Lab. 1 - Matrici e sistemi lineari

Laboratorio di Calcolo Numerico

Annalisa Buffa   Fausto Cavalli

Corso di Laurea in Chimica

A.A. 2009-2010

Matrici

Le matrici vanno assegnate elencando gli elementi per riga: per definire la matrice

$$A = \begin{pmatrix} 1 & 5 & -4 \\ 3.04 & 0 & 7 \end{pmatrix}$$

basta digitare

```
>> A=[1 5 -4;3.04 0 7]
```

Attenzione!

Il numero di elementi su ogni riga deve essere lo stesso altrimenti Matlab genera un messaggio di errore

Per accedere agli elementi di una matrice è sufficiente indicare fra tonde gli indici di riga e di colonna dell'elemento che si vuole richiamare

matrice(riga,colonna)

Comandi che riguardano gli array

Array

In Matlab, scalari, vettori e matrici sono variabili dello stesso tipo dette **array**: gli scalari hanno dimensione 1×1, i vettori colonna $n \times 1$ mentre quelli riga $1 \times n$, le matrici hanno dimensione $m \times n$)

Per conoscere la dimensione di un array (e quindi in particolare di una matrice) si usa il comando **size** la cui sintassi è

$$[righe,colonne] = \texttt{size}(matrice)$$

Matlab offre alcuni comandi per la creazioni di particolari array (per un elenco completo `help elmat`).
Per generare un array nullo si usa il comando **zeros(righe,colonne)**

```
>> B=zeros(2,3)
B =
     0     0     0
     0     0     0
```

mentre per creare un array in cui tutti gli elementi valgano 1 si usa il comando analogo **ones(righe,colonne)**

Comandi che riguardano gli array

Attenzione!

Sia `zeros` sia `ones` possono creare vettori o matrici. Se vengono richiamati con un solo valore in parentesi **creano una matrice quadrata** e **non un vettore**. Per esempio

```
>> zeros(2)
ans =
    0       0
    0       0
```

Per creare la matrice identità si usa il comando `eye(n)`

```
>> eye(2)
ans =
    1       0
    0       1
```

Operazioni con le matrici

La prima operazione che si può fare è la **trasposizione**, che scambia le righe e le colonne e si realizza aggiungendo `'` a ciò che si vuole trasporre. Tutte le operazioni lecite dell'algebra lineare quali somma, sottrazione, prodotto per uno scalare e prodotto riga per colonna sono possibili anche in matlab usando le operazioni usuali **+,-,*** (senza punto!). Si ricorda che il prodotto è poi definito tra matrici di dimensione compatibile, cioè posso moltiplicare A e B se le colonne di A sono tante quante le righe di B. Si ricorda che questo prodotto non è commutativo, come mostra il seguente esempio

```
>> A=[1 2;3 4];
>> B=[-1 -3;0 2];
>> A*B
ans =
     -1      1
     -3     -1
>> B*A
ans =
     -10    -14
       6      8
```

Operazioni componente per componente

Come già visto per i vettori, esistono anche le operazioni componente per componente, che agiscono tra matrici di uguale dimensione, come per il **prodotto** `.*`

```
>> A=[1 2 -1;-2 0 3];
>> B=[1 4 2;1 2 3];
>> A.*B   % in questo caso il prodotto * non è  definito
ans =
    1      8     -2
   -2      0      9
```

e la **divisione** `./`,

```
>> A=[1 2 -1;-2 0 3];
>> B=[1 4 2;1 2 3];
>> A./B
ans =
   1.0000    0.5000   -0.5000
  -2.0000         0    1.0000
```

