Appunti Calcolo Numerico

Luca Seggiani
2025

1 Lezione del 24-02-25

1.1 Introduzione al corso

Il corso di **calcolo numerico**, o come viene definito oggi *analisi numerica*, tratta lo studio degli algoritmi per problemi in campi continui (incognite in $\mathbb{R}$, siano queste numeri o funzioni, ecc...) o su grandi moli di dati.

Il programma del corso è così suddiviso:

1. Analisi dell'errore per funzioni scalari;
2. Richiami di algebra lineare (calcolo vettoriale e matriciale, ecc...);
3. Risoluzione di sistemi lineari, cioè forme $A\mathbf{x} = b$;
4. Interpolazione e approssimazione di funzioni nel senso dei minimi quadrati;
5. Metodi per l'integrazione, cioè per forme $\int_a^b f(x)\,dx$;
6. Equazioni non lineari, cioè ricerca dei punti $g(x) = 0$ per $g(x)$ non lineari;
7. Problemi agli autovalori, cioè date matrici $A \in \mathbb{C}^{m\times n}$, trovare $(\lambda, \mathbf{x})$ tali che $A\mathbf{x} = \lambda\mathbf{x}$.

1.1.1 Matematica del continuo

Abbiamo detto che i valori trattati sono continui, che intendiamo per appartenenti ad $\mathbb{R}$. Un problema apparente dei numeri reali è che richiedono teoricamente un numero infinito di cifre per la loro rappresentazione. Si rende quindi necessaria un'approssimazione in modo da ovviare ai problemi:

- Rappresentare oggetti matematici con un numero infinito di parametri;
- Risolvere problemi che non hanno formule chiuse per la soluzione, ma richiedono approcci iterativi (discesa a gradiente, ecc...) e quindi che richiedono un approssimazione data dall'impossibilità di effettuare infiniti passi.

1.1.2 Errori

Avremo quindi bisogno di valutare degli **errori**, che saranno gli *errori di approssimazione* come riportata sopra, uniti agli *errori nei dati* già presenti nella nostra mole di dati.

Una domanda che potremo porci è come questi errori influiscono sul risultato che ci interessa. Una prima distinzione può essere fra algoritmi **instabili** e **stabili**, cioè che *amplificano* l'errore o lo mantengono costante. Una seconda distinzione può essere sul **condizionamento** del problema, cioè la tendenza del problema a reagire in maniera drastica a piccole variazioni delle condizioni iniziali.

1.1.3 Efficienza

Un'altra considerazione importante è quella dell'**efficienza** degli algoritmi, cioè il tempo che questi richiedono per convergere ad una soluzione valida (non ottima, in quanto abbiamo visto dobbiamo troncare il numero di passi nel caso di approcci iterativi, che altrimenti potrebbe tendere ad infinito). Questo viene stimato attraverso il *costo computazionale*, tenendo conto di una certa *accuratezza* che vogliamo stabilire.

1.2 Rappresentazione dei reali

Introduciamo la classe dei **numeri reali in virgola mobile**, atta a rappresentare numeri $x \in \mathbb{R}$ usando parametri che stanno in $\mathbb{N}$, o al limite in $\mathbb{Z}$, in quanto abbiamo visto possiamo gestire numeri di questo tipo in modo esatto nei calcolatori.

Teorema 1.1: Rappresentazione dei reali in virgola mobile

Fissata una base $\beta > 1$, con $\beta \in \mathbb{N}$, si può sempre trovare un esponente $e \in \mathbb{Z}$ e una successione di cifre $\{\alpha_j\}_{j=1,2,...}$ tali per cui:

$$x = \text{sgn}(x) \cdot \beta^e \cdot \sum_{j=1}^{\infty} \alpha_j \beta^{-j}$$

Osserviamo che scelte tipiche per β sono 10 (decimale), 2 (binario) e 16 (esadecimale).

Un problema di questa rappresentazione è che non è propriamente **unica**: ad esempio, potremo scrivere un approssimazione di π come 3.14, o come $0.314 \cdot 10$, equivalentemente. Inoltre, possono esistere casi di numeri periodici, come $2.\overline{9}$, che sono effettivamente uguali ad altri (in questo caso 3).

Sfruttiamo allora il seguente teorema, dato senza dimostrazione:

Teorema 1.2: Teorema di rappresentazione

Dati $\beta \in \mathbb{N}, \beta > 1$ base e $x \in \mathbb{R}, x \neq 0$, allora esiste ed è unica la rappresentazione $x = \text{sgn}(x) \cdot \beta^e \cdot \sum_{j=1}^{\infty} \alpha_j \beta^{-j}$ (dal Teorema 1.1) tale che:

- $\alpha_1 \neq 0$;
- $!\exists k \in \mathbb{N} : \alpha_j = \beta - 1 \; \forall j > k$.

Introducendo queste due limitazioni possiamo quindi ovviare al problema dell'unicità.

Diamo quindi alcune definizioni, riguardo alla rappresentazione appena vista:

Definizione 1.1: Rappresentazione in virgola mobile normalizzata

L'unica rappresentazione che verifica il Teorema 1.2 si dice **rappresentazione in virgola mobile normalizzata**.

e riguardo alla successione $\{\alpha_j\}$:

Definizione 1.2: Mantissa

Prende il nome di **mantissa** la serie $\sum_{j=1}^{\infty} \alpha_j \beta^{-j}$.

Notiamo che vale:

$$\frac{1}{\beta} \leq \sum_{j=1}^{\infty} \alpha_j \beta^{-j} < 1$$

In quanto:

- Per il limite inferiore:

$$\sum_{j=1}^{\infty} \alpha_j \beta^{-j} \geq \beta^{-1} = \frac{1}{\beta}$$

- Per il limite superiore:

$$\sum_{j=1}^{\infty} \alpha_j \beta^{-j} < (\beta - 1) \sum_{j=1}^{\infty} \beta^{-j} = (\beta - 1) \left(\frac{1}{1 - \beta^{-1}} - 1 \right)$$

$$= (\beta - 1) \frac{\beta^{-1}}{1 - \beta^{-1}} = (\beta - 1) \frac{1}{\beta - 1} = 1$$

□

Vediamo che la mantissa non è effettivamente rappresentabile nella sua interezza in un calcolatore, in quanto richiederebbe un numero infinito di cifre. Si tronca quindi la mantissa, e si considera un range limitato per l'esponente. Si definisce quindi un sottinsieme:

Definizione 1.3: Numeri di macchina

Dati: $\beta \in \mathbb{N}$, $m \in \mathbb{N}$, $L, U \in \mathbb{Z}$ tali che $L \leq U$, si definisce $F(\beta, m, L, U)$ come:

$$F = \{x \in \mathbb{R} : x = \text{sgn}(x) \cdot \beta^e \cdot \sum_{j=1}^{m} \alpha_j \beta^{-j},\ L \leq e \leq U,\ \alpha_j = \{0, ..., \beta - 1\},\ \alpha_1 \neq 0\} \cup \{0\}$$

detto **numero di macchina**.

L'inclusione dello zero è necessario in quanto questo non è compreso nel primo insieme dato.

Notiamo che l'insieme dei numeri di macchina non è equispaziato (ci sono più numeri vicino allo zero). Presi intervalli $[\beta^{e-1}, \beta^e)$, questi risulterebbero equispaziati se venissero considerati su una scala logaritmica base β. All'interno di questi intervalli, invece, si hanno effettivamente numeri equispaziati, con periodo $\beta^e \cdot \beta^{-m} = \beta^{e-m}$.

1.2.1 Limiti inferiori e superiori

Potremmo chiederci quali sono i numeri **minimi** e **massimi** rappresentabili.

Osservando la definizione di F si ha che il numero più piccolo possibile è quello che si ha prendendo $\beta = L$, e $\alpha_j = 0$ per ogni $j > 1$, e $\alpha_1 = 1$, quindi:

$$\beta^L \cdot 1 \cdot \beta^{-1} = \beta^{L-1}$$

Il numero più grande si ha invece prendendo $\beta = U$ e $\alpha_j = \beta - 1$, e quindi:

$$\beta^U(\beta-1)\sum_{j=1}^{m}\beta^{-j} = \beta^U\left(\frac{1-\beta^{-m-1}}{1-\beta^{-1}} - 1\right)(\beta-1)$$

$$= \beta^U\left(\frac{1-\beta^{-m}}{\beta-1}\right)(\beta-1) = \beta^U(1-\beta^{-m})$$

Potremmo poi chiederci quanti numeri macchina esistono fissati β, m, L e U. Si hanno intanto due scelte di segno, $(U-L+1)$ scelte di esponenti e $(\beta^m - \beta^{m-1})$ scelte di mantisse (tutti i numeri rappresentabili su m cifre base β meno quelli con prima cifra nulla) più lo zero, da cui:

$$2 \cdot (U-L+1) \cdot (\beta^m - \beta^{m-1}) + 1$$

Le rappresentazioni più comuni dei numeri macchina sono definite dallo standard IEEE 754, che definisce:

Precisione	β	m	L	U	**Dimensione**
Singola	2	24	-126	127	4 byte (32 bit)
Doppia	2	53	-1022	1023	8 byte (64 bit)
Quadrupla	2	113	-16382	16383	16 byte (128 bit)

Casi particolari potrebbero richiedere precisioni più grandi o più lasche. Ultimamente, in particolare, si sono diffuse rappresentazioni a precisione più bassa (su 16 o addirittura 8 bit), in particolare nel campo delle reti neurali.

Il pacchetto MATLAB utilizza di default la precisione **doppia** come definita dallo standard IEEE 754.

1.3 Arrotondamento

Abbiamo visto come ci stiamo spostando dalla matematica esatta a una serie di approssimazioni. In generale, vorremo partire da un certo numero reale $x \in \mathbb{R}$, ma non appartenente all'insieme dei numeri macchina ($x \notin F(\beta, m, L, U)$). L'obiettivo è quello di riportare x ad una sua *approssimazione* appartenente ad $F(\beta, m, L, U)$.

Esistono 3 possibili situazioni:

- $|x| > \beta^U(1-\beta^{-m})$, cioè x maggiore del massimo rappresentabile (**overflow**);
- $|x| < \beta^{L-1}$, cioè x minore del minimo rappresentabile (**underflow**);
- $\beta^{L-1} \leq x \leq \beta^U(1-\beta^{-m})$. Se nei casi precedenti si poteva scegliere, rispettivamente, un M molto grande, o ∞, e 0, qui si può effettivamente procedere con l'arrotondamento.

Definizione 1.4: Arrotondamento

Un arrotondamento è una funzione $RD : \mathbb{R} \to F(\beta, m, L, U)$, con $RD(x)$ uno dei numeri di macchina in $F(\beta, m, L, U)$ "vicini" ad x.

