flag]  = sor(A, x, b, w, max_it, tol)
%
% sor.m solves the linear system Ax=b using the
% Successive Over-Relaxation Method (Gauss-Seidel method when omega = 1 ).
%
% input   A        REAL matrix
%         x        REAL initial guess vector
%         b        REAL right hand side vector
%         w        REAL relaxation scalar
%         max_it   INTEGER maximum number of iterations
%         tol      REAL error tolerance
%
% output  x        REAL solution vector
%         error    REAL error norm
%         iter     INTEGER number of iterations performed
%         flag     INTEGER: 0 = solution found to tolerance
%                           1 = no convergence given max_it

  flag = 0;                                  % initialization
  iter = 0;

  bnrm2 = norm( b );
  if  ( bnrm2 == 0.0 ), bnrm2 = 1.0; end

  r = b - A*x;
  error = norm( r ) / bnrm2;
  if ( error < tol ) return, end

  [ M, N, b ] = split( A, b, w, 2 );         % matrix splitting

  for iter = 1:max_it                        % begin iteration

     x_1 = x;
     x   = M \ ( N*x + b );                  % update approximation

     error = norm( x - x_1 ) / norm( x );    % compute error
```

```
    if ( error <= tol ), break, end          % check convergence

  end
  b = b / w;                                 % restore rhs

  if ( error > tol ) flag = 1; end;          % no convergence
```

Come per il metodo di Jacobi, il processo si interrompe quando la norma 2 dello step relativo

$$\frac{\|x^{(k+1)} - x^{(k)}\|_2}{\|x^{(k+1)}\|_2}$$

è inferiore ad una tolleranza `tol` prefissata oppure un numero massimo di iterazioni `max_it` è raggiunto.

3.9.3. Implementazione del metodo del gradiente coniugato. Per quanto riguarda il codice del Gradiente Coniugato, un esempio è il file `cg.m` tratto dal sito di Netlib.

In input, la matrice `A` e il vettore `b` sono relativi al sistema $Ax = b$ da risolvere, `x` è il vettore colonna iniziale da cui far partire le iterazioni, `max_it` il numero massimo di iterazioni da compiere e `tol` la tolleranza richiesta da un certo criterio di arresto. Infine `M` è un possibile precondizionatore. Se non si sa bene cosa scegliere quale precondizionatore, si ponga `M=eye(size(A))`. Delle variabili in output, `x` è l'approssimazione della soluzione, `error` l'errore in norma 2 compiuto, `iter` il numero di iterazioni compiute, `flag` una variabile che a seconda sia 0 o 1 l'algoritmo è uscito correttamente oppure per il numero massimo di iterazioni richieste.

```
function [x, error, iter, flag] = cg(A, x, b, M, max_it, tol)

%  -- Iterative template routine --
%     Univ. of Tennessee and Oak Ridge National Laboratory
%     October 1, 1993
%     Details of this algorithm are described in "Templates for the
%     Solution of Linear Systems: Building Blocks for Iterative
%     Methods", Barrett, Berry, Chan, Demmel, Donato, Dongarra,
%     Eijkhout, Pozo, Romine, and van der Vorst, SIAM Publications,
%     1993. (ftp netlib2.cs.utk.edu; cd linalg; get templates.ps).
%
%  [x, error, iter, flag] = cg(A, x, b, M, max_it, tol)
%
% cg.m solves the symmetric positive definite linear system Ax=b
% using the Conjugate Gradient method with preconditioning.
%
% input   A        REAL symmetric positive definite matrix
%         x        REAL initial guess vector
%         b        REAL right hand side vector
%         M        REAL preconditioner matrix
%         max_it   INTEGER maximum number of iterations
%         tol      REAL error tolerance
%
% output  x        REAL solution vector
%         error    REAL error norm
%         iter     INTEGER number of iterations performed
%         flag     INTEGER: 0 = solution found to tolerance
```

```
%                           1 = no convergence given max_it

  flag = 0;                                  % initialization
  iter = 0;

  bnrm2 = norm( b );
  if  ( bnrm2 == 0.0 ), bnrm2 = 1.0; end

  r = b - A*x;
  error = norm( r ) / bnrm2;
  if ( error < tol ) return, end

  for iter = 1:max_it                        % begin iteration

     z  = M \ r;
     rho = (r'*z);

     if ( iter > 1 ),                        % direction vector
        beta = rho / rho_1;
        p = z + beta*p;
     else
        p = z;
     end

     q = A*p;
     alpha = rho / (p'*q );
     x = x + alpha * p;                     % update approximation vector

     r = r - alpha*q;                       % compute residual
     error = norm( r ) / bnrm2;             % check convergence
     if ( error <= tol ), break, end

     rho_1 = rho;

  end

  if ( error > tol ) flag = 1; end          % no convergence

% END cg.m
```

Osserviamo che il procedimento itera finchè un numero massimo di iterazioni è raggiunto oppure la norma 2 del residuo (relativo)

$$\frac{\|b - Ax^{(k)}\|_2}{\|b\|_2}$$

immagazzinata nella variabile `error` risulta inferiore ad una tolleranza prefissata `tol`. In questo caso il criterio d'arresto del metodo del gradiente coniugato è diverso da quello dello step relativo utilizzato nelle precedenti versioni di `Jacobi` ed `SOR`.

3.9.4. Un esempio. Prendiamo quale esempio una matrice di Toeplitz, simmetrica e definita positiva, dalla `gallery` di Matlab.

```
>> % MATRICE DI TOEPLITZ, SIMMETRICA E DEFINITA POSITIVA.
```

```
>> A=gallery('toeppd',5)
A =
    3.3932    1.8817    0.8909    1.3019    0.9454
    1.8817    3.3932    1.8817    0.8909    1.3019
    0.8909    1.8817    3.3932    1.8817    0.8909
    1.3019    0.8909    1.8817    3.3932    1.8817
    0.9454    1.3019    0.8909    1.8817    3.3932
 >> % CONTROLLO SIMMETRIA.
 >> A-A'
ans =
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
>> % CONTROLLO DEF. POSITIVA.
>> eig(A)
ans =
    0.5499
    1.8948
    2.5502
    3.0555
    8.9159
>> % TERMINE NOTO.
>> b=[1; 1; 1; 1; 1]
b =
     1
     1
     1
     1
     1
>> % SOLUZIONE COL METODO LU.
>> format long e
>> x=A\b
x =
     1.743211621943181e-01
     2.855210174362449e-02
     1.715052207727628e-01
     2.855210174362447e-02
     1.743211621943181e-01
>> %-----------------------------------------------------
>> % JACOBI (NON E' DETTO CONVERGA PER MATRICI SIMMETRICHE!!!)
>> %-----------------------------------------------------
>> x0=b; max_it=1000; tol=10^(-12);
>> t0=cputime; [x1,error,iter,flag]=jacobi(A,x0,b,max_it,tol); t1=cputime;
>> x1
x2 =
    2.812150840584267e+211
    3.182477056364826e+211
    3.075904398497104e+211
    3.182477056364826e+211
    2.812150840584267e+211
>> t1-t0
ans =
```