Analogo discorso per **l'elevamento a potenza** `.^`

Operazioni array

Riepilogo operazioni che coinvolgono array

Op.	Oggetto	Oggetto	Azione	Risultato
+;-	Scalare	Array $m \times n$	componentwise	Array $m \times n$
+;-	Array $m \times n$	Array $m \times n$	componentwise	Array $m \times n$
*	Scalare	Array $m \times n$	componentwise	Array $m \times n$
*	Array $m \times n$	Array $n \times p$	righe/colonne	Array $m \times p$
^	Array $n \times n$	Scalare	righe/colonne	Array $n \times n$
.*;./	Array $m \times n$	Array $m \times n$	componentwise	Array $m \times n$
.^	Array $m \times n$	Scalare	componentwise	Array $m \times n$
.^	Array $m \times n$	Array $m \times n$	componentwise	Array $m \times n$

Sistemi lineari

Sistemi lineari

Un sistema lineare di n equazioni in n incognite si può scrivere in forma matriciale come $Ax = b$, dove A è una matrice $n \times n$ che contiene i coefficienti del sistema, b è il vettore colonna dei termini noti, x è il vettore colonna delle incognite.

Per esempio per il sistema

$$\begin{cases} 2x_1 - 3x_2 = 0 \\ 3x_1 - 2x_2 + 4x_3 = -1 \\ x_2 - x_3 = 1 \end{cases}$$

vale che

$$A = \begin{pmatrix} 2 & -3 & 0 \\ 3 & -2 & 4 \\ 0 & 1 & -1 \end{pmatrix} \quad x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad b = \begin{pmatrix} 0 \\ -1 \\ 1 \end{pmatrix}$$

Sistemi lineari

In analisi numerica non si procede **mai** con l'invertire la matrice dei coefficienti per risolvere un sistema lineare, essendo ciò molto costoso in termini di operazioni ed estremamente delicato in termini di propagazione degli errori.
Ci sono due categorie di approcci che si possono usare per risolvere un sistema lineare: i **metodi diretti**, che costruiscono la soluzione in un numero finito di passaggi e che si basano sul metodo di eliminazione di Gauss, ed i **metodi iterativi**, dove la soluzione è il limite di una successione di approssimazioni successive.

Ciascun approccio ha i suoi pregi e difetti ed i suoi campi di applicazione: in generale i metodi diretti funzionano bene per matrici piene (cioè con pochi elementi nulli), mentre quelli iterativi sono adatti a trattare matrici sparse, cioè in cui il numero degli elementi non nulli è preponderante rispetto a quello degli elementi nulli.

Matrici piene e sparse

Esempio

L'equazione di Laplace è una equazione molto importante della fisica matematica che descrive, per esempio, la distribuzione di un potenziale di una forza. Per risolverla numericamente su un dominio rettangolare, si può introdurre una scacchiera di punti in cui si cerca la soluzione. Il tutto si riduce alla fine a dover risolvere un sistema lineare: se per esempio la scacchiera era costituita da 100x100 punti (che in genere sono un numero ragionevole), la matrice del sistema ha 10000 righe e 10000 colonne! Tuttavia questa matrice ha una struttura particolare: solo pochi elementi per ogni riga e colonna sono diversi da zero. Se si costruisce questa matrice con matlab e la si memorizza come piena, si scopre che la memoria necessaria è di circa 763 Megabytes, mentre se la si memorizza come sparsa, occuperà solo 831 Kylobytes, avendo in tutto solo circa 50000 elementi non nulli su 10^8!

Metodi diretti per sistemi lineari

Matlab offre un'operazione apposita per la soluzione dei sistemi lineari con metodi diretti, indicata con $\boxed{\backslash}$: il sistema $Ax = b$ si traduce in $x = A\backslash b$. Quindi per risolvere il sistema precedente è sufficiente scrivere

```
>> A=[2 -3 0;3 -2 4;0 1 -1];
>> b=[0;-1;1];
>> x=A\b
```

Attenzione

Bisogna prestare attenzione alle dimensioni della matrice e del vettore dei termini noti, che devono avere lo stesso numero di righe. In particolare bisogna ricordarsi che il **termine noto** deve essere un **vettore colonna**