Preso un certo reale x, avremo un numero di macchina a sinistra è uno a destra, cioè il primo più piccolo e il primo più grande. Possiamo allora definire i seguenti arrotondamenti:

- **Troncamento** (*round down*):
$$Tr(x) = \lfloor x \rfloor$$
- **Round-up:**
$$Ru(x) = \lceil x \rceil$$
- **Round-to-nearest:**
$$Rn(x) = \begin{cases} \lceil x \rceil, & \alpha_{m+1} \geq \frac{\beta}{2} \\ \lfloor x \rfloor, & \alpha_{m+1} < \frac{\beta}{2} \end{cases}$$
con α_{m+1} la prima cifra che viene scartata dall'arrotondamento.

Notiamo poi che preso il troncamento:

$$Tr(x) = \text{sgn}(x)\beta^e \sum_{j=1}^{m} \alpha_j \beta^{-j}$$

il round up corrispondente sarà:

$$Ru(x) = \text{sgn}(x)\beta^e \left(\sum_{j=1}^{m} \alpha_j \beta^{-j} + \beta^{-m} \right)$$

1.3.1 Errore di arrotondamento

Valgono le disuguaglianze:

- $|Tr(x) - x| \leq \beta^{e-m}$
- $|Rn(x) - x| \leq \frac{\beta^{e-m}}{2}$, che è il minimo errore possibile cioè:
$$|Rn(x) - x| = \min_{RD(x)}(RD(x) - x)$$
Per questo motivo da qui in poi assumeremo quindi di prendere sempre $Rn(x)$.

Definizione 1.5: Errori di arrotondamento

Definiamo:

- **Errore assoluto di arrotondamento:** $x - Rn(x) = \sigma_x$
- **Errore relativo di arrotondamento:** $\frac{x - Rn(x)}{x} = \epsilon_x$

La definizione di errore relativo è utile per avere un errore pressochè costante su tutta la retta dei reali. Si ha quindi che:

- Riguardo all'errore assoluto:
$$|\sigma_x| \leq \frac{\beta^{e-m}}{2}$$
- Riguardo all'errore relativo:
$$|\epsilon_x| \leq \frac{\beta^{e-m}}{2\beta^{e-1}} = \frac{1}{2}\beta^{1-m}$$
visto che $|x| > \beta^{e-1}$. Notiamo che questo errore non dipende dall'esponente e, che è quello che desideravamo.

Abbiamo quindi che l'insieme dei numeri in virgola mobile garantiscono un errore relativo limitato in modo uniforme, se non si incombe in overflow o underflow.

Definizione 1.6: Precisione

La quantità $U = \beta^{1-m}$ viene detta **precisione di macchina** di una certa rappresentazione a virgola mobile.

Ad esempio, nella precisioni doppia e singola, $U \approx 10^{-16}$ e $U \approx 10^{-8}$, che significa rispettivamente prime 15 o prime 7 cifre esatte.

1.3.2 Dettagli di implementazione

Prendiamo ad esempio la doppia precisione. Avremo:

- 1 bit di segno;
- 52 bit di mantissa (con 1 bit implicito impostato ad 1, per $\alpha_1 = 1$);
- 11 bit di esponente in rappresentazione con offset.

Riguardo agli altri formati si ha:

Precisione	Segno	Esponente	Mantissa	U
Singola	1	8	23	$\approx 10^{-8}$
Doppia	1	11	52	$\approx 10^{-16}$
Quadrupla	1	15	112	$\approx 10^{-34}$

1.4 Numeri sottonormalizzati

Vediamo una tecnica per la rappresentazione di numeri più piccoli di β^{L-1}, implementata nella maggior parte dei pacchetti software moderni (e nelle implementazioni degli standard a virgola mobile disponibili nei processori moderni). Si ha che se $x \in [0, \beta^{L-1}]$, allora si assume come prima cifra della mantissa 0, con esponente fisso ad L. Questo ci permette di rappresentare più numeri vicino allo zero.

Si avranno quindi numeri equispaziati fra 0 e β^{L-1} con distanza β^{L-m}. Inoltre, sui numeri sottonormalizzati si avrà un errore relativo non limitato da $\frac{1}{2}\beta^{1-m}$, ma che invece aumenta avvicinandosi a 0.

I numeri sottonormalizzati sono indicati da un **valore speciale dell'esponente**, cosa che accade anche per:

- I valori $+\infty$ e $-\infty$;
- I valori NaN (Not a Number) con relativi codici di errore in mantissa.

La presenza di questi valori rende necessaria l'approssimazione delle precisioni per i numeri in virgola mobile.

2 Lezione del 28-02-25

2.1 Operazioni sui numeri macchina

Abbiamo introdotto l'insieme dei numeri macchina. Vediamo adesso come eseguire **operazioni** fra elementi di questi insiemi.

Notiamo che, di base, dati $x, y \in F(\beta, m, L, U)$, non necessariamente $x \circ y \in F(\beta, m, L, U)$ per le comuni operazioni aritmetiche $+, -, \times, \div$.

Quello che faremo è quindi approssimare tali operazioni, cioè dire:

- $x \oplus y = Rn(x + y)$
- $x \ominus y = Rn(x - y)$
- $x \otimes y = Rn(x \times y)$
- $x \oslash y = Rn(x \div y)$

Effetto di questa approsimazione è negare proprietà note dei reali, come ad esempio l'associativa:

$$(x \oplus y) \oplus z \neq x \oplus (y \oplus z)$$

$$(x \oplus y) \otimes z \neq x \oplus (y \otimes z)$$

Cioè, la valutazione di una formula con ordini diversi ma equivalenti in aritmetica esatta può portare a risultati differenti nell'insieme dei numeri di mmacchina.

2.1.1 Errore nel calcolo di funzione

Sia $f : \mathbb{R}^m \to \mathbb{R}$, e si voglia calcolare $f(P_0)$, con $P_0 = \left(x_1^{(0)}, x_2^{(0)}, ...x_m^{(0)}\right) \in \mathbb{R}^m$. Le operazioni aritmetiche $+, -, \times, \div$ possono essere viste come funzioni di questo tipo. Ci interroghiamo quindi sulla fonte dell'errore nella loro valutazione:

1. Nel caso contenga funzioni irrazionali o trascendenti, f verrà approssimata con una funzione $\overline{f}$ che coinvolge solo operazioni aritmetiche di base $+, -, \times, \div$;
2. $\overline{f}$ viene tradotta in un *algoritmo* $\overline{f}_a$, ovvero in una formula che coinvolge $\oplus, \ominus, \otimes, \oslash \leftarrow +, -, \times, \div$. La formula ottenuta finora viene detta **algoritmo**;
3. Potrebbe essere che $P_0 \neq F(\beta, m, L, U)$, e quindi viene approssimato a $P_1 = Rn(P_0)$.

Quindi, vogliamo $f(P_0)$ ma possiamo solo approssimarla come $\overline{f}_a(P_1)$.

Ad esempio, poniamo di voler calcolare e^π. I passaggi nell'ordine appena visto saranno:

1. Approssimamo l'esponenziale al secondo grado dello svillupppo di Taylor:

$$e^x \approx 1 + x + \frac{x^2}{2} = \overline{f}(x)$$

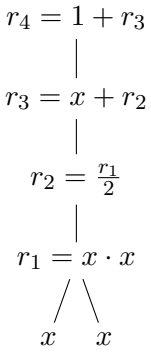
2. Si riporta la $\overline{f}(x)$ a $\overline{f}_a(x)$:

$$1 \oplus (x \oplus ((x \otimes x) \oslash 2))$$

Indicheremo algoritmi di questo tipo anche attraverso **risultati intermedi**:

- $r_1 = x \cdot x$
- $r_2 = \frac{r_1}{2}$
- $r_3 = x + r_2$
- $r_4 = 1 + r_3$

con il risultato finale inteso come l'ultimo risultato intermedio. Un modo di visualizzare i risultati intermedi di un algoritmo può essere un albero:



dove la *radice* rappresenta il **risultato** e le *foglie* rappresentano le variabili della funzione.

3. Si approssima π al numero macchina più vicino:

$$Rn(\pi) = 3.1415 = P_1$$

Avremo quindi la formula finale:

$$1 \oplus (P_1 \oplus ((P_1 \otimes P_1) \oslash 2))$$

Diamo quindi la definizione di **errore assoluto**:

Definizione 2.1: Errore assoluto

Data $f : \mathbb{R}^m \to \mathbb{R}$, un punto $P_0 \in \mathbb{R}^m$ ed un algoritmo $\overline{f}_a$, l'errore assoluto è dato da:

$$\sigma_f = \overline{f}_a(P_1) - f(P_0), \quad P_1 = Rn(P_0)$$

e di errore relativo:

Definizione 2.2: Errore relativo

Date le ipotesi della definizione 2.1, l'errore relativo è dato da:

$$\epsilon_f = \frac{\overline{f}_a(P_1) - f(P_0)}{f(P_0)} = \frac{\sigma_f}{f(P_0)}$$

2.1.2 Errore di funzioni razionali

Assumiamo per adesso f **funzione razionale**, ovvero f si può definire con un numero di operazioni in $+, -, \times, \div$. Assumere funzioni razionali ci permette di prendere $f = \overline{f}$ e $f_a = \overline{f}_a$ (non ci sono irrazionali da riportare ai razionali). Potremo allora dire:

$$\sigma_f = f_a(P_1) - f(P_0) = f_a(P_1) - f(P_1) + f(P1) - f(P_0)$$

che possiamo dividere in:

$$\sigma_f = \sigma_a + \sigma_d, \quad \sigma_a = f_a(P_1) - f(P_1), \quad \sigma_d = f(P_1) - f(P_0)$$

che chiamiamo rispettivamente **errore assoluto algoritmico** e **errore assoluto inerente**.

Allo stesso modo possiamo definire l'**errore relativo**:

$$\epsilon_f = \frac{\sigma_f}{f(P_0)} = \frac{f_a(P_1) - f(P_0)}{f(P_0)} = \frac{f_a(P_1) - f(P_1)}{f(P_0)} + \frac{f(P_1) - f(P_0)}{f(P_0)}$$

$$= \frac{f_a(P_1) - f(P_1)}{f(P_1)} \cdot \frac{f(P_1)}{f(P_0)} + \frac{f(P_1) - f(P_0)}{f(P_0)}$$

che si divide nuovamente in :

$$\epsilon_f = \epsilon_a + \epsilon_d, \quad \epsilon_a = \frac{f_a(P_1) - f(P_1)}{f(P_1)}, \quad \epsilon_d = \frac{f(P_1) - f(P_0)}{f(P_0)}$$

che chiamiamo rispetivamente **errore relativo algoritmico** e **errore relativo inerente**. Questo viene da:

$$\epsilon_f = [...] = \frac{f_a(P_1) - f(P_1)}{f(P_1)} \cdot \frac{f(P_1)}{f(P_0)} + \frac{f(P_1) - f(P_0)}{f(P_0)}$$

si nota che $\frac{f(P_1)}{f(P_0)} = 1 + \frac{f(P_1) - f(P_0)}{f(P_0)}$, e quindi:

$$= \epsilon_a \cdot \left(1 + \frac{f(P_1) - f(P_0)}{f(P_0)}\right) + \epsilon_d = \epsilon_a(1 + \epsilon_d) + \epsilon_d = \epsilon_a + \epsilon_d + \epsilon_a\epsilon_d \approx \epsilon_a + \epsilon_d$$

assumendo $\epsilon_a\epsilon_d \approx 0$.