```
     3.000000000000114e-02
>> flag
flag =
     1
>> % ANALISI: JACOBI NON CONVERGE.
>> %-----------------------------------------------------
>> % GAUSS-SEIDEL (CONVERGE!). "sor" CON "w=1".
>> %-----------------------------------------------------
>> w=1;
>> t0=cputime; [x2,error,iter,flag]=sor(A,x0,b,w,max_it,tol);; t1=cputime;
>> x2
x2 =
     1.743211621945437e-01
     2.855210174332565e-02
     1.715052207730155e-01
     2.855210174338184e-02
     1.743211621944382e-01
>> error
error =
     7.271979844164592e-13
>> iter
iter =
    76
>> flag
flag =
     0
>> t1-t0 % CPUTIME: SCRIVE 0 PERCHE' MOLTO PICCOLO.
ans =
     0
>> %----------------------------------------
>> % GRADIENTE CONIUGATO.
>> %----------------------------------------
>> M=eye(size(A)); % NESSUN PRECONDIZIONATORE.
>> t0=cputime; [x3,error,iter,flag]=cg(A,x0,b,M,max_it,tol); t1=cputime;
>> x3
x3 =
     1.743211621943214e-01
     2.855210174363323e-02
     1.715052207727673e-01
     2.855210174363175e-02
     1.743211621943232e-01
>> error
error =
     5.198603009807900e-14
>> iter
iter =
     3
>> % ANALISI: NUMERO DI ITERAZIONI INFERIORE O UGUALE
>> % ALLA DIMENSIONE DELLA MATRICE.
>> flag
flag =
     0
>> t1-t0
ans =
```

```
    9.999999999998010e-03
>>
```

3.10. Esercizio svolto 1. Si consideri la seguente matrice pentadiagonale $A \in \mathbb{R}^{100\times 100}$:

$$A = \begin{bmatrix} 4+\beta & -1 & -1 & & & & \\ -1 & 4+\beta & -1 & -1 & & & \\ -1 & -1 & 4+\beta & -1 & -1 & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \\ & & -1 & -1 & 4+\beta & -1 & -1 \\ & & & -1 & -1 & 4+\beta & -1 \\ & & & & -1 & -1 & 4+\beta \end{bmatrix}$$

1. Creare la matrice in Matlab utilizzando il comando `diag`, dopo aver posto $\beta = 0.1$. Determinare il numero di elementi non nulli tramite il comando `nnz` e visualizzare il pattern di sparsità della matrice con il comando `spy`. Convertire la matrice in formato sparso utilizzando il comando `sparse` e verificare la differente occupazione di memoria nei due casi (in caso di matrici con struttura a banda come quella nellesempio, è possibile creare direttamente la matrice in formato sparso con il comando `spdiags`)
2. Si costruiscano esplicitamente le matrici di iterazione P_J e P_{GS} associate rispettivamente ai metodi di Jacobi e GaussSeidel, a partire dallo splitting della matrice A (si usino i comandi `diag`, `tril`, `inv`). Si calcolino i relativi raggi spettrali $\rho(P_J)$ e $\rho(P_{GS})$, utilizzando il comando `eig`. La condizione necessaria e sufficiente per la convergenza del metodo iterativo è soddisfattain entrambi i casi? Commentare, alla luce della teoria, i risultati ottenuti.
3. Utilizzare la funzione `jacobi` che implementa il metodo di Jacobi per un generico sistema lineare $Ax = b$ per calcolare la soluzione del sistema lineare con A matrice di partenza e $b = [1, 1, \ldots, 1]^T$. Si ponga come vettore iniziale $x = [0, 0, \ldots, 0]^T$ tolleranza pari a $1e^{-10}$ e come numero massimo di iterazioni 1000.
4. Utilizzare la funzione `sor` che implementa il metodo di Gauss-Seidel ($\omega = 1$) per un generico sistema lineare $Ax = b$ per risolvere il sistema precedente.
5. Si confrontino i risultati ottenuti tramite i due metodi e si commentino i risultati ottenuti considerando i raggi spettrali delle matrici di iterazione.
6. Si ponga ora $\beta = -0.1$ nella matrice A. Si utilizzino nuovamente i metodi di Jacobi e Gauss-Seidel per risolvere il sistema lineare con i medesimi parametri scelti al punto 3. Si calcolino i raggi spettrali delle matrici di iterazione associate ai due metodi. Commentare, alla luce della teoria, i risultati ottenuti.

Soluzione

1. La matrice può essere costruita tramite i seguenti comandi:

```
>>n = 100;
>>UNIT = ones(n,1);
>>beta = 0.1;
>>A = (4+beta)*diag(UNIT) - diag(UNIT(1:n-1),-1) ...
- diag(UNIT(1:n-1),1) - diag(UNIT(1:n-2),-2) - diag(UNIT(1:n-2),2);
```

Il comando `nnz` restituisce il numero di elementi non nulli della matrice data in ingresso; l'output in questo caso è :

```
>>nnz(A)
ans =
    494
```

Si può convertire la matrice in formato sparso tramite il comando `sparse` e verificare la diversa occupazione di memoria con il comando `whos`:

```
>>Asp = sparse(A);
>>whos A Asp
  Name       Size                    Bytes  Class

  A        100x100                   80000  double array
  Asp      100x100                    6332  double array (sparse)
```

Si può vedere come il formato sparso riduca la memoria richiesta per memorizzare una stessa matrice rispetto al formato pieno.

2. Le matrici di iterazione dei due metodi si calcolano a partire dalla definizione:

```
>>D = diag(diag(A));       % D diagonale estratta da A
>>PJ = inv(D) * (D-A);     % matrice di iterazione di Jacobi
>>E = -tril(A,-1);         % E matrice tr. inferiore di A
>>F = -triu(A,1);          % F matrice tr. superiore di A
>>PGs = inv(D-E)*F         % matrice di iterazione di Gauss-Seidel

>>rho_j = max(abs(eig(PJ)))
>>rho_gs = max(abs(eig(PGs)))

rho_j =
    0.9744
rho_gs =
    0.9496
```

Dal calcolo del raggio spettrale delle matrici si può concludere che sia il metodo di Jacobi che il metodo di GaussSeidel convergono, in quanto entrambi gli autovalori massimi risultano in modulo strettamente minore di 1.

3. Per risolvere il sistema è sufficiente richiamare la funzione, dopo aver definito tutti i parametri di ingresso che richiede:

```
>>x0 = zeros(n,1); b = ones(n,1); max_it = 1000;tol = 1e-10;
>>[xJ,errorJ,iterJ,flagJ]  = jacobi(A, x0, b, max_it, tol)
xJ =

    1.8133
    2.7016
    3.7329
    4.5302
    5.2599
    5.8795
    6.4229
```

```
    6.8929
    7.3017
    ......
    1.8133

errorJ =
  9.8528e-011
iterJ =
   748
flagJ =
     0
```

Il metodo di Jacobi converge in 748 iterazioni.

4. Per risolvere il sistema è sufficiente richiamare la funzione, utilizzando gli stessi parametri di ingresso del caso precedente:

```
>>w=1;
>>[xGs,errorGs,iterGs,flagGs] = sor(A, x0, b, w, max_it, tol)
xGs =

    1.8133
    2.7016
    3.7329
    ......
    2.7016
    1.8133

errorGs =
  9.7019e-011
iterGs =
   391
flagGs =
     0
```

Il metodo di GaussSeidel converge in 391 iterazioni.

5. Grazie alla teoria, dato che i due metodi risultano convergenti, il metodo di Gauss-Seidel risulta più veloce del metodo di Jacobi. Infatti, in questo esempio, il numero di iterazioni del metodo di Jacobi è circa il doppio del numero di iterazioni del metodo di GaussSeidel.

6. Con il nuovo parametro β si crea la nuova matrice e si richiamano le funzioni

```
beta = -0.1;
>>A2 = (4+beta)*diag(UNIT) - diag(UNIT(1:n-1),-1)...
-diag(UNIT(1:n-1),1) - diag(UNIT(1:n-2),-2) - diag(UNIT(1:n-2),2);

>>[xJ,errorJ,iterJ,flagJ]  = jacobi(A2, x0, b, max_it, tol)
>>[xGs,errorGs,iterGS,flagGs] = sor(A2, x0, b, w, max_it, tol)
```

In uscita si avranno flagJ e flagGs uguali a 1, ovvero i metodi non convergono. In effetti i raggi spettrali delle matrici di iterazione sono entrambi maggiori di 1 quindi per entrambi i metodi la condizione necessaria per la convergenza non è soddisfatta.

```
>>D = diag(diag(A2));
>>Pj = inv(D) * (D-A2);
```

```
>>E = -tril(A2,-1);
>>F = -triu(A2,1);
>>Pgs = inv(D-E)*F

>>rho_j = max(abs(eig(Pj)))
>>rho_gs = max(abs(eig(Pgs)))

rho_j =
    1.0244
rho_gs =
    1.0495
```

Se volessimo rappresentare graficamente l'andamento dell'errore al variare delle iterazioni, basterà modificare le function in modo che venga memorizzati ad ogni iterazione.

```
function [x, E, iter, flag]  = jacobiE(A, x, b, max_it, tol)
        .
        .
        .
        .
  iter = 0;                                          % initialization
  flag = 0;

  bnrm2 = norm( b );
  if  ( bnrm2 == 0.0 ), bnrm2 = 1.0; end

  r = b - A*x;
  error = norm( r ) / bnrm2;
  E= [error];                             %----> Inizialization error vector
  if ( error < tol ) return, end

  [m,n]=size(A);
  [ M, N ] = split( A , b, 1.0, 1 );                 % matrix splitting

  for iter = 1:max_it,                               % begin iteration

     x_1 = x;
     x   = M \ (N*x + b);                            % update approximation

     error = norm( x - x_1 ) / norm( x );            % compute error
     E = [E; error];                      %----> collect error vector

     if ( error <= tol ), break, end                 % check convergence

  end

  if ( error > tol ) flag = 1; end                   % no convergence
```

Si richiama la funzione sempre con gli stessi parametri e poi si ottiene il grafico in scala semilogaritmica dell'andamento dell'errore memorizzato nel vettore EJ:

```
[xJ,EJ,iterJ, flagJ]  = jacobiE(A, x0, b, max_it, tol);
semilogy([0:iterJ],EJ,'b');
```

```
legend('jacobi')
grid
```

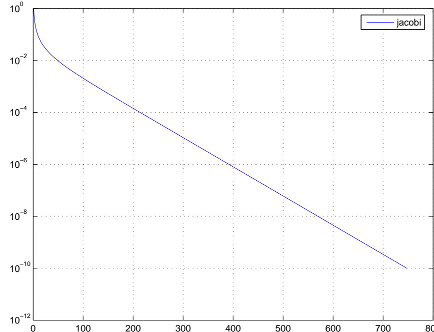

FIGURA 3.1. *Andamento dell'errore nel metodo di Jacobi*

Coerentemente con il criterio d'arresto utilizzato, il metodo si arresta quando l'errore raggiunge la tolleranza fissata di $1e^{-10}$

3.11. Esercizi proposti.

1. Si modifichino le function `sor` e `cg` in modo che diano in uscita il vettore degli errori e le si utilizzino per graficare l'andamento dell'errore dei tre metodi come visto nell'esercizio svolto relativamente al sistema precedente (si usi il comando `hold` on per trattenere l'ambiente grafico e plottare i tre grafici sulla stessa finestra.)

2. Si considerino le matrici $T \in \mathbb{R}^{N\times N}$ ed $F \in \mathbb{R}^{N\times N}$ $N = 47$, definite attraverso i seguenti comandi Matlab,