Metodi diretti per sistemi lineari

Molti metodi diretti permettono di fattorizzare la matrice dei coefficienti, cioè di riscriverla come prodotto di matrici con una struttura più semplice. Questo in genere si traduce nel poter riscrivere il sistema di partenza come sistemi più semplici da risolvere.
La fattorizzazione che deriva dall'eliminazione di Gauss e la cosiddetta **fattorizzazione LU**, che riscrive la matrice A come prodotto di una matrice triangolare inferiore (L) ed una superiore (U). Si ha quindi che

$$Ax = b \Leftrightarrow LUx = b \Leftrightarrow \begin{cases} Ly = b \\ Ux = y \end{cases}$$

Il sistema di partenza si traduce quindi in due sistemi con matrici dei coefficienti triangolari, molto veloci da risolvere.

Nota

Se si hanno da risolvere più sistemi con la stessa matrice dei coefficienti e diversi termini noti, la fattorizzazione LU è molto comoda, dato che può essere fatta una volta per tutte, ripetendo solo la soluzione dei sistemi triangolari finali, che ha un costo molto inferiore rispetto alla procedura completa

Metodi diretti per sistemi lineari

Matlab offre il comando **lu** per effettuare la fattorizzazione LU di una matrice A : la sintassi è

$$[L,U] = \texttt{lu}(A)$$

Attenzione

In realtà il comando `lu` cerca di riorganizzare il sistema di partenza modificando l'ordine delle equazioni in modo che la soluzione della fattorizzazione la meno affetta da errori di cancellazione dovuti all'artimetica floating point. Questa strategia è chiamata **pivoting** e consiste proprio nello scambiare righe o colonne del sistema attraverso la moltiplicazione con matrici particolari, dette matrici di permutazione. Potrebbe capitare quindi che L **non** sia una matrice triangolare inferiore, anche se nei casi che prenderemo in esame ciò non succederà.

Metodi diretti per sistemi lineari

Esempio

Si risolva attraverso la fattorizzazione LU il seguente sistema lineare

$$A = \begin{pmatrix} 4 & -2 & 1 \\ 1 & 3 & 2 \\ 2 & 1 & 2 \end{pmatrix} \quad b = \begin{pmatrix} 3 \\ 6 \\ 5 \end{pmatrix}$$

Si può procedere così

```
>> A=[4 -2 1;1 3 2;2 1 2];
>> [L,U]=lu(A);
>> y=L\b;
>> x=U\y
```

Metodi diretti per sistemi lineari

Esempio

Si costruisca ora la matrice B usando i seguenti comandi

```
>> n=15;
>> e=ones(n,1);
>> B=4*diag(e);
>> B(1,:)=0.1;
>> B(:,1)=0.1;
```

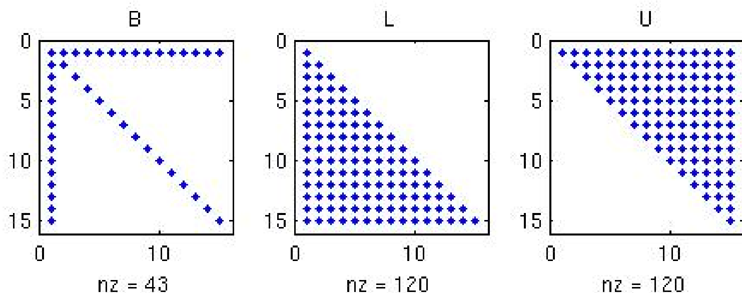
Si effettui la fattorizzazione LU e si analizzi la struttura delle matrici B, L, U con il comando `spy`. Cosa si può osservare?