Ci interessa dare stime superiori per i valori assoluti di errori assoluti e relativi, come avevamo fatto per gli errori delle funzioni di arrotondamento da reali a numeri macchina. In generale, quindi, per limitare $|\sigma_f|$ cercheremo disuguaglianze $|\sigma_a| < \tau_1$, $|\sigma_d| < \tau_2$, da cui:

$$|\sigma_f| < \tau_1 + \tau_2$$

2.1.3 Stima dell'errore inerente

Avevamo quindi definito l'**errore assoluto inerente**:

$$\sigma_d = f(P_1) - f(P_0)$$

Sotto l'ipotesi $f \in C^1(D)$ per $D \subset \mathbb{R}^m$ che contiene P_0, si può usare lo sviluppo di Taylor di f in P_0, troncato al primo ordine:

$$f(P_1) - f(P_0) = f(P_0) + \nabla f(\overline{P})^T(P_1 - P_0) - f(P_0) = \nabla f(\overline{P})^T(P_1 - P_0)$$

$$= \sum_{j=1}^{m} \frac{\partial f}{\partial x_j}(\overline{P}) \cdot \left(x_j^{(1)} - x_j^{(0)}\right) \approx \sum_{j=1}^{m} \frac{\partial f}{\partial x_j} P_0 \cdot \left(x_j^{(1)} - x_j^{(0)}\right)$$

dove $\overline{P}$ è un punto che sta sul segmento $\overline{P_1 P_0}$. Da questo potremo dire:

$$\sigma_d = \sum_{j=1}^{m} \frac{\partial f}{\partial x_j} P_0 \cdot \sigma_j$$

dove $\sigma_j = \left(x_j^{(1)} - x_j^{(0)}\right)$ è l'**errore di arrotondemento** nella componente j di P_0, e $\frac{\partial f}{\partial x_j} P_0$ viene detto **coefficiente di amplificazione**.

Per l'**errore relativo inerente**, che avevamo definito come:

$$\epsilon_d = \frac{f(P_1) - f(P_0)}{f(P_0)}$$

potremo fare considerazioni simili:

$$\epsilon_d = \frac{\sum_{j=1}^{m} \frac{\partial f}{\partial x_j}(P_0) \cdot \sigma_j}{f(P_0)} = \sum_{j=1}^{m} \frac{x_j^{(1)} - x_j^{(0)}}{x_j^{(0)}} \cdot \frac{\partial f}{\partial x_j}(P_0) \cdot \frac{x_j^{(0)}}{f(P_0)}$$

dove $\epsilon_j = \frac{x_j^{(1)} - x_j^{(0)}}{x_j^{(0)}}$ sarà l'**errore di arrotondamento relativo** nella componente j di P_0 e $P_j = \frac{\partial f}{\partial x_j}(P_0) \cdot \frac{x_j^{(0)}}{f(P_0)}$ viene detto **coefficiente di amplificazione dell'errore relativo**.

La formula finale sarà quindi:

$$\epsilon_d = \sum_{j=1}^{m} \epsilon_j P_j$$

I problemi in cui si devono calcolare f i cui coefficienti di amplificazione degli errori relativi sono grandi in modulo (o ce n'è almeno uno sufficientemente grande) si dicono **malcondizionati**. Viceversa, se $|P_j|$ è vicino a 1 per ogni j il problema si dice **ben condizionato**, cioè che ϵ_d è di un ordine di grandezza comparabile a $\max(\epsilon_i)$

Notiamo inoltre che il condizionamento di un problema dipende solamente dalla sua struttura matematica.

2.1.4 Errori inerenti delle operazioni aritmetiche

Vediamo gli errori inerenti associati alle 4 operazioni aritmetiche $+, -, \times, \div$:

Operazione	σ_d	ϵ_d
$x \oplus y$	$\sigma_x + \sigma_y$	$\frac{x}{x+y}\epsilon_x + \frac{y}{x+y}\epsilon_y$
$x \ominus y$	$\sigma_x - \sigma_y$	$\frac{x}{x-y}\epsilon_x - \frac{y}{x-y}\epsilon_y$
$x \otimes y$	$y\sigma_x + x\sigma_y$	$\epsilon_x + \epsilon_y$
$x \oslash y$	$\frac{1}{y}\sigma_x - \frac{x}{y^2}\sigma_y$	$\epsilon_x - \epsilon_y$

Notiamo come somme e sottrazioni non amplificano gli errori totali, mentre prodotti e divisioni non amplificano gli errori relativi (riguardo agli errori inerenti). Questo significa che somme e sottrazioni possono avere errori relativi grandi quando $|x + y| << \min\{|x|, |y|\}$. Questo effetto viene detto **cancellazione numerica**.

2.1.5 Stima dell'errore algoritmico

Avevamo definito un algoritmo $f_a(x)$ di cui vogliamo stimare l'errore algoritmico assoluto $\sigma_a = f_a(P_1) - f(P_1)$. Assumiamo $P_1 = Rn(P_0) \in F(\beta, m, L, U)$, cioè gli operandi come privi di errori iniziali di arrotondamento.

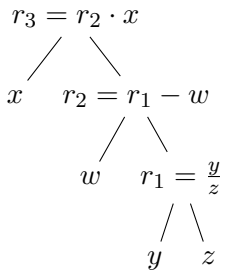
L'idea è di seguire l'errore generato dall'algoritmo sul grafo (o albero) che lo rappresenta sfruttando le relazioni per l'errore inerente nelle 4 operazioni aritmetiche. Prendiamo ad esempio la funzione:

$$f(x, y, z, w) = x \cdot \left(\frac{y}{z} - w \right)$$

Avremo i risultati intermedi:

- $r_1 = \frac{y}{z}$
- $r_2 = r_1 - w$
- $r_3 = r_2 \cdot x$

Di cui riportiamo il grafo:



dove ϵ_3, ϵ_2 e ϵ_1 saranno termini associati ad ogni risultato intermedio che rappresenteranno gli errori di troncamento dei risultati intermedi stessi, e ϵ_{r3}, ϵ_{r2} e ϵ_{r1} rappresenteranno gli errori inerenti delle singole operazioni per il calcolo dei risultati intermedi.

Partiamo dalla radice per valutare gli errori:

$$\epsilon_d = \epsilon_{r_3} = \epsilon_3 + \epsilon_x + \epsilon_{r_2} = \epsilon_3 + \epsilon_{r_2}$$

$$= \epsilon_3 + \epsilon_2 + \frac{-zw}{y - zw} \cdot \epsilon_w + \frac{y}{y - zw} \cdot \epsilon_{r_1} = \epsilon_3 + \epsilon_2 + \frac{y}{y - zw} \cdot \epsilon_{r_1}$$

$$= \epsilon_3 + \epsilon_2 + \frac{y}{y - zw} (\epsilon_1 + \epsilon_y - \epsilon_z) = \epsilon_3 + \epsilon_2 + \frac{y}{y - zw} \cdot \epsilon_1 = \epsilon_a$$

Dove abbiamo ignorato i termini di errore relativo ϵ_i legati ad ogni variabile (come avevamo detto sopra, gli operandi sono considerati come privi di errore di arrotondamento). Per la stima di ϵ_3, ϵ_2 e ϵ_1, vale $\epsilon_i \leq U$ precisione macchina. Nel caso di errori assoluti vale $|\sigma_i| \leq U \cdot \max(x_i)$ considerata ogni variabile x_i.

Chiaramente, diversi algoritmi equivalenti in aritmetica esatta potranno avere errori algoritmici diversi fatte tutte le approssimazioni.

Prendiamo ad esempio la funzione $f(x, y) = x^2 - y^2$. Potremmo adottare due algoritmi:

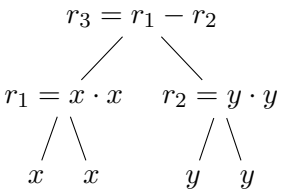
1. Si prendono i risultati intermedi:

$$r_1 = x \cdot x$$

$$r_2 = y \cdot y$$

$$r_3 = x - y$$

Da cui il grafo:



Questo approccio potrebbe risultare il più intuitivo: dalla stima dell'errore si ha:

$$\epsilon_{r_3} = \epsilon_1 + \frac{x^2}{x^2 - y^2}\epsilon_{r_1} - \frac{y^2}{x^2 - y^2}\epsilon_{r_2} = \epsilon_1 + \frac{x^2}{x^2 - y^2}(\epsilon_1 + \epsilon_x + \epsilon_x) - \frac{y^2}{x^2 - y^2}(\epsilon_2 + \epsilon_y + \epsilon_y)$$

$$= \epsilon_1 + \frac{x^2}{x^2 - y^2}\epsilon_1 - \frac{y^2}{x^2 - y^2}\epsilon_2$$

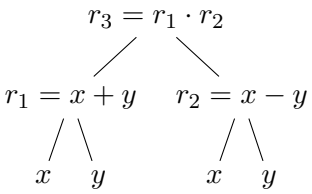
2. Un altro approccio è quello di prendere i risultati intermedi:

$$r_1 = x + y$$

$$r_2 = x - y$$

$$r_3 = r_1 \cdot r_2$$

Da cui il grafo:



Vediamo la stima dell'errore:

$$\epsilon_{r_3} = \epsilon_3 + \epsilon_{r_1} + \epsilon_{r_2} = \epsilon_3 + \epsilon_1 + \frac{x}{x + y}\epsilon_x + \frac{y}{x + y}\epsilon_y + \epsilon_2 + \frac{x}{x - y}\epsilon_x - \frac{y}{x - y}\epsilon_y$$

$$= \epsilon_3 + \epsilon_1 + \epsilon_2$$

Notiamo che la stima dell'errore del secondo approccio è più conveniente, in quanto più strettamente limitata al di sotto di un valore fisso: $\epsilon_1 + \epsilon_2 + \epsilon_3 = 3U$, contro il $\left(1 + \frac{|x^2+y^2|}{|x^2-y^2|}\right) U$ del primo approccio.

Abbiamo visto quindi tenciche per la stima di ϵ_a e ϵ_d (σ_a e σ_d), che ci permettono di calcolare $|\epsilon_f| \leq |\epsilon_a| + |\epsilon_d|$ ($|\sigma_f| \leq |\sigma_a| + |\sigma_d|$).

2.1.6 Problema diretto

Un problema classico sarà quello di, data f, un algoritmo risolutivo f_a e una stima degli errori d_{x_i}, di stimare σ_f per $P_0 \in D \subset \mathbb{R}^m$.