```
N=47;
T=2*diag(ones(N,1),0) - diag(ones(N-1,1),1) - diag(ones(N-1,1),-1);
F=6*diag(ones(N,1),0) - 4*diag(ones(N-1,1),1) - 4*diag(ones(N-1,1),-1) ...
+ diag(ones(N-2,1),2) + diag(ones(N-2,1),-2);
```

Sia fissato $h = 1/(N+3) = 1/50$, definire la matrice $A \in \mathbb{R}^{N\times N}$ e il termine noto $f \in \mathbb{R}^N$ tali che

$$A = \frac{T}{h^2} + \frac{F}{h^4}, \qquad f = -[1, 1, \dots, 1]^T$$

dove T ed F sono le matrici definite sopra. il sistema lineare $Au = f$ deriva dall'approssimazione numerica dello spostamento verticale di una trave di lunghezza unitaria incastrata agli estremi e flessa da una forzante uniforme di intensità unitaria diretta verso il basso.

(a) Verificare che A è simmetrica e definita positiva e calcolare il numero di condizionamento $K_2(A)$ (si usi il comando `cond` aiutandosi eventualmente con l'Help di Matlab).

(b) Risolvere il sistema lineare $Au = f$ con i metodi di Gauss Seidel, Jacobi e metodo del gradiente coniugato non precondizionato con una tolleranza $tol = 10^{-10}$ e un numero massimo di iterazioni $nmax = 1000$. Riportare il numero

di iterazioni effettuate e visualizzare il grafico della funzione discreta u_i per $i = 1, \ldots, N$ rispetto a vettore $x_i = (i + 1)h$, $i = 1, \ldots, N$.

(c) Si risolva lo stesso sistema usando il metodo del gradiente coniugato precondizionato. Come precondizionatori si confrontino le seguenti scelte per il precondizionatore:
• il precondizionatore P_1 uguale alla matrice diagonale estratta da A.
• il precondizionatore P_2 uguale alla matrice T/h^2.
Si commentino i risultati ottenuti relativamente al numero di iterazioni richieste per la convergenza in relazione al numero di condizionamento della matrice A e della matrice $P^{-1}A$. Quale precondizionatore risulta più efficace?

3.12. Esercizi facoltativi.

1. Si consideri la seguente matrice $A \in \mathbb{R}^{N \times N}$:

$$A = \begin{bmatrix} 1 & -1 & 1 & 1 & \cdots & 1 \\ R_1 & R_2 & 0 & 0 & \cdots & 0 \\ 0 & R_1 & R_2 & 0 & \cdots & 0 \\ \vdots & 0 & \ddots & \ddots & & \vdots \\ \vdots & \vdots & & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots R_1 & R_2 & \end{bmatrix}$$

Si ponga $n = 100$, $R_1 = 1$, $R_2 = 3$.

(a) Memorizzare la matrice in formato pieno, e determinare il numero di elementi non nulli tramite il comando `nnz` e visualizzare il pattern di sparsità della matrice con il comando `spy`.

(b) Si costruiscano esplicitamente le matrici di iterazione P_J e P_{GS} associate rispettivamente ai metodi di Jacobi e GaussSeidel, a partire dallo splitting della matrice A. Si calcolino i relativi raggi spettrali $\rho(P_J)$ e $\rho(P_{GS})$, utilizzando il comando `eig`. La condizione necessaria e sufficiente per la convergenza del metodo iterativo è soddisfattain entrambi i casi? Commentare, alla luce della teoria, i risultati ottenuti.

(c) Si determini la soluzione del sistema lineare $Ax = b$ con b vettore di dimensione n:

$$b = [2, 1, \ldots, 1]^T$$

con il metodo di Jacobi e Gauss Seidel.
Si ponga $x_0 = [1, 1, \cdots, 1]^T$ tolleranza $tol = 10^{-6}$ e numero massimo di iterazioni $nmax = 1000$.

(d) Si confrontino i risultati ottenuti tramite i due metodi e si commentino i risultati ottenuti alla luce del numero di condizionamento per la matrice A.

(e) Si calcoli la fattorizzazione LU della matrice A; cosa si può osservare confrontando il pattern di sparsità delle matrici L e U con quello di A? Che fenomeno si è verificato? Commentare i risultati ottenuti confrontando il metodo LU con i metodi iterativi visti.

2. Sia data la matrice $A \in \mathbb{R}^{10\times 10}$ seguente,

$$A = \begin{bmatrix} 3 & -1 & 0 & 0 & \\ -1 & 3 & -1 & 0 & \\ 0 & -1 & 3 & -1 & \\ & \ddots & \ddots & \ddots & \\ & 0 & 0 & -1 & 3 \end{bmatrix}$$

e il sistema lineare associato $Ax = b$, ove b è scelto in modo tale che la soluzione esatta del sistema sia $x = \left[\ 1, 1, \cdots, 1\ \right]^T$

(a) Verificare che la matrice A è simmetrica, definita positiva e a dominanza diagonale stretta. Stabilire inoltre se i metodi iterativi di Jacobi, Gauss-Seidel sono convergenti. Si cosideri inoltre il metodo SOR, con matrice di iterazione

$$P_{SOR} = (D - \omega E)^{-1}(\omega F + (1 - \omega)D)$$

utilizzando il programma `sor` applicato al sistema lineare $Ax = b$ con tolleranza *toll*=10^{-6}, numero massimo di iterazioni $N_{max} = 1000$, e per 50 valori di ω equidistribuiti su $[-3, 3]$ si traccino in funzione di ω i grafici sovrapposti di

• raggio spettrale $\rho(P_{SOR(\omega)})$

• rapporto k/N_{max}, dove k è il numero di iterazioni del metodo SOR eseguite.

Commentare i risultati ottenuti per fornire una stima sperimentale dell'intervallo di valori di ω per i quali il metodo SOR converge, e per fornire il valore ottimale ω_{opt} del parametro di rilassamento che massimizza la velocità di convergenza.

(b) Verificare numericamente che $\rho(P_{Gs}) = \rho(P_J)^2$. Determinare quindi la relazione tra le velocità di convergenza dei due metodi e verificarla sperimentalmente usando i programmi `jacobi` e `sor`.

4. Autovalori. Il problema del calcolo degli autovalori di una matrice quadrata A di ordine n consiste nel trovare gli n numeri (possibilmente complessi) λ tali che

$$Ax = \lambda x, \;\; x \neq 0 \tag{4.1}$$

A seconda delle esigenze talvolta è richiesto solamente il calcolo di alcuni autovalori (ad esempio quelli di massimo modulo, per determinare lo spettro della matrice) mentre in altri casi si vogliono determinare tutti gli n autovalori in $\mathbb{C}$.

4.1. Il metodo delle potenze. Il metodo delle potenze è particolarmente indicato per il calcolo dell'autovalore di massimo modulo di una matrice.

Sia A una matrice quadrata di ordine n con n autovettori x_1, ..., x_n linearmente indipendenti e autovalori λ_1, ..., λ_n tali che

$$|\lambda_1| > |\lambda_2| \geq \ldots \geq |\lambda_n|. \tag{4.2}$$

Ricordiamo i seguenti risultati.

1. Una matrice A è diagonalizzabile se e solo se possiede n autovettori linearmente indipendenti.
2. Se tutti gli autovalori di A sono distinti la matrice è diagonalizzabile; l'opposto è ovviamente falso (si pensi alla matrice identica).

3. Una matrice simmetrica (hermitiana) è diagonalizzabile. L'opposto è ovviamente falso, visto che la matrice

$$A = \begin{pmatrix} 15 & 0 \\ 1 & 10 \end{pmatrix} \tag{4.3}$$

è diagonalizzabile visto che ha tutti gli autovalori distinti ma non è simmetrica.

Il metodo delle potenze è definito come segue. Sia $t_0 \in \mathcal{R}^n$ definito da

$$t_0 = \sum_{i=1}^{n} \alpha_i \, x_i, \;\; \alpha_1 \neq 0,$$

e si generi la successione

$$y_0 = t_0 \tag{4.4}$$

$$y_k = A y_{k-1}, \;\; k = 1, 2, \ldots \tag{4.5}$$

TEOREMA 4.1. *Sia A è una matrice quadrata diagonalizzabile avente autovalori λ_k tali che*

$$|\lambda_1| > |\lambda_2| \geq \ldots \geq |\lambda_n|.$$

Siano $u_k \neq 0$ autovettori relativi all'autovalore λ_k, cioè

$$A u_k = \lambda_k u_k.$$

Sia

$$y_0 = \sum_k \alpha_k u_k, \;\;\; \alpha_1 \neq 0.$$

La successione $\{y_s\}$ definita da $y_{s+1} = A y_s$ converge ad un vettore parallelo a x_1 e che il coefficiente di Rayleigh (relativo al prodotto scalare euclideo)

$$\rho(y_s, A) := \frac{(y_s, A y_s)}{(y_s, y_s)} \tag{4.6}$$

converge a λ_1.