Una volta costruita la matrice B , si proceda con

```
>> [L,U]=lu(B);
>> figure(1)
>> spy(B);  % spy disegna la struttura della matrice
>> figure(2)% indicando gli elementi non nulli
>> spy(L);
>> figure(3)
>> spy(U);
```

Metodi diretti per sistemi lineari

I grafici che si ottengono mostrano come anche se la matrice B di partenza è quasi tutta nulla, le matrici L, U della fattorizzazione hanno elementi non nulli che complessivamente vanno a riempire ogni posizione



Con matrici con la stessa struttura di questa ma dimensioni molto maggiori sono decisamente più convenienti (soprattutto ai fini di un utilizzo razionale della memoria) metodi iterativi.

Metodo di Jacobi

Come già accennato nei metodi iterativi si costruisce una successione di approssimazioni della soluzione che converga alla soluzione del sistema. Il metodo iterativo più semplice è il metodo di **Jacobi,** che consiste nelle iterazioni

$$x_i^{k+1} = \frac{1}{A_{ii}}(b_i - \sum_{j=1, j\neq i}^{n} A_{ij} x_j^k), \quad i = 1, \ldots, n$$

Il metodo può anche essere riscritto convenientemente nella forma

$$x_i^{k+1} = x_i^k + \frac{1}{A_{ii}}(b_i - \sum_{j=1}^{n} A_{ij} x_j^k), \quad i = 1, \ldots, n$$

La quantità in parentesi è detto residuo e viene indicato con r_k. In generale, quando non si sa la soluzione esatta e si vuole avere una stima dell'errore, si può usare il residuo, che fornisce un criterio anche per un test di arresto per i metodi iterativi: si fermano le iterazioni quando

$$\frac{||r_k||}{||b||} \leq toll$$

Metodo di Jacobi

Esercizio

Si scriva una `function` di nome `jacobi` che, prendendo la matrice dei coefficienti del sistema, il vettore dei termini noti, una stima iniziale della soluzione, una tolleranza ed il numero massimo di iterazioni, restituisca l'approssimazione della soluzione del sistema $Ax = b$ attraverso il metodo di Jacobi ed il numero delle iterazioni effettuate dal metodo.

Suggerimento

Scrivere l'intestazione della `function` usando le variabili in input

- `A` per la matrice dei coefficienti A
- `b` per il vettore dei termini noti b
- `x0` per la stima iniziale
- `toll` per la tolleranza
- `nmax` per il numero massimo di iterazioni

ed in output

- `x` per l'approssimazione della soluzione
- `nit` per il numero di iterazioni

Metodo di Jacobi

Fase di inizializzazione

- Si inizializzi `nit=0;`
- Si inizializzi `n` delle equazioni del sistema, pari alla lunghezza di `b`
- Si ponga `x=x0;`
- Si calcoli il residuo iniziale `r=b-A*x;`
- Si calcoli l'approssimazione iniziale dell'errore `err` pari a `norm(r)/norm(b)`

Ciclo principale: ripetere finchè `nit<nmax` e `err>toll`

- Devo fare un ciclo su tutte le equazioni del sistema: per `i` che va da 1 fino a `n`
    - Calcolo il nuovo valore della `i`-esima componente della soluzione con la formula `xn(i)=x(i)+1/A(i,i)*r(i)`
- Pongo `x=xn';`
- Si incrementi di 1 il numero delle iterazioni `nit`
- Si aggiorni con la stessa formula il valore del residuo `r`
- Si aggiorni con la stessa formula il valore dell'errore `err`

Metodo di Jacobi

Esercizio

Si applichi il metodo di Jacobi ai due sistemi le cui matrici sono date da

$$A = \begin{pmatrix} 1 & -2 & 2 \\ -1 & 1 & -1 \\ -2 & -2 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 2 & -1 & 1 \\ 2 & 2 & 2 \\ -1 & -1 & 2 \end{pmatrix}$$

e il cui termine noto è

$$b = \begin{pmatrix} 1 \\ 2 \\ 0 \end{pmatrix}$$

Si considerino $x_0 = [0, 0, 0]^T$, $toll = 1e - 6$, $nmax = 100$.
Quali considerazioni si possono fare sui due esempi?
Si provino a calcolare le soluzioni "esatte" dei sistemi con il comando di matlab e si calcoli l'errore, verificando la previsione d'errore basata sul residuo.