Ad esempio, prendiamo la funzione:

$$f : D \to \mathbb{R}, \quad f(x_1, x_2) = \frac{x_1}{x_2}, \quad D = [1, 3] \times [4, 5]$$

Diamo una stima dell'errore di arrotondamento delle variabili:

$$|\sigma_{x_i}| \leq \frac{1}{2} \cdot 10^{-2}$$

- Iniziamo con la stima dell'errore inerente. Si ha, dall'errore assoluto inerente della divisione:

  $$\sigma_d = \frac{1}{x_2}\sigma_{x_1} - \frac{x_1}{x_2^2}\sigma_{x_2}$$

  Quello che ci interessa è dare stime superiori, quindi prendiamo i due valori:

  $$\left|\frac{1}{x_2}\right| \approx \frac{1}{4}, \quad \left|\frac{x_1}{x_2^2}\right| \approx \frac{7}{16}$$

  Dai valori massimi che si possono ottenere sul dominio D, da cui:

  $$|\sigma_d| \leq \frac{1}{4} \cdot \frac{1}{2} \cdot 10^{-2} + \frac{7}{16} \cdot \frac{1}{2} \cdot 10^{-2} = \left(\frac{1}{8} + \frac{7}{32}\right) \cdot 10^{-2} = \frac{7}{32} \cdot 10^{-2}$$

- Vediamo quindi la stima dell'errore algoritmico. L'albero dell'algoritmo sarà banalmente:

  $$\begin{array}{c} r_1 = \frac{x_1}{x_2} \\ \diagup \quad \diagdown \\ x_1 \quad x_2 \end{array}$$

  da cui posto $\sigma_{x_i} = 0$ resterà solo l'errore di arrotondamento assoluto σ_1:

  $$|\sigma_a| = \sigma_1 = U \cdot \max\left(\frac{x_1}{x_2}\right) = \frac{3}{4} \cdot \frac{1}{2} \cdot 10^{-2} = \frac{3}{8} \cdot 10^{-2}$$

Otteniamo quindi l'errore totale come la somma dell'errore algoritmico e dell'errore inerente, cioè:

$$|\sigma_f| = |\sigma_a| + |\sigma_f| = \left(\frac{7}{32} + \frac{3}{8}\right) \cdot 10^{-2} = \frac{19}{32} \cdot 10^{-2}$$

2.1.7 Problema inverso

Il problema inverso potrebbe essere quello di, dato $\tau > 0$, f e un punto $P_0 \in \mathbb{R}^n$, determinare un algoritmo ed un valore di precisione macchina U tali per cui $|s_f| < \tau$.

Prendiamo ad esempio il problema considerato prima, ma sul dominio:

$$D = [1, 2] \times [-2, -1]$$

e cerchiamo una precisione macchina U tale per cui l'errore assoluto σ_f è limitato a $\tau = 10^{-2}$, cioè:

$$|\sigma_f| \leq 10^{-2}$$

- Iniziamo di nuovo con la stima dell'errore inerente. Dagli stessi errori inerenti considerati prima si hanno le stime:

$$\left|\frac{1}{x_2}\right| \approx 1, \quad \left|\frac{x_1}{x_2^2}\right| \approx 2$$

Dai valori massimi che si possono ottenere sul dominio D, da cui:

$$|\sigma_d| \leq U + 2U = 3U$$

- Vediamo quindi la stima dell'errore algoritmico, che prendendo il valore massimo della funzione in maniera analoga a prima sarà:

$$|\sigma_a| = \sigma_1 = U \cdot \max\left(\frac{x_1}{x_2}\right) = 2U$$

L'errore totale sarà quindi: $|\sigma_f| = |\sigma_a| + |\sigma_f| = 3U + 2U = 5U$

Imponiamo quindi la diseguaglianza iniziale, cioè:

$$|\sigma_f| = 5U \leq 10^{-2} \implies U \leq \frac{1}{5} \cdot 10^{-2}$$

3 Lezione del 03-03-25

3.1 Riassunto sulla stima dell'errore

Riassumiamo quindi le regole viste per la stima dell'errore su funzioni razionali. Avevamo dato la definizione di errore **assoluto** σ_f e errore **relativo** ϵ_f, entrambi composti da due fattori denominati errore **algoritmico** e errore **inerente**, con pedici rispettivamente a e d.

- Riguardo all'errore **inerente assoluto** avevamo preso su un dominio D la stime:

$$|\sigma_d| \leq \sum_{j=1}^{m} A_j \cdot |\sigma_j|$$

con $|\sigma^j|$ **errore di arrotondamento** e A_j **coefficiente di amplificazione**:

$$A_j = \max_{P \in D}\left(\frac{\partial f}{\partial x_j}(P)\right)$$

Per l'errore di arrotondamento avevamo visto potevamo prendere:

$$|\sigma_j| \leq U \cdot |x_j|$$

con U precisione macchina.

- Riguardo all'**errore inerente relativo** avevamo invece preso:

$$|\epsilon_d| \leq \sum_{j=1}^{m} \overline{A}_j \cdot |\epsilon_j|$$

con $|\epsilon_j|$ **errore di arrotondamento relativo** e $\overline{\sigma_j}$ **coefficiente di amplificazione relativo**:

$$\overline{A_j} = \max_{P \in D} \left(\frac{x_j \cdot \frac{\partial f}{\partial x_j}(P)}{f(P)} \right)$$

Per l'errore di arrotondamento relativo potevamo quindi prendere:

$$|\epsilon_j| \leq U$$

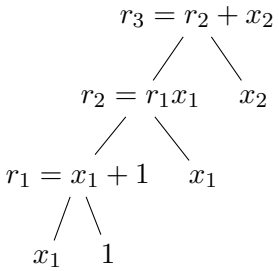
Vediamo come ultimo esempio il calcolo dell'errore relativo $|\epsilon_f|$ per la funzione:

$$f(x_1, x_2) = (x_1 + 1)x_1 + x_2$$

- Iniziamo con il calcolo dell'errore inerente, senza fare assunzioni su ϵ_{x_i}:

$$|\epsilon_d| \leq \frac{x_1 \cdot \frac{\partial f}{\partial x_1}(x_1, x_2)}{f(x_1, x_2)} \epsilon_{x_1} + \frac{x_2 \cdot \frac{\partial f}{\partial x_2}(x_1, x_2)}{f(x_1, x_2)} \epsilon_{x_2} = \frac{x_1(2x_1 + 1)}{(x_1 + 1)x_1 + x_2} \epsilon_{x_1} + \frac{x_2}{(x_1 + 1)x_1 + x_2} \epsilon_{x_1}$$

- Calcoliamo poi l'errore algoritmico dell'algoritmo:



da cui:

$$|\epsilon_a| \leq \epsilon_3 + \frac{(x_1 + 1)x_1}{(x_1 + 1)x_1 + x_2} \epsilon_{r_2} + \frac{x_2}{(x_1 + 1)x_1 + x_2} \epsilon_{x_2} = \epsilon_3 + \frac{(x_1 + 1)x_1}{(x_1 + 1)x_1 + x_2} (\epsilon_2 + \epsilon_{r_1} + \epsilon_{x_1})$$

$$= \epsilon_3 + \frac{(x_1 + 1)x_1}{(x_1 + 1)x_1 + x_2} (\epsilon_2 + \epsilon_1 + \frac{x_1}{x_1 + 1} \epsilon_{x_1} + \frac{1}{x_1 + 1} \cdot 0) = \epsilon_3 + \frac{(x_1 + 1)x_1}{(x_1 + 1)x_1 + x_2} (\epsilon_2 + \epsilon_1)$$

Abbiamo quindi l'errore complessivo:

$$|\epsilon_f| \leq |\epsilon_a| + |\epsilon_d| = \epsilon_3 + \frac{(x_1 + 1)x_1}{(x_1 + 1)x_1 + x_2} (\epsilon_2 + \epsilon_1) + \frac{x_1(2x_1 + 1)}{(x_1 + 1)x_1 + x_2} \epsilon_{x_1} + \frac{x_2}{(x_1 + 1)x_1 + x_2} \epsilon_{x_1}$$

3.2 Errori di funzioni non razionali

Abbiamo finora trascurato il caso di funzioni non razionali. Prendiamo ad esempio di voler calcolare l'errore su funzioni come $e^{\cos(x+y)}$. In questo caso sarà necessario usare un approssimazione razionale di f che chiamiamo $\overline{f}$, che poi porteremo a $\overline{f}_a$ che usa operazioni macchina detta **algoritmo**. In questo caso l'errore sarà dato dall'*errore inerente*, dall'*errore algortimico* e dall'**errore analitico** σ_{an} della funzione, cioè potremo dire:

$$\overline{f}_a(P_1) - f(P_0) = \overline{f}_a(P_1) - \overline{f}(P_1) + \overline{f}(P_1) - f(P_1) + f(P_1) - f(P_0) = \sigma_a + \sigma_{an} + \sigma_d$$

L'errore inerente sarà calcolato sulla f originale, mentre l'errore analitico sarà calcolato con la nuova $\overline{f}$, e in particolare dipenderà dall'approssimazione razionale che usiamo.

Vediamo per adesso approssimazioni polinomiali attraverso la **formula di Taylor**. Nel caso scalare si ha:

Teorema 3.1: Formula di Taylor

Data $f : \mathbb{R} \to \mathbb{R}$, $f \in C^1$, allora dato $x_0 \in \mathbb{R}$ si ha:

$$f(x) = \sum_{n=0}^{k} \frac{f^{(n)}(x_0)}{n!} \cdot (x - x_0)^n + \frac{f^{(k+1)}(\eta)}{(k+1)!}(x - x_0)^{k+1}$$

Dove:

$$\epsilon_l = \frac{f^{(k+1)}(\eta)}{(k+1)!}(x - x_0)^{k+1}$$

rappresenta l'**errore di Lagrange** al k-esimo grado, con $\eta \in [x_0, x]$ il punto di massimo della $k+1$-esima derivata di f.