NOTA 4.2. *Il metodo converge anche nel caso in cui*

$$\lambda_1 = \ldots = \lambda_r$$

per $r > 1$, tuttavia non è da applicarsi quando l'autovalore di modulo massimo non è unico.

NOTA 4.3. *In virtú di alcune possibili problemi di underflow e underflow si preferisce normalizzare passo passo il vettore y_k definito in (4.4). Conseguentemente l'algoritmo del metodo delle potenze risulta*

$$u_k = A t_{k-1} \tag{4.7}$$

$$t_k = \frac{u_k}{\beta_k}, \;\; \beta_k = \|u_k\|_2 \tag{4.8}$$

$$l_k = \rho(t_k, A) \tag{4.9}$$

dove $\rho(t_k, A)$ è il coefficiente di Rayleigh definito in (4.6).

4.2. Il metodo delle potenze inverse. Una variante particolarmente interessante del metodo delle potenze è particolarmente utile nel caso in cui A sia una matrice quadrata con n autovettori linearmente indipendenti,

$$|\lambda_1| \geq |\lambda_2| \geq \ldots > |\lambda_n| > 0. \tag{4.10}$$

e si desideri calcolare il piú piccolo autovalore in modulo, cioè λ_n, applicando il metodo delle potenze ad A^{-1}. Si ottiene cosí la successione $\{t_k\}$ definita da

$$Au_k = t_{k-1} \tag{4.11}$$

$$t_k = \frac{u_k}{\beta_k},\ \beta_k = \|u_k\|_2 \tag{4.12}$$

e convergente ad un vettore parallelo a x_n. La successione di coefficienti di Rayleigh

$$\rho(t_k, A^{-1}) := \frac{(t_k, A^{-1}t_k)}{(t_k, t_k)} = \frac{(t_k, u_{k+1})}{(t_k, t_k)} \to 1/\lambda_n. \tag{4.13}$$

da cui è immediato calcolare λ_n.

Vediamo in dettaglio questo punto. Se $\{\xi_i\}$ sono gli autovalori di A^{-1} con

$$|\xi_1| > |\xi_2| \geq |\xi_3| \geq \ldots \geq |\xi_n|$$

allora il metodo delle potenze inverse calcola un'approssimazione di ξ_1 e di un suo autoversore x. Si osserva subito che se $A^{-1}x = \xi_i x$ (con $\xi_i \neq 0$) allora moltiplicando ambo membri per ${\xi_i}^{-1}A$ si ottiene leggendo da destra a sinistra $Ax = \xi_i^{-1}x$ cioè ξ_i^{-1} è un autovalore di A e x è non solo autovettore di A^{-1} relativo all'autovalore ξ_i, ma pure autovettore di A relativo all'autovalore ξ_i^{-1}. Conseguentemente se ξ_1 è l'autovalore di massimo modulo di A^{-1} e λ_n è l'autovalore di minimo modulo di A si ha $\lambda_n = \xi_i^{-1}$ e che

$$A^{-1}x = \xi_1 x \Rightarrow Ax = \xi_1^{-1}x = \lambda_n x$$

Notiamo che il metodo delle potenze inverse, calcola $\xi_1 = \lambda_n^{-1}$ e il relativo autovettore x. Per ottenere λ_n viene naturale calcolare ξ_1^{-1}, ma usualmente essendo x autovettore di A relativo a λ_n si preferisce per questioni numeriche calcolare λ_n via coefficente di Rayleigh

$$\rho(x, A) := \frac{(x, Ax)}{(x, x)}.$$

In generale, fissato $\mu \in \mathbb{C}$ è possibile calcolare, se esiste unico, l'autovalore λ più vicino a μ considerando il seguente pseudocodice:

$$(A - \mu I)\, z_k = q_{k-1} \tag{4.14}$$

$$q_k = z_k / \|z_k\|_2 \tag{4.15}$$

$$\sigma_k = (q_k)^H A q_k \tag{4.16}$$

Ricordiamo che se λ è autovalore di A allora

$$Ax = \lambda x \Rightarrow (A - \mu I)x = \lambda x - \mu x = (\lambda - \mu)x$$

e quindi $\lambda - \mu$ è autovalore di $A - \mu I$. Il metodo delle potenze inverse applicato a $A - \mu I$ calcola il minimo autovalore $\sigma = \lambda - \mu$ in modulo di $A - \mu I$ cioè il σ che rende minimo il valore di $|\sigma| = |\lambda_i - \mu|$, dove λ_i sono gli autovalori di A. Quindi essendo $\lambda_i = \sigma_i - \mu$ si ottiene pure il λ_i più vicino a μ.

4.3. Il metodo QR. Sia A una matrice quadrata di ordine n. Utilizzando il metodo di Householder è possibile fattorizzare la matrice A come prodotto di due matrici Q ed R con Q unitaria (cioè $Q^T * Q = Q * Q^T = I$) ed R triangolare superiore.
Citiamo alcune cose:

1. La matrice A ha quale sola particolarità di essere quadrata. Nel caso generale però la sua fattorizzazione QR in generale non è unica bensì determinata a meno di una matrice di *fase*.
2. Tale fattorizzazione non è unica (i segni delle componenti sulla diagonale della matrice A possono essere scelti arbitrariamente). Nel caso sia non singolare, allora tale fattorizzazione è unica qualora si chieda che i coefficienti diagonali di R siano positivi.
3. La routine Matlab `qr` effettua tale fattorizzazione.
4. Se la matrice H è simile a K (cioè esiste una matrice non singolare S tale che $H = S^{-1}KS$) allora H e K hanno gli stessi autovalori. Si può vedere facilmente che la relazione di similitudine è transitiva, cioè se H_1 è simile ad H_2 e H_2 è simile ad H_3 allora H_1 è simile ad H_3.

Il metodo QR venne pubblicato indipendemente nel 1961 da Francis e da Kublanovskaya e successivamente implementato in EISPACK. Ci limiteremo a considerare versioni di base del metodo.

Sia

$$A_0 = A = Q_0 R_0$$

e si ponga

$$A_1 := R_0 Q_0.$$

Poichè

$$Q_0 A_1 Q_0^T = Q_0 A_1 Q_0^T = Q_0 R_0 Q_0 Q_0^T = A_0$$

la matrice A_1 è simile ad A_0 (si ponga $S = Q_0^{-1} = Q_0^T$) e quindi ha gli stessi autovalori. Sia quindi in generale

$$A_k = Q_k R_k$$

$$A_{k+1} = R_k Q_k.$$

Per le stesse motivazioni A_{k+1} è simile ad A_k, e per transitività ad A_0. Quindi A_{k+1} ha gli stessi autovalori di A_0.

Per la convergenza del metodo esistono vari risultati. Ricordiamo principalmente

TEOREMA 4.4. *Se la matrice $A \in \mathbb{R}^{n \times n}$ è regolare e con autovalori tutti distinti in modulo, con*

$$|\lambda_1| > \ldots > |\lambda_n| \tag{4.17}$$

allora l'algoritmo QR converge ad una matrice A_∞ triangolare superiore. Se la matrice è simmetrica, allora

$$A_\infty = diag(\lambda_1, \ldots, \lambda_1).$$

Inoltre se A è una matrice Hessenberg superiore allora l'algoritmo QR converge ad una matrice A_∞ triangolare a blocchi, simile ad A e con gli autovalori di ogni blocco diagonale tutti uguali in modulo.

Alcuni dettagli.

1. Nelle implementazioni si calcola con un metodo scoperto da Householder (ma esiste un metodo alternativo dovuto a Givens) una matrice di Hessenberg T

$$T = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & \dots & a_{1,n} \\ a_{2,1} & a_{2,2} & a_{2,3} & \dots & a_{2,n} \\ 0 & a_{3,2} & a_{3,3} & \dots & a_{3,n} \\ 0 & 0 & a_{4,3} & \dots & a_{4,n} \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & a_{n,n-1} & a_{n,n} \end{pmatrix}$$

   simile ad A ed in seguito si applica il metodo QR relativamente alla matrice T. Se A è simmetrica la matrice T risulta tridiagonale simmetrica. In entrambi i casi le iterazioni mantengono la struttura, cioè se $A_0 = T$ è di Hessenberg, allora A_k è di Hessenberg, se $A_0 = T$ è tridiagonale allora A_k è tridiagonale.

2. Per inciso, si può dimostrare che la velocità di convergenza dipende dal rapporto

$$\rho := \max_{1 \leq i \leq n-1} \max \frac{|\lambda_{i+1}|}{|\lambda_i|}. \tag{4.18}$$

   Il numero di moltiplicazioni necessarie all'algoritmo di Givens per calcolare tale matrice T a partire da A è approssimativamente $10n^3/3$ mentre per quanto riguarda l'algoritmo di Householder è $5n^3/3$. Il metodo QR applicato ad una matrice A in forma di Hessenberg superiore ha ad ogni passo un costo di $2n^2$ operazioni moltiplicative.

3. Se la condizione (4.17) non è verificata si può dimostrare che la successione $\{A_k\}$ tende a una forma triangolare a blocchi.

4.4. Implementazione Matlab di alcuni metodi per il calcolo degli autovalori. Partiamo con una versione semplice `power_basic` del metodo delle potenze