Metodo di Gauss-Seidel

Un altro metodo iterativo semplice è il metodo di **Gauss-Seidel,** dove, a differenza del metodo di Jacobi, i valori della soluzione già calcolati vengono subito utilizzati. L'iterazione del metodo di Gauss-Seidel è la seguente

$$x_i^{k+1} = \frac{1}{A_{ii}}(b_i - \sum_{j=1}^{i-1} A_{ij} x_j^{k+1} - \sum_{j=i+1}^{n} A_{ij} x_j^{k}), \quad i = 1, \ldots, n$$

L'obiettivo è quello di scrivere una function che implementi il metodo di Gauss-Seidel: a tal proposito, dovremo calcolare le somme che di volta in volta compaiono nella formula, per cui sarà opportuno riscriverla come

$$\begin{aligned} s_1 &= \sum_{j=1}^{i-1} A_{ij} x_j^{k+1} \\ s_2 &= \sum_{j=i+1}^{n} A_{ij} x_j^{k} \\ x_i^{k+1} &= \frac{1}{A_{ii}}(b_i - s_1 - s_2), \quad i = 1, \ldots, n \end{aligned}$$

Metodo di Gauss-Seidel

Esercizio

Si scriva una `function` di nome `gseidel` che, prendendo la matrice dei coefficienti del sistema, il vettore dei termini noti, una stima iniziale della soluzione, una tolleranza ed il numero massimo di iterazioni, restituisca l'approssimazione della soluzione del sistema $Ax = b$ attraverso il metodo di Gauss-Seidel ed il numero delle iterazioni effettuate dal metodo.

Suggerimento

L'intestazione della `function` ricalca quella di Jacobi

- function [x0,nit]=gseidel(A,b,x0,toll,nmax)

Fase di inizializzazione

- Si inizializzi `nit=0;`
- Si inizializzi `n` numero delle equazioni del sistema, pari alla lunghezza di `b`
- Si ponga `x=x0;`
- Si calcoli il residuo iniziale `r=b-A*x;`
- Si calcoli l'approssimazione iniziale dell'errore `err` pari a `norm(r)/norm(b)`

Metodo di Gauss-Seidel

Suggerimento

Ciclo principale: ripetere finchè `nit<nmax` e `err>toll`

+ Devo fare un ciclo su tutte le equazioni del sistema: per `i` che va da 1 fino a `n`
  - Pongo $s1 = 0$;
  - Pongo $s2 = 0$;
  - Devo calcolare la prima sommatoria sulle incognite del sistema già calcolate: per `j` che va da 1 fino a `i-1`
    - `s1=s1+A(i,j)*xn(j);`
  - fine del ciclo `for` su `j`
  - Devo calcolare la seconda sommatoria sulle incognite del sistema non ancora calcolate: per `j` che va da `i+1` fino a `i-1`
    - `s2=s2+A(i,j)*x(j);`
  - fine del ciclo `for` su `j`
  - Calcolo il nuovo valore della `i`-esima componente della soluzione con la formula `xn(i)=1/A(i,i)*(b(i)-s1-s2)`
+ fine del ciclo `for` su `i`

Metodo di Gauss-Seidel

  + Pongo `x=xn';`
  + Si incrementi di 1 il numero delle iterazioni `nit`
  + Si aggiorni con la stessa formula il valore del residuo `r`
  + Si aggiorni con la stessa formula il valore dell'errore `err`

- fine del ciclo `while`

Esercizio

Si applichi il metodo di Gauss-Seidel ai due sistemi dell'esercizio di Jacobi, con gli stessi dati. Cosa succede? Quali considerazioni si possono fare?