In questo caso:

$$\overline{f}(x) = \sum_{n=0}^{k} \frac{f^{(n)}(x_0)}{n!} \cdot (x - x_0)^n$$

cioè la serie di Taylor troncata al k-esimo grado sarà una buona approssimazione per f, e l'errore analitico sarà dato da:

$$\sigma_{an} = R(x) = f(x) - T(x, k, x_0)$$

con $R(x)$ il resto fra lo sviluppo di Taylor troncato $T(x, k)$ e la funzione stessa $f(x)$. In questo caso fissato k si potrà dare una stima di errore direttamente dall'errore di Lagrange, cioè:

$$R(x) = f(x) - T(x, k, x_0) = \frac{f^{(k+1)}(\eta)}{(k+1)!}(x - x_0)^{k+1}$$

e più in particolare, sfruttando il limite superiore dato da:

$$R(x) \leq \epsilon_l = \frac{\max_{[x_0,x]}\left(f^{(k+1)}\right)}{(k+1)!}(x - x_0)^{k+1} = \frac{M}{(k+1)!}(x - x_0)^{k+1}$$

3.2.1 Approssimazione dell'esponenziale

Prendiamo di volere calcolare l'errore dato dall'approssimazione dell'esponenziale al k-esimo grado su un intervallo comprensivo di $x_0 = 0$, che chiamiamo $D = [-l, u]$. Avremo quindi l'approssimazione:

$$e^x = T(x, k, 0) = \sum_{n=0}^{k} \frac{(x - x_0)^n}{n!}$$

e la funzione resto:

$$R(x) = e^x - T(x, k, 0) = \frac{f^{(k+1)}(\eta)}{(k+1)!}x^{k+1}$$

prendiamo quindi la stima superiore di $f^{(k+1)}(\eta)$ su $\eta \in D$ (il caso peggiore sarà quello in cui valutiamo x sull'estremo superiore u):

$$f^{(k+1)}(\eta) \leq \max_{\eta \in [0,u]} f^{(k+1)}(\eta) = e^u$$

da cui:

$$R(x) \leq \frac{e^u x^{k+1}}{(k+1)!}$$

Stimiamo allora l'errore analitico relativo come:

$$|\epsilon_{an}| = \frac{R(x)}{f(x)} \leq \frac{\max_{[-l,u]} \left| \frac{e^u x^{k+1}}{(k+1)!} \right|}{\min_{[-l,u]} e^x} = \frac{e^{u-l} u^{k+1}}{(k+1)!}$$

3.2.2 Note sull'implementazione dell'esponenziale

Vediamo alcuni dettagli sull'implementazione pratica di un'approssimazione dell'esponenziale nel linguaggio MATLAB/Octave. Il primo modo che potrebbe venire in mente prevede di mantenere un numeratore e un denominatore:

```
function val = myexp1(x, n)
    num = 1;
    den = 1;

    val = 1;

    for i = 1:n
      num = num * x;
      den = den * i;

      val = val + num / den;
    end
end
```

Notiamo però che questo approccio presenta notevole instabilità numerica con grandi valori di k, ad esempio si veda:

```
>> myexp(20, 500)
ans =
   NaN
```

Questo deriva dal fatto che per grandi valori di k si ottiene eventualmente `num = Inf` e `den = Inf`, da cui il `NaN`. Un'approccio migliore si ha quindi mantenendo direttamente il termine e accumulandolo:

```
function acc = myexp2(x, n)
    term = 1;
    acc = 1;

    for i = 1:n
        term = term * x / i;
        acc = acc + term;
    end
end
```

Notiamo però che in questo caso si hanno problemi di cancellazione per argomenti negativi, ad esempio si veda:

```
>> myexp(-30, 500)
ans =
  -3.0668e-05
```

In questo caso si ha più accuratezza imponendo argomenti positivi, ad esempio sfruttando:

$$e^{-x} = \frac{1}{e^x}$$

Si veda ad esempio:

```
>> 1/myexp(30, 500)
ans =
   9.3576e-14
```

Modifichiamo quindi la funzione per rilevare automaticamente esponenti negativi e adottare la forma corretta:

```
function acc = myexp3(x, n)
    neg = false;
    if x < 0
        neg = true;
        x = -x;
    end

    term = 1;
    acc = 1;

    for i = 1:n
        term = term * x / i;
        acc = acc + term;
    end

    if neg
        acc = 1 / acc;
    end
end
```

Provando questa funzione su un range di interi da -30 a 30, con la serie di Taylor troncata a 500, dà un errore massimo rispetto al valore reale di circa $10^{-16} \sim 10^{-17}$.

Veniamo quindi a fare alcuni richiami di algebra lineare. Nella maggior parte dei casi che ci interessano vorremo trattare non di scalari, ma di quantità vettoriali, ad esempio $x \in \mathbb{C}^n$, con $n > 1$.

3.3 Matrici complesse

Ci saranno utili le matrici perchè rappresentano direttamente tutte le **funzioni lineari**. Ad esempio, posta $f : \mathbb{C}^n \to \mathbb{C}^m$ tale che:

- $f(x + y) = f(x) + f(y)$ (addittività);
- $f(\lambda x) = \lambda f(x)$ (omogeneità)

detta *funzione lineare* allora $\exists! A \in \mathbb{C}^{m \times n}$ tale che $f(x) = Ax$, $\forall x \in \mathbb{C}^n$.

Nel corso useremo sia matrici in $\mathbb{R}$ che matrici in $\mathbb{C}$, dove l'appartenenza a ciascuno di questi campi dipende dalla appartenenza di essi delle **entrate** A_{ij} della matrice. In ogni caso, una matrice reale non sarà che un caso particolare delle matrici complesse.

Si danno poi per scontate le definizioni di matrici:

- *Quadrate* ($n = m$):
- *Rettangolari* ($n \neq m$);
- *Diagonali* (elementi nulli fuori dalla diagonale);
- *Triangolari superiori/inferiori* (elementi nulli sotto/sopra la diagonale)

3.3.1 Trasposta coniugata

Definiamo infine la **trasposta coniugata** di una certa matrice, generalizzata al campo complesso dalla **matrice hermitiana**:

Definizione 3.1: Trasposta coniugata

Data una matrice $A \in \mathbb{C}^{n \times m}$, la trasposta coniugata A^T sarà:

$$(A^T)_{ij} = A_{ji}$$

e la matrice hermitiana A^H sarà:

$$(A^H)_{ij} = \overline{A_{ji}}$$

3.4 Algebra vettoriale

Diamo alcune nozioni riguardo alle operazioni fra vettori.

3.4.1 Prodotto scalare

Diamo la definizione di prodotto scalare, generalizzato al campo complesso dal **prodotto hermitiano** (entrambi *prodotti interni*):

Definizione 3.2: Prodotto interno

Definiamo il prodotto interno fra due vettori $x, y \in \mathbb{C}^n$ come:

$$< x, y >= \sum_{j=1}^{n} x_j \overline{y_j}$$

dove $\overline{y_j}$ rappresenta il **coniugato** di y_j, che chiaramente in $\mathbb{R}$ si riduce a y_j stesso e quindi:

$$< x, y >= \sum_{j=1}^{n} x_j y_j, \quad x, y \in \mathbb{R}^n$$

3.4.2 Indipendenza lineare

Diamo la definizione di indipendenza linare:

Definizione 3.3:

Un insieme di vettori $\{x_1, ..., x_s\}$ si dice linearemente indipendente se:

$$x_1 + ... + x_s = 0 \Leftrightarrow x_1, ..., x_s = 0$$

Possiamo sfruttare l'indipendenza lineare per definire **basi**:

Definizione 3.4: Base

Se $s = n$ l'insieme $\{x_1, ..., x_s\}$ si dice **base** di $\mathbb{C}^n$.

Questo significa che $\forall y \in \mathbb{C}^m$, $\exists!\{c1, ..., c_s\}$ tali che $y = \sum_{j=1}^{n} c_j x_j$, cioè combinazioni lineari dei vettori di base individuano qualsiasi vettore nello spazio $\mathbb{C}^n$.

3.5 Operazioni fra matrici

Date due matrici A e B con lo stesso numero di righe e colonne si possono definire le operazioni:

- **Somma:** $A, B, C \in \mathbb{C}^{m \times n}$, $A + B = C$, $c_{ij} = a_{ij} + b_{ij}$;
- **Prodotto:** $A \in \mathbb{C}^{m \times n}, B \in \mathbb{C}^{n \times p}, C \in \mathbb{C}^{m \times p}$, $A \cdot B = C$, $c_{ij} = \sum_{h=1}^{n} a_{ih} b_{hj}$ sia in reali che in complessi.

3.5.1 Considerazioni computazionali sulle operazioni matriciali

Dal punto di vista computazionale, è immediato che il prodotto scalare ha complessità $O(n)$, la somma matriciale ha complessità $O(m \cdot n)$.

Per la moltiplicazione matriciale, poste:

$$A \in \mathbb{C}^{m \times n}, \quad B \in \mathbb{C}^{n \times p} \implies A \cdot B = C \in \mathbb{C}^{m \times p}$$

si ha che la complessità è $O(m \cdot n \cdot p)$.

L'ultimo risultato dipende dal fatto che la moltiplicazione richiede di effetuarre effettivamente $m \cdot p$ prodotti scalari, e n è la lunghezza di uno di questi prodotti scalari, cio il numero di colonne di A o di righe di B (che sappiamo essere uguali perché la moltplicazione sia possibile in primo luogo).

Questo significa che per matrici quadrate si ha generalmente complessità $O(n^3)$ (anche se esistono algoritmi che si avvicinano al bound inferiore di $O(n^2)$).

4 Lezione del 07-03-25

Proseguiamo i richiami di algebra lineare.

4.0.1 Approfondimento sulle prestazioni delle operazioni matriciali

Usiamo MATLAB per studiare il tempo di computazione richiesto per effettuare alcune delle operazioni che abbiamo visto.

- Potremmo iniziare con il **prodotto scalare**. Avevamo detto che la complessità era $O(n)$. Scriviamo allora uno script per valutare, in scala logaritmica, il prodotto scalare fra vettori di dimensioni crescenti:

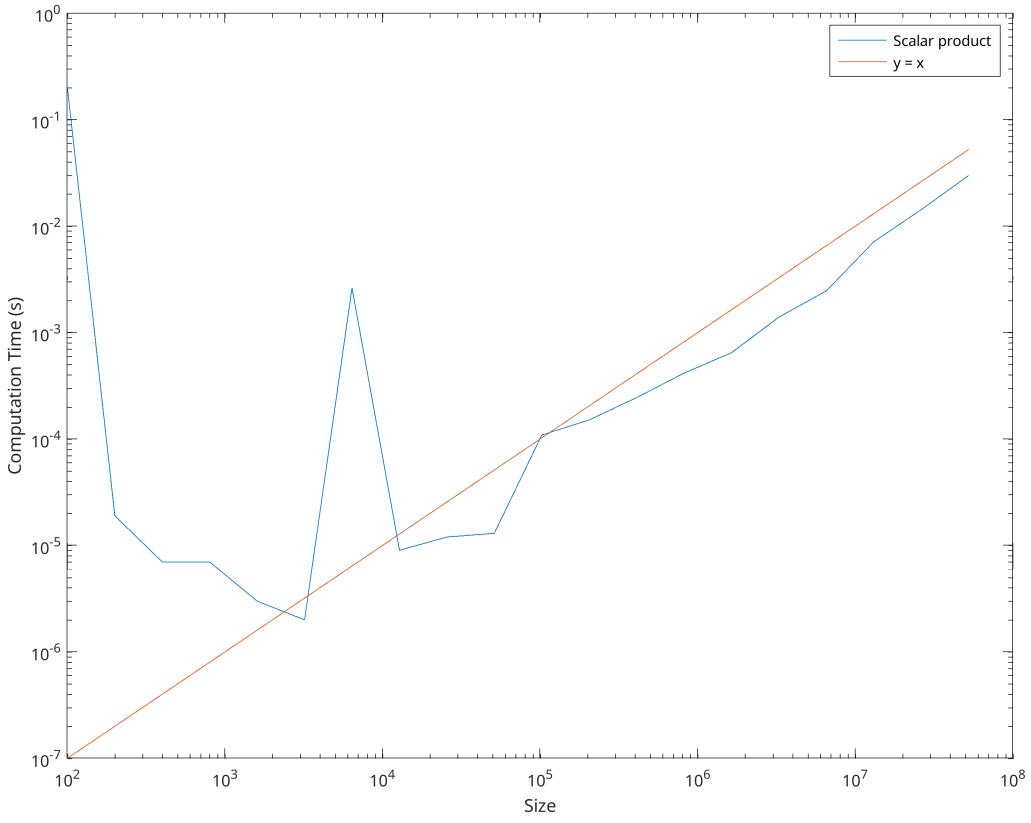
```
function n = time_scalar(num, base, mul)
    if nargin < 3
        mul = 100;
    end
    if nargin < 2
        base = 2;
```

```
    end

    n = zeros(num, 1);
    for i = 1:num
        n(i) = base^(i - 1) * mul;
    end

    times = zeros(num, 1);

    for i = 1:num
        A = randn(1, n(i));
        B = randn(n(i), 1);
        tic;
        C = A * B;
        times(i) = toc;
    end

    loglog(n, times);
    xlabel('Size');
    ylabel('Computation Time (s)');
end
```

Un esempio di esecuzione potrebbe allora essere:

```
>> n = time_scalar(20);
>> hold on;
>> loglog(n, n * 10^-9);
>> hold off;
```

dove si è cercato di fare il *fitting* della curva ottenuta con la funzione $y = x$ (dove la variabile è chiaramante n, e si è aggiunta una costante moltiplicativa k per avvicinarci di più al grafico originale). Il grafico ottenuto è quindi:



Vediamo dal grafico che le prestazioni sono effettivamente vicine a quelle previste dalla complessità computazionale, se non per errori dati all'overhead di inizializzazione del timer `tic` e `toc` di MATLAB (lato sinistro del grafico) e della velocità generale molto elevata delle operazioni, che rende più visibile il rumore dato dallo scheduling della CPU e altri fattori di basso livello.