```
function [lambda1, x1, niter, err]=power_basic(A,z0,toll,nmax)

% INPUT:
% A    : MATRICE DI CUI VOGLIAMO CALCOLARE L'AUTOVALORE DI MASSIMO MODULO.
% z0   : VETTORE INIZIALE (NON NULLO).
% toll: TOLLERANZA.
% nmax: NUMERO MASSIMO DI ITERAZIONI.
%
% OUTPUT:
% lambda1 : VETTORE DELLE APPROSSIMAZIONI DELL'AUTOVALORE DI MASSIMO MODULO.
% x1      : AUTOVETTORE RELATIVO ALL'AUTOVALORE DI MASSIMO MODULO.
% niter   : NUMERO DI ITERAZIONI.
% err     : VETTORE DEI RESIDUI PESATI RELATIVI A "lambda1".
%
% TRATTO DA QUARTERONI-SALERI, "MATEMATICA NUMERICA", p. 184.
%
```

```
q=z0/norm(z0); q2=q; err=[]; lambda1=[];
res=toll+1; niter=0; z=A*q;
while (res >= toll & niter <= nmax)
    q=z/norm(z); z=A*q; lam=q'*z; x1=q;
    z2=q2'*A; q2=z2/norm(z2); q2=q2'; y1=q2; costheta=abs(y1'*x1);
    niter=niter+1; res=norm(z-lam*q)/costheta;
    err=[err; res]; lambda1=[lambda1; lam];
end
```

Qualche nota

1. il vettore iniziale `z0` e' normalizzato ed in `err`, `lambda1` vengono memorizzati rispettivamente i valori dell'errore compiuto e dell'autovalore di massimo modulo $\lambda_{\max}$;
2. l'assegnazione `res=toll+1;` forza l'algoritmo ad entrare nel ciclo `while`, mentre `z=A*q;` è una quantità da utilizzarsi per il calcolo dell'autovalore $\lambda_{\max}$;
3. nel ciclo `while`, `q` è un'approssimazione di un autoversore relativo a $\lambda_{\max}$, mentre `lam` di $\lambda_{\max}$;
4. il ciclo si interrompe se un numero massimo di iterazioni `niter` è raggiunto oppure

$$\frac{||Aq^k - \lambda^k||_2}{|\cos(\theta_{\lambda_k})|} < \text{tol}$$

dove θ_{λ_k} è l'angolo formato tra (un'approssimazione del)l'autovalore destro `x1` e sinistro `y1` associati a `lam`

4.4.1. Esempio 1. Testiamo il codice relativamente al calcolo dell'autovalore di massimo modulo di

$$\begin{aligned} A &= \begin{pmatrix} -15.5 & 7.5 & 1.5 \\ -51 & 25 & 3 \\ -25.5 & 7.5 & 11.5 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 2 & 3 \\ 2 & 5 & 6 \\ 7 & 9 & 3 \end{pmatrix} \cdot \begin{pmatrix} 10 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 \\ 2 & 5 & 6 \\ 7 & 9 & 3 \end{pmatrix}^{-1} \end{aligned} \tag{4.19}$$

La matrice A è diagonalizzabile e ha autovalori $10, 10, 1$. Si può vedere che una base di autovettori relativa agli autovalori $10, 10, 1$ è composta da $(1, 2, 7)$, $(2, 5, 9)$, $(3, 6, 3)$. Quale vettore iniziale del metodo delle potenze consideriamo

$$z_0 = (1, 1, 1) = (7/6) \cdot (1, 2, 7) - 1 \cdot (2, 5, 9) + (11/18) \cdot (3, 6, 3)$$

e quindi il metodo delle potenze applicato ad A, e avente quale punto iniziale z_0 può essere utilizzato per il calcolo dell'autovalore di massimo modulo di A, poichè $\alpha_1 = 7/6 \neq 0$. Dalla shell di Matlab/Octave:

```
>> S=[1 2 3; 2 5 6; 7 9 3];
>> D=diag([10 10 1]);
>> A=S*D*inv(S)
A =
  -15.5000    7.5000    1.5000
```

```
  -51.0000   25.0000    3.0000
  -25.5000    7.5000   11.5000
>> z0=[1 1 1]';
>> toll=10^(-8);
>> nmax=10;
>> format short e;
>> [lambda1, x1, niter, err]=power_basic(A,z0,toll,nmax)
lambda1 =
  1.1587e+001
  1.0138e+001
  1.0014e+001
  1.0001e+001
  1.0000e+001
  1.0000e+001
  1.0000e+001
  1.0000e+001
  1.0000e+001
  1.0000e+001
x1 =
 -2.8583e-001
 -9.1466e-001
 -2.8583e-001
niter =
    10
err =
  2.2466e+000
  2.1028e-001
  2.0934e-002
  2.0925e-003
  2.0924e-004
  2.0924e-005
  2.0924e-006
  2.0924e-007
  2.0924e-008
  2.0924e-009
>>
```

La convergenza è abbastanza veloce come si vede dalla quantità `err`, che consiste in un particolare residuo pesato.

Una questione sorge spontanea. Cosa sarebbe successo se avessimo utilizzato l'algoritmo senza normalizzazione come ad esempio `power_method` definito da

```
function [lambda,v]=power_method(A,x0,maxit)

v=x0;

for index=1:maxit
    v_old=v;
    v=A*v_old;
    lambda=(v_old'*v)/(v_old'*v_old);
end
```

Proviamo il test, facendo iterare il metodo prima 5, poi 100 volte e alla fine 1000 volte (si noti il settaggio della variabile `maxit` relativa al numero di iterazioni da compiere):

```
>> x0=[1 1 1]'
x0 =
     1
     1
     1
>> A=[-15.5 7.5 1.5; -51 25 3; -25.5 7.5 11.5]
A =
  -15.5000    7.5000    1.5000
  -51.0000   25.0000    3.0000
  -25.5000    7.5000   11.5000
>> [lambda,v]=power_method(A,x0,5)
lambda =
   10.0014
v =
  1.0e+005 *
   -0.8333
   -2.6666
   -0.8333
>> [lambda,v]=power_method(A,x0,100)
lambda =
   10.0000
v =
  1.0e+100 *
   -0.8333
   -2.6667
   -0.8333
>> [lambda,v]=power_method(A,x0,1000)
lambda =
   NaN
v =
   NaN
   NaN
   NaN
>>
```

La ragione è semplice. Per k relativamente piccolo si ha $A \cdot t_k \approx 10 \cdot t_k$ e quindi per $s \geq k$

$$t_s \approx A^{s-k} \cdot t_k \approx 10 \cdot A^{s-k-1} \cdot t_k \approx \cdots \approx 10^{s-k} \cdot t_k$$

da cui

$$\|t_s\|_2 \approx 10^{s-k} \cdot \|t_k\|_2$$

spiegando quindi perchè si possano avere problemi di overflow applicando l'algoritmo di base.

4.4.2. Esempio 2. Proviamo un test diverso, questa volta con la matrice (diagonalizzabile)

$$A = \begin{pmatrix} 1 & 2 \\ 0 & -1 \end{pmatrix},$$

avente autovalori $\lambda_1 = 1$ e $\lambda_2 = -1$ e autovettori linearmente indipendenti (1, 0), $(-1, 1)$. Quale vettore iniziale poniamo

$$x_0 = (1, 3) = 4 \cdot (1, 0) + 3 \cdot (-1, 1)$$

e quindi il metodo delle potenze applicato ad A, partendo da x_0 può essere sicuramente applicato. D'altra parte dubitiamo converga in quanto $|\lambda_1| = |\lambda_2| = 1$ pur essendo $\lambda_1 \neq \lambda_2$. Dalla shell di Matlab/Octave:

```
>> A=[1 2; 0 -1]
A =
     1     2
     0    -1
>> [lambda1, x1, niter, err]=power_basic(A,[1; 3],10^(-8),15)
lambda1 =
 -3.4483e-002
 -2.0000e-001
 -3.4483e-002
 -2.0000e-001
 -3.4483e-002
 -2.0000e-001
 -3.4483e-002
 -2.0000e-001
 -3.4483e-002
 -2.0000e-001
 -3.4483e-002
 -2.0000e-001
 -3.4483e-002
 -2.0000e-001
 -3.4483e-002
 -2.0000e-001
x1 =
  3.1623e-001
  9.4868e-001
niter =
    16
err =
  4.4567e-001
  2.4000e+000
  4.4567e-001
  2.4000e+000
  4.4567e-001
  2.4000e+000
  4.4567e-001
  2.4000e+000
  4.4567e-001
  2.4000e+000
  4.4567e-001
  2.4000e+000
  4.4567e-001
  2.4000e+000
  4.4567e-001
  2.4000e+000
```

```
>>
```

Dal residuo pesato è chiaro che il metodo non converge, e come anticipato il motivo è la presenza di autovalori distinti aventi modulo massimo.

4.4.3. Esempio 3. Per terminare, vediamo il caso della matrice diagonalizzabile (avendo autovalori distinti)

$$A = \begin{pmatrix} 1 & 2 \\ 0 & 10 \end{pmatrix},$$

in cui il metodo funziona rapidamente, in quanto esiste un solo autovalore di modulo massimo, uguale a 10.

```
>> A=[1 2; 0 10]
A =
     1     2
     0    10
>> [lambda1, x1, niter, err]=power_basic(A,[1; 3],10^(-8),15)
lambda1 =
  9.9779e+000
  9.9979e+000
  9.9998e+000
  1.0000e+001
  1.0000e+001
  1.0000e+001
  1.0000e+001
  1.0000e+001
x1 =
  2.1693e-001
  9.7619e-001
niter =
     8
err =
  9.6726e-002
  9.7529e-003
  9.7610e-004
  9.7618e-005
  9.7619e-006
  9.7619e-007
  9.7619e-008
  9.7619e-009
```

Si osservi che il metodo termina in quanto l'errore pesato `err` è minore della tolleranza $toll = 10^{-8}$.

4.5. Il metodo delle potenze inverse. Una versione di base `invpower` del metodo delle potenze inverse è

```
function [lambda, x, niter, err]=invpower(A,z0,mu,toll,nmax)

% DATO UN VALORE mu, SI CALCOLA L'AUTOVALORE "lambda_mu" PIU' VICINO A mu.
```

```
% INPUT:
% A    : MATRICE DI CUI VOGLIAMO CALCOLARE L'AUTOVALORE "lambda_mu".
% z0   : VETTORE INIZIALE (NON NULLO).
% mu   : VALORE DI CUI VOGLIAMO CALCOLARE L'AUTOVALORE PIU' VICINO.
% toll: TOLLERANZA.
% nmax: NUMERO MASSIMO DI ITERAZIONI.
%
% OUTPUT:
% lambda : VETTORE DELLE APPROSSIMAZIONI DELL'AUTOVALORE DI MINIMO MODULO.
% x      : AUTOVETTORE RELATIVO ALL'AUTOVALORE DI MINIMO MODULO.
% niter  : NUMERO DI ITERAZIONI.
% err    : VETTORE DEI RESIDUI PESATI RELATIVI A "lambda".
%
% TRATTO DA QUARTERONI-SALERI, "MATEMATICA NUMERICA", p. 184.
%