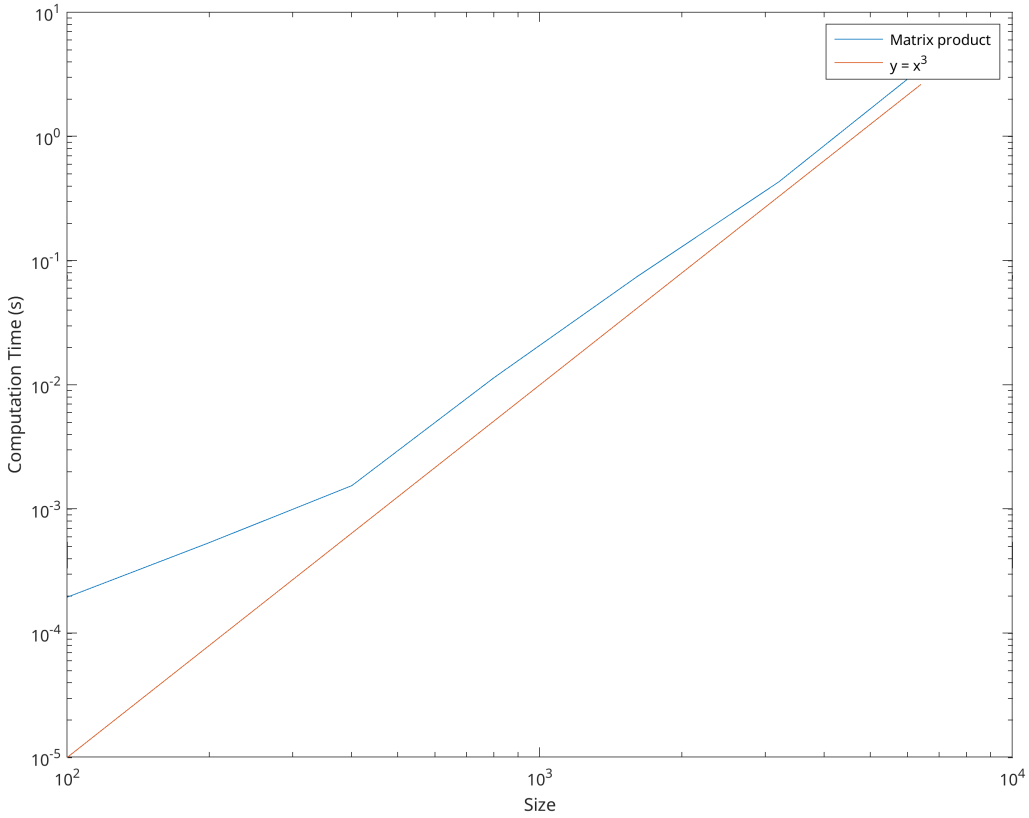
- Vediamo allora l'andamento del **prodotto fra matrici**. Lo script MATLAB è praticamente identico al precedente:

```
function n = time_matrix(num, base, mul)
    if nargin < 3
        mul = 100;
    end
    if nargin < 2
        base = 2;
    end

    n = zeros(num, 1);
    for i = 1:num
        n(i) = base^(i - 1) * mul;
    end

    times = zeros(num, 1);

    for i = 1:num
        A = randn(n(i));
        B = randn(n(i));
        tic;
        C = A * B;
        times(i) = toc;
    end

    loglog(n, times);
    xlabel('Size');
    ylabel('Computation Time (s)');
end
```

che eseguito come:

```
>> n = time_matrix(7);
>> hold on;
>> loglog(n, n.^3 * 10^-11);
>> hold off;
```

ci restituisce il grafico:



che notiamo è già più vicino del prodotto scalare (a causa del tempo di esecuzione maggiore richiesto, che "maschera" bene gli effetti a basso livello della CPU).

4.1 Proprietà del prodotto matriciale

Abbiamo che in genere il prodotto fra matrici non è **commutativo**, cioè:

$$AB \neq BA$$

di contro, vale l'**associativa**:

$$(A \cdot B) \cdot C = A \cdot (B \cdot C)$$

e la **distributiva**, separatamente ai due lati:

$$(A + B) \cdot C = A \cdot C + B \cdot C$$

$$C \cdot (A + B) = C \cdot A + C \cdot B$$

Inoltre, notiamo che quello delle matrici non è un *dominio integrale*:

$$A \cdot B = 0 \not\Longrightarrow A = 0 \vee B = 0$$

Vediamo ad esempio come sfruttare la proprietà associativa può permetterci di ottenere algoritmi più veloci. Supponiamo di voler calcolare:

$$(A \cdot B) \cdot C = A \cdot (B \cdot C)$$

con $A \in \mathbb{C}^{m \times n}$; $B \in \mathbb{C}^{n \times p}$, $C \in \mathbb{C}^{p \times q}$.

L'uguaglianza ci darà due metodi:

- $(A \cdot B) \cdot C$:
  - $A \cdot B = P \in \mathbb{C}^{m \times p}, \quad O(m \cdot n \cdot p)$
  - $P \cdot C = R \in \mathbb{C}^{m \times q}, \quad O(m \cdot p \cdot q)$

  $\implies O(mnp + mpq) = O\left(mp(n+q)\right)$
- $A \cdot (B \cdot C)$:
  - $B \cdot C = P \in \mathbb{C}^{n \times q}, \quad O(n \cdot p \cdot q)$
  - $A \cdot P = R \in \mathbb{C}^{m \times q}, \quad O(m \cdot n \cdot q)$

  $\implies O(npq + mnq) = O\left(nq(p+m)\right)$

che sono quindi uguali per $m = n = p$, ma (anche radicalmente diversi) diversi al variare di questi parametri.

In generale, quindi, se si hanno matrici con poche righe o poche colonne, è opportuno cercare di mantenere questa properietà più a lungo possibile nei risultati intermedi.

Possiamo fare considerazioni simili a quelle fatte sull'associativa con la distributiva. Ad esempio, posta l'uguaglianza:

$$(A + B) \cdot C = AC + BC$$

con $A, B \in \mathbb{C}^{m \times n}$ e $C \in \mathbb{C}^{n \times p}$, abbiamo due metodi:

- $(A + B) \cdot C$:
  - $A + B = P \in \mathbb{C}^{m \times n}, \quad O(m \cdot n)$
  - $P \cdot C = R \in \mathbb{C}^{m \times p}, \quad O(m \cdot n \cdot p)$

  $\implies O(mn + mnp) = O\left(mn(1+p)\right)$
- $AC + BC$:
  - $A \cdot C = P_1 \in \mathbb{C}^{m \times p}, \quad O(m \cdot n \cdot p)$
  - $B \cdot C = P_2 \in \mathbb{C}^{m \times p}, \quad O(m \cdot n \cdot p)$

  $\implies O(mnp + mnp + mp) = O\left(mp(1+2n)\right)$

che sono sì asintoticamente uguali per $m = n = p$ ($O(n^3)$), ma non esattamente uguali in quanto da un lato si ha $O(n^2 + n^3)$ e dall'altro $O(n^2 + 2n^3)$. Questo risulta chiaramente dal fatto che col secondo metodo decidiamo di effettuare il prodotto matriciale (che è l'operazione più costosa) non una ma 2 volte.

Un altra proprietà del prodotto di matrici e che si può vedere il risultato riga per riga o colonna per colonna nei seguenti modi:

$$A = \begin{pmatrix} A_1 \\ ... \\ A_m \end{pmatrix}, \quad B = \begin{pmatrix} B_1 & ... & B_p \end{pmatrix}$$

$$\implies A \cdot B = \begin{pmatrix} A \cdot B_1 & ... & A \cdot B_p \end{pmatrix} = \begin{pmatrix} A_1 B \\ ... \\ A_m B \end{pmatrix}$$

Questo ci dice che le colonne (le righe) di $C = A \times B$ sono combinazioni lineari delle colonne (delle righe) di A.

4.2 Determinante

Abbiamo visto la definizione di **determinante** per matrici quadrate:

Definizione 4.1: Determinante

Si definisce la funzione:

$$\det(A) : \mathbb{C}^{n\times n} \to \mathbb{C}$$

con:

$$\det(A) = \begin{cases} a_{11}, & n = 1 \\ a_{11}a_{22} - a_{12}a_{21}, & n = 2 \\ \sum_{j=1}^{n}(-1)^{i+j}\det(A_{ij}), & n > 2 \quad \text{(sviluppo di Laplace)} \end{cases}$$

determinante.

Osserviamo che il calcolo del determinante attraverso lo sviluppo di Laplace ha complessità algoritmica $O(n!)$.

4.2.1 Proprietà del determinante

Sappiamo che A invertibile $\Leftrightarrow \det(A)$ (cioè A **singolare**). Inoltre valgono:

$$\det(A^T) = \det(A)$$
$$\det(A^H) = \overline{\det(A)}$$

4.3 Rango

Vediamo poi come ci può servire il determinante delle **sottomatrici**:

Definizione 4.2: Sottomatrice

Una sottomatrice di A è una matrice ottenuta prendendo la restrizione di A a un sottoinsieme di righe e di colonne, cioè data $A \in \mathbb{C}^{n\times n}$, $I, J \subseteq \{1, ..., n\}$ con $|I| = n_1$ e $|J| = n_2$, sarà allora:

$$A(I, J) \in \mathbb{C}^{n_1\times n_2}$$

ottenuta incrociando le righe in I con le colonne in J.