n=max(size(A)); M=A-mu*eye(n); [L,U,P]=lu(M);
q=z0/norm(z0); q2=q'; err=[]; lambda=[];
res=toll+1; niter=0;
while (res >= toll & niter <= nmax)
    niter=niter+1; b=P*q; y=L\b; z=U\y;
    q=z/norm(z); z=A*q; lam=q'*z;
    b=q2'; y=U'\b; w=L'\y;
    q2=(P'*w)'; q2=q2/norm(q2); costheta=abs(q2*q);
    if (costheta > 5e-2)
        res=norm(z-lam*q)/costheta; err=[err; res]; lambda=[lambda; lam];
    else
        disp('\n \t [ATTENZIONE]: AUTOVALORE MULTIPLO'); break;
    end
    x=q;
end
```

Forniamo ora alcune spiegazioni del codice in `invpower`.

1. Per risolvere il sistema lineare in 4.14, si effettua una fattorizzazione $PM = LU$ della matrice $M = A - \mu I$;
2. All'interno del ciclo while, nella prima riga si calcola z_k, mentre nella successiva un suo versore q_k, e σ_k è immagazzinato in `lam`;
3. Similmente al metodo diretto si effettua il prodotto scalare di un'autovalore sinistro con uno destro.

4.5.1. Esempio 1. Applichiamo il metodo delle potenze inverse per il calcolo dell'autovalore più piccolo in modulo della matrice

$$\begin{aligned} A &= \begin{pmatrix} -15.5 & 7.5 & 1.5 \\ -51 & 25 & 3 \\ -25.5 & 7.5 & 11.5 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 2 & 3 \\ 2 & 5 & 6 \\ 7 & 9 & 3 \end{pmatrix} \cdot \begin{pmatrix} 10 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 \\ 2 & 5 & 6 \\ 7 & 9 & 3 \end{pmatrix}^{-1} \end{aligned} \tag{4.20}$$

Come visto la matrice A è quindi diagonalizzabile, ha autovalori $10, 10, 1$ e relativi autovettori è $(1, 2, 7)$, $(2, 5, 9)$, $(3, 6, 3)$ formanti una base di $\mathbb{R}^3$. Quale vettore iniziale del metodo delle

potenze consideriamo

$$z_0 = (1, 1, 1) = (7/6) \cdot (1, 2, 7) - 1 \cdot (2, 5, 9) + (11/18) \cdot (3, 6, 3)$$

e quindi il metodo delle potenze inverse applicato ad A, e avente quale punto iniziale z_0 può essere utilizzato per il calcolo dell'autovalore di minimo modulo di A.

```
>> z0=[1;1;1]; mu=0; toll=10^(-8); nmax=10;
>> A=[-15.5 7.5 1.5; -51 25 3; -25.5 7.5 11.5]
A =
 -15.50000000000000   7.50000000000000   1.50000000000000
 -51.00000000000000  25.00000000000000   3.00000000000000
 -25.50000000000000   7.50000000000000  11.50000000000000
>> [lambda, x, niter, err]=invpower(A,z0,mu,toll,nmax)
lambda =
   0.39016115351993
   0.94237563941268
   0.99426922936854
   0.99942723776656
   0.99994272692315
   0.99999427272378
   0.99999942727270
   0.99999994272728
   0.99999999427273
x =
   0.40824829053809
   0.81649658085350
   0.40824829053809
niter =
     9
err =
   0.81535216507377
   0.08358101289062
   0.00838126258396
   0.00083836078891
   0.00008383842712
   0.00000838386620
   0.00000083838685
   0.00000008383868
   0.00000000838387
>>
```

La convergenza è lineare (come si intuisce dalle approssimazioni contenute nel vettore `lambda`).

Per vederlo, dalla shell di Matlab/Octave calcoliamo l'errore assoluto/relativo relativo all'autovalore 1:

```
>> s=1-lambda
s =
   0.60983884648007
   0.05762436058732
   0.00573077063146
   0.00057276223344
   0.00005727307685
   0.00000572727622
```

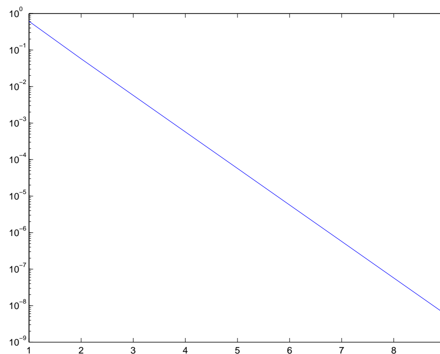

FIGURA 4.1. *Grafico che illustra la convergenza lineare del metodo delle potenze inverse nell'esempio 1.*

```
   0.00000057272730
   0.00000005727272
   0.00000000572727
>> semilogy(1:length(s),s)
```

generando il grafico in scala semi-logaritmica in figura che evidentemente sottolinea la convergenza lineare.

4.6. Il metodo QR. Una versione di base del metodo QR è la seguente. Si salvi il files `houshess.m` che implementa la trasformazione per similitudine di A in una matrice di Hessenberg

```
function [H,Q]=houshess(A)

% REDUCTION OF A MATRIX TO A SIMILAR HESSENBERG ONE.
% SEE QUARTERONI, SACCO, SALERI P. 192.

n=max(size(A)); Q=eye(n); H=A;

for k=1:(n-2)
    [v,beta]=vhouse(H(k+1:n,k)); I=eye(k); N=zeros(k,n-k);
    m=length(v); R=eye(m)-beta*v*v'; H(k+1:n,k:n)=R*H(k+1:n,k:n);
    H(1:n,k+1:n)=H(1:n,k+1:n)*R; P=[I,N; N',R]; Q=Q*P;
end
```

ove `vhouse.m` è definito da

```
function [v,beta]=vhouse(x)

% BUILDING HOUSEHOLDER VECTOR.
% SEE QUARTERONI, SACCO, SALERI P. 197.

n=length(x); x=x/norm(x); s=x(2:n)'*x(2:n); v=[1; x(2:n)];
if (s==0)
    beta=0;
else
    mu=sqrt(x(1)^2+s);
    if (x(1) <= 0)
```

```
        v(1)=x(1)-mu;
    else
        v(1)=-s/(x(1)+mu);
    end
    beta=2*v(1)^2/(s+v(1)^2);
    v=v/v(1);
end
```

quindi `QRbasicmethod.m` che calcola la matrice triangolare T relativa a QR:

```
function [T,hist]=QRbasicmethod(T_input,maxit)

% QR METHOD FOR A SYMMETRIC TRIDIAGONAL MATRIX "T_input".

T=T_input;
hist=sort(diag(T));

for index=1:maxit
    [Q,R]=qr(T);
    T=R*Q;                          % NEW SIMILAR MATRIX.
    hist=[hist sort(diag(T))]; % IT STORES THE DIAGONAL ELEMENTS
                                    % OF THE "index" ITERATION.
end
```

4.7. Esercizio: Calcolo degli zeri di un polinomio, via autovalori della matrice compagna. Si può dimostrare che gli zeri del polinomio monico di grado n

$$p(t) = c_0 + c_1 t + \cdots + c_{n-1} t^{n-1} + t^n$$

sono gli autovalori della matrice compagna (o equivalentemente della sua trasposta visto che $eig(A) = eig(A^T)$)

$$C = \begin{bmatrix} 0 & 0 & \dots & 0 & -c_0 \\ 1 & 0 & \dots & 0 & -c_1 \\ 0 & 1 & \dots & 0 & -c_2 \\ \vdots & \vdots & \dots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & -c_{n-1} \end{bmatrix}.$$

Sfruttando il comando `eig`, e osservando che

```
>> A=[1 2; 3 4]

A =

     1     2
     3     4

>> B=[5 6; 7 8]

B =

     5     6
     7     8
```

```
>> C=[A B]

C =

     1     2     5     6
     3     4     7     8

>> D=[A; B]

D =

     1     2
     3     4
     5     6
     7     8

>>
```

scrivere un codice Matlab che risolva un'equazione algebrica calcolando gli autovalori della matrice compagna associata. Effettuare quindi un test su un polinomio di cui si conoscono gli zeri e valutare la bontà del proprio codice. Di seguito si calcolino gli zeri di modulo massimo e minimo usando il metodo delle potenze e la sua variante inversa, come pure il metodo QR (se applicabile).