Definiamo quindi i **minori**:

Definizione 4.3: Minore

Si dice minore di ordine k, con $k \in \{1, ..., n\}$ il determinante di una sottomatrice quadrata $k \times k$.

E le sottomatrici **principali** e **principali di testa**:

Definizione 4.4: Sottomatrice principale

Una sottomatrice si dice principale se gli insiemi I, J usati per estrarla sono $I = J$.

Definizione 4.5: Sottomatrice principale di testa

Una sottomatrice quadrata di ordine k si dice sottomatrice principale di testa se $I = J = \{1, ..., k\}$.

Allo stesso modo si possono definire **matrici di coda** (basterà prendere indici da k ad n).

Possiamo quindi definire il **rango** di matrice:

Definizione 4.6: Rango

Il rango $\text{rank}(A)$ di una matrice A è definito come il massimo numero di colonne (o di righe) linearmente indipendenti, ed è uguale all'ordine massimo dei minori $\neq 0$ in A.

4.3.1 Proprietà del rango

Caso particolare del rango sarà chiaramente quello dove prendiamo come sottomatrice la matrice stessa: $\det(A) \neq 0 \Leftrightarrow \text{rank}(A) = n$, e quindi gli insiemi delle colonne e delle righe di A sono tutte linearmente indipendenti.

Si ha poi che:

$$\text{rank}(A) = \dim(\text{Im}(A))$$

dove $\text{Im}(A)$ è la dimensione dell'**immagine** di A:

$$\text{Im}(A) = \{y \in \mathbb{C}^m : y = Ax,\ x \in \mathbb{C}^n\}$$

4.3.2 Teorema di Binet-Cauchy

Concludiamo enunciando il teorema di Binet-Cauchy sul determinante rispetto al prodotto matriciale.

Teorema 4.1: di Binet-Cauchy

Prese due matrici, $A \in \mathbb{C}^{n\times n}$ e $B \in \mathbb{C}^{n\times n}$, e $C = A \cdot B$ di dimensioni uguali, sarà:

$$\det(C) = \det(A) \cdot \det(B)$$

Nel caso generale avremo $A \in \mathbb{C}^{n\times p}$ e $B \in \mathbb{C}^{p\times n}$, da cui:

$$\det(C) = \begin{cases} 0, & p \leq n \\ \sum_j A_{[j]} \cdot B_{[j]} \end{cases}$$

dove $A_{[j]}$ e $B_{[j]}$ sono i minori di ordine n relativi ala stessa scelta di indici in A e in B.

La dimostrazione di questo teorema si basa sullo sviluppo dei cofattori della matrice $A \cdot B$, dove si possono riarrangiare i termini per ottenere gli sviluppi dei cofattori delle matrici A e B.

4.4 Matrice inversa

Diamo la definizione di **matrice inversa**:

Definizione 4.7: Matrice inversa

Data $A \in \mathbb{C}^{n\times n}$ tale che $\det(A) \neq 0$, si definisce $A^{-1} \in \mathbb{C}^{n\times n}$ tale che:

$$A \cdot A^{-1} = A^{-1} \cdot A = I$$

Se si guarda ad A come una funzione lineare $A : \mathbb{C}^n \to \mathbb{C}^n$, sarà che:

$$A : x \to Ax, \quad A^{-1} : Ax \to x$$

questo da:

$$\left(A^{-1} \circ A\right) = A^{-1}y = A^{-1}Ax = Ix = x$$

□

4.4.1 Proprietà della matrice inversa

Vediamo alcune proprietà di A^{-1}:

1. Ricordiamo di nuovo che A^{-1} esiste se e solo se $\det(A) \neq 0$, cioè equivalentemente se $\text{rank}(A) = n$, o A ha spazi riga e colonna linearmente indipendenti;
2. Vediamo poi il calcolo di $\det(A^{-1})$: da $\det(I) = 1$, e:

$$\det(A \cdot A^{-1}) = \det(A) \cdot \det(A^{-1}), \quad \text{(Binet)}$$

$$\implies \det(A^{-1}) = \frac{1}{\det(A)}$$

3. Vediamo poi che:

$$(A \cdot B)^{-1} = B^{-1}A^{-1}$$

per matrici quadrate $A, B \in \mathbb{C}^{n\times n}$;

4. Riguardo alle trasposte e alle Hermitiane vale:

$$(A^T)^{-1} = (A^{-1})^T = A^{-T}$$

$$(A^H)^{-1} = (A^{-1})^H = A^{-H}$$

e:

$$(AB)^T = B^T A^T$$

$$(AB)^H = B^H A^H$$

4.5 Matrici particolari

Definiamo alcune matrici particolari:

Definizione 4.8: Matrici particolari

Data $A \in \mathbb{C}^{n \times n}$, si dice di A che è:

- **Hermitiana:** $A = A^H$;
- **Antiermitiana:** $A = -A^H$;
- **Unitaria** $A^H A = AA^H = I$, $A^{-1} = A^H$;
- **Normale** $A^H A = AA^H$;
- **Simmetrica** $A = A^T$;
- **Antisimmetrica** $A = -A^T$;
- **Ortogonale** $A^T A = AA^T = I$, $A^T = A^{-1}$

dove notiamo che **simmetrica** e **antisimmetrica** significano *hermitiana* e *antihermitiana* in $\mathbb{R}$, e **ortogonale** significa *unitaria* in $\mathbb{R}$.

Per di più vale:

$$\{\text{matrici simmetriche reali}\} \subseteq \{\text{matrici hermitiane}\} \subseteq \{\text{matrici normali}\}$$

e:

$$\{\text{matrici ortogonali}\} \subseteq \{\text{matrici unitarie}\}$$

dove notiamo che unitarie ed ortogonali hanno l'inversa "facile", nel senso che basta trasporre o trovare l'hermitiana.

4.6 Matrici di permutazione

Definiamo le **matrici di permutazione**:

Definizione 4.9: Matrice di permutazione

$A \in \mathbb{R}^{n \times n}$ si dice matrice di permutazione se A si ottiene dalla matrice di identità permutandone righe e colonne.

4.6.1 Proprietà delle matrici di permutazione

Tutte le matrici di permutazione sono matrici ortogonali (questo si vede dalla loro unimodularità, o osservando che P^T si ottiene dalla permutazione inversa).

Inoltre, vediamo che il prodotto di A con una matrice di permutazione si limita a scambiare righe e colonne in accordo con la permutazione della matrice. In particolare, $A \cdot P$ dà la permutazione delle colonne, mentre $P \cdot A$ dà la permutazione delle righe.

4.7 Sistemi lineari

Abbiamo un sistema di m equazioni lineari in n incognite:

$$\begin{cases} a_{11}x_1 + ...a_{1n}x_n = b_1 \\ ... \\ a_{m1}x_1 + ...a_{mn}x_n = b_m \end{cases}$$

Ci è spesso utile riscrivere sistemi di questo tipo in forma matriciale:

$$A = \begin{pmatrix} a_{11} & ... & a_{1n} \\ ... & ... & ... \\ a_{m1} & ... & a_{mn} \end{pmatrix} \in \mathbb{C}^{m\times n}, \quad x = \begin{pmatrix} x_1 \\ ... \\ x_n \end{pmatrix} \in \mathbb{C}^n, \quad b = \begin{pmatrix} b1 \\ ... \\ b_m \end{pmatrix} \in \mathbb{C}^m$$

Il problema principe sarà quello, date A e b, di trovare x.

4.7.1 Teorema di Rouché-Capelli

Richiamiamo il teorema:

Teorema 4.2: di Rouché-Capelli

$Ax = b$ ammette almeno una soluzione se $\text{rank}(A) = \text{rank}(A|b)$, con $(A|b) \in \mathbb{C}^{m\times(n+1)}$ la matrice ottenuta aumentando A con b.
Per quanto riguarda l'unicità, supposto $m \geq n$ (sistema *sovradeterminato*):

- Se $\text{rank}(A) = n$ allora la soluzione è unica;
- Se $\text{rank}(A) < n$ allora ci sono ∞ soluzioni, e l'insieme delle soluzioni forma uno spazio vettoriale affine di dimensione $n - \text{rank}(A)$;

Nel caso il vettore $b = 0$, il sistema si dice **omogeneo**, e la soluzione nulla esiste sempre.

Abbiamo poi che:

Definizione 4.10:

L'insieme delle soluzioni di $Ax = 0$ si chiama Kernel (nucleo) della matrice:

$$\ker(A) = \{x \in \mathbb{R}^n : Ax = 0\}$$

notiamo che il kernel è un sottospazio vettoriale di dimensione $n - \text{rank}(A)$.

Osserviamo infine che nel caso $m = n$, se $\det(A) \neq 0$, cioè $\text{rank}(A) = n$, la soluzione di $Ax = b$ è unica per ogni $b \in \mathbb{R}^m$ e si scrive:

$$x = A^{-1}b$$

Vedremo che in generale calcolare l'inversa non è conveniente rispetto ad altri metodi di approssimazione.

4.7.2 Regola di Cramer

Il vettore soluzione si può scrivere ache componente per componente come:

$$x_j = \frac{\det(A_j)}{\det(A)}$$

dove A_j è la matrice ottenuta da A sotituendo la colonna j con b.

Questa via costa come calcolare $O(n)$ determinanti di matrici $n \times n$, ed è quindi poco conveniente ($O(n^3)$).

5 Lezione del 10-03-25

5.1 Autovalori e autovettori

Continuiamo a parlare di matrici quadrate, e introduciamo gli **autovalori** e **autovettori**:

Definizione 5.1: Autovalori e autovettori destri

Data $A \in \mathbb{C}^{n \times n}$, $\lambda \in \mathbb{C}$ si dice autovalore se esiste un vettore v in $\mathbb{C}^n$, con $v \neq 0$, tale che:

$$Av = \lambda v$$

L'equazione $Av = \lambda v$ non rappresenta un sistema lineare, in quanto sia v che λ sono variabili (e fra l'altro moltiplicate fra loro).

In particolare, v si dice **autovettore destro** per A rispetto a λ. Similmente, si può definire w **autovettore sinistro**:

Definizione 5.2: Autovalori e autovettori sinistri

Data $A \in \mathbb{C}^{n \times n}$, $\lambda \in \mathbb{C}$ si dice autovalore se esiste un vettore w in $\mathbb{C}^n$, con $w \neq 0$, tale che:

$$w^H A = w^H \lambda$$

Solitamente ci basta trovare gli autovettori destri in quanto:

$$(w^H A)^H = A^H w$$

$$(\lambda w^H)^H = \overline{\lambda} w$$

cioè w è un autovettore destro per λ se e solo se w è autovettore destro per A^H rispetto all'autovalore $\overline{\lambda}$. In questo caso, se la matrice è *simmetrica* o *hermitiana*, gli autovalori destri e sinistri coincidono.

5.1.1 Caratterizzazione degli autovalori

Potremmo interrogarci sull'esistenza di questi oggetti. Poniamo allora:

$$Av = \lambda v \implies Av - \lambda v = 0 \implies (A - \lambda I)v = 0$$

Chiaramente il sistema è risolto per $v = 0$, ma abbiamo escluso a priori tale possibilità. Abbiamo allora un sistema $A - \lambda I$, che ha almeno una soluzione per Rouché-Capelli, che è 0 se $\det(A - \lambda I) \neq 0$, e un valore $\neq 0$ altrimenti. Vogliamo quindi impostare:

$$p(\lambda) = \det(A - \lambda I) = 0$$

detta **equazione caratteristica** (del *polinomio caratteristico* posto nullo). Gli autovalori non saranno altro che le soluzioni dell'equazione caratteristica.

L'esistenza di tali soluzioni è data dal fatto che $\deg(p(\lambda)) = n$, e quindi dal teorema fondamentale dell'algebra esistono $c_0, c_1, ..., c_n$ tali che:

$$p(\lambda) = c_0 + c_1\lambda + ... + c_n\lambda^n = c_n(\lambda - \lambda_1)...(\lambda - \lambda_n) \sim (-1)^n(\lambda - \lambda_1)...(\lambda - \lambda_n)$$

e dal punto di vista dei minori:

$$= (-1)^n\lambda^n + (-1)^{n-1}\sigma_1\lambda^{n-1} + ... + (-1)^1\sigma_{n-1}\lambda + (-1)^0\sigma_n$$

$$= (-1)^n\lambda^n + (-1)^{n-1}\sigma_1\lambda^{n-1} + ... - \sigma_{n-1}\lambda + \sigma_n$$

dove σ_j è la somma dei determinanti delle sottomatrici di A di ordine j. □

Alcuni di questi σ_j sono semplici da calcolare:

$$\sigma_1 = \sum_{i=1}^{n} a_{ii} = \mathrm{tr}(A) \quad \text{(traccia)}$$

$$\sigma_n = \prod_{j=1}^{m} \lambda_j = \det(A)$$

vediamo quest'ultima relazione: si ha che il determinante di una matrice è uguale al prodotto dei suoi autovalori, ergo:

$$\det(A) = 0 \Leftrightarrow \exists\lambda_j = 0$$

Riepilogando una matrice $\in \mathbb{C}^{n\times n}$ ha sempre esattamante n autovalori, contati con la loro molteplicità. In particolare:

Definizione 5.3: Molteplicità algebrica

Un autovalore $\overline{\lambda}$ ha molteplicità algebrica k ($\alpha(\overline{\lambda}) = k$) se k è la sua molteplicità μ come radice del polinomio caratteristico.

Chiaramente:

$$\sum_{\overline{\lambda}} \alpha(\overline{\lambda}) = n$$

5.1.2 Caratterizzazione degli autovettori

Potremmo chiederci quanti sono gli autovettori. Abbiamo allora che:

$$Av = \lambda v$$

e preso $\theta \in \mathbb{C}$ allora:

$$A(\theta v) = \theta Av = \theta\lambda v = \lambda(\theta v)$$

cioè anche θv è autovalettore per $\lambda^* = \theta\lambda$. Questo ci dice che gli autovettori non sono mai unici. Inoltre, possiamo dire che dati $Av_1 = \lambda v_1$ e $Av_2 = \lambda v_2$:

$$A(v_1 + v_2) = Av_1 + Av_2 = \lambda v_1 + \lambda v_2 = \lambda(v_1 + v_2)$$

cioè sono soddisfatte le proprietà di omogeneità e additività, e quindi l'insieme degli autovettori relativi a un certo autovalore λ forma uno spazio vettoriale (detto **autospazio** rispetto a λ).

Questo ci permette di definire un'altra molteplicità:

Definizione 5.4: Molteplicità geometrica

Dato $\lambda \in \mathbb{C}$ autovalore di $A \in \mathbb{C}^{n \times n}$ si dice molteplicità geometrica k ($\gamma(\lambda) = k$) la dimensione dell'autospazio associato a λ, ovvero:

$$\gamma(\lambda) = \dim\left(\ker(A - \lambda I)\right)$$

Per Rouché-Capelli, si ha che:

$$\gamma(\lambda) = n - \operatorname{rank}(A - \lambda I)$$

Valgono poi le relazioni fra molteplicità algebrica e molteplicità geometrica:

$$1 \leq \gamma(\lambda) \leq \alpha(\lambda) \leq n$$

Inoltre se si hanno $\lambda_i \neq \lambda_j$ autovalori distinti ($\forall i \neq j,\ i, j \in \{1, ..., n\}$), allora necessariamente:

$$\gamma(\lambda_i) = \alpha(\lambda_i) = 1, \quad \forall i = 1, ..., n$$

cioè se tutti gli autovalori di A sono distinti, le loro molteplicità, sia α che γ, devono essere uguali a 1.

5.1.3 Matrici simili

Possiamo quindi definire la **similarità** fra matrici:

Definizione 5.5: Matrici simili

Date $A, B \in \mathbb{C}^{n \times n}$, si dicono simili ($B \sim A$) se esiste $S \in \mathbb{C}^{n \times n}$ invertibile tale che:

$$B = S^{-1}AS$$

in questo caso vale anche l'inverso:

$$A = SBS^{-1}$$

Le matrici fra di loro simili hanno diverse proprietà in comune. Ad esempio, $A \sim B$ hanno gli stessi autovalori con le stesse molteplicità α e γ. Inoltre se v è autovettore per A associato a λ, allora $S^{-1}v$ è autovettore per B associato sempre a λ. Questo si dimostra da:

$$\det(B - \lambda I) = 0 = \det(S^{-1}AS - \lambda I) = \det(S^{-1}AS - \lambda S^{-1}S) = \det(S^{-1}(A - \lambda I)S)$$

e da Binet-Cauchy:

$$\det(S^{-1}(A - \lambda I)S) = \det(S^{-1})\det(A - \lambda I)\det(S) = \det(A - \lambda I)$$

in quanto S^{-1} e S sono costanti moltiplicative, e quindi gli autovalori λ di A e B coincidono. Inoltre:

$$B - \lambda I = 0 \Rightarrow S^{-1}AS - \lambda I = 0 \Rightarrow S^{-1}(A - \lambda I)S = 0 \Rightarrow A - \lambda I = 0$$

cioè si ha lo stesso spazio delle soluzioni.

Presa (λ, v) *autocoppia* (coppia autovalore-autovettore) per A abbiamo allora:

$$B(S^{-1}v) = S^{-1}AS(S^{-1}v) = S^{-1}Av = S^{-1}\lambda v = \lambda(S^{-1}v)$$

e quindi $S^{-1}v$ è autovettore per B associato a λ. □

5.1.4 Matrici diagonalizzabili

Definiamo le matrici diagonalizzabili come matrici simili ad una matrice diagonale, cioè:

Definizione 5.6: Matrice diagonalizzabile

Data $A \in \mathbb{C}^{n\times n}$, A si dice diagonalizzabile se $\exists S$ invertibile tale che:

$$S^{-1}AS = A_D$$

con $A_D = \text{diag}(\lambda_1, ..., \lambda_n)$ diagonale.

Osserviamo che se A è diagonalizzabile, la matrice diagonale (sopra, A_D) contiene sulla diagonale gli autovalori di A.

Vediamo poi che:

$$S^{-1}AS = A_D \implies AS = SA_D$$

risultato piuttosto triste, e posto $S = \begin{pmatrix} s_1 & ... & s_n \end{pmatrix}$:

$$AS = \begin{pmatrix} As_1 & ... & As_n \end{pmatrix} = \begin{pmatrix} \lambda_1 s_1 & ... & \lambda_n s_n \end{pmatrix}$$

da dove notiamo che da $As_i = \lambda_i s_i$, la matrice S ha per colonne gli autovettori di A (comodo per il calcolo della matrice S quando A è definita in base canonica).

Non tutte le matrici quadrate sono diagonalizzabili. La diagonalizzabilità dipende infatti dalla possibilità di scegliere n autovettori di A linearmente indpendenti, cioè di trovare una base di $\mathbb{C}^n$ di autovalori di A.

Da questo deriva:

Teorema 5.1: Diagonalizzabilità di matrici

$A \in \mathbb{C}^{n\times n}$ è diagonalizzabile se e solo se:

$$\alpha(\lambda) = \gamma(\lambda), \quad \forall\lambda$$

dove i λ sono gli autovalori di A.

Questo semplicemente significa che una matrice è diagonalizzabile se le molteplicità algebriche e geometriche di tutti gli autovalori coincidono.

Si ha quindi che, per A diagonalizzabile, con $x \in \mathbb{C}^n$ si ha che $\exists c_1, ..., c_n \in \mathbb{C}$ tali che:

$$x = \sum_{i=1}^{n} c_i v_j$$

sugli autovettori v_j.

L'utilità sta nel fatto che possiamo scegliere il caso particolare dove la base di autovettori è unitaria/ortogonale (unitaria in $\mathbb{C}$ e ortogonale in $\mathbb{R}$). Da questo deriva:

Teorema 5.2: Teorema spettrale

Se la matrice A è unitaria, cioe $A = A^H$, allora:

1. A è sempre diagonalizzabile;
2. Gli autovalori di A sono reali;
3. Si può scegliere una base di autovettori unitaria/ortogonale, cioè tale che:

$$< v_i, v_j >= \begin{cases} 0, & i \neq j \\ 1, & i = j \end{cases}$$

o in modo equivalente si può trovare $V \in \mathbb{C}^{n \times n}$ tale che:

$$V^H A V = \text{diag}(\lambda_1, ..., \lambda_n)$$

e

$$V^H V = V V^H = I$$

Da questo teorema vengono le definizioni:

Definizione 5.7: Matrice definita positiva/negativa

Una matrice hermitiana viene detta definita positiva (definita negativa) quado gli autovalori λ_i sono tutti > 0 (< 0).

5.1.5 Proprietà di autovalori e autovettori

Vediamo come si comportano autovalori e autovettori sottoposti a diverse operazioni fra matrici.

1. **Somma per identità:** data (λ, v) autocoppia per $A \in \mathbb{C}^{n \times n}$, $\delta \in \mathbb{C}$, allora $A + \delta I$ ha come autocoppia $\lambda + \delta, v$. Questo semplicemente da:

$$(A + \delta I)v = Av + \delta v = \lambda v + \delta v = (\lambda + \delta)v$$

2. **Somma di potenze:** supponiamo $q(x) = \sum_{i=0}^{d} q_i x^i$. Si può allora definire:

$$q(A) = \sum_{i=0}^{d} q_i A^i$$

Allora con (λ, v) autocoppia di A, si avrà che $(q(\lambda), v)$ è autocoppia per $q(A)$.

Notiamo che la proprietà (1) è il caso $q(x) = x + \delta$.

3. **Inversa:** data (λ, v) autocoppia per A, si avrà che (λ^{-1}, v) è autocoppia per A^{-1}. Questo da:

$$Av = \lambda v \implies A^{-1}Av = \lambda A^{-1} v \implies v = \lambda A^{-1} v$$

da cui:

$$\frac{1}{\lambda} v = A^{-1} v$$

□

4. **Autovalori complessi coniugati:** se $A \in \mathbb{R}^{n\times n}$, allora $\lambda \in \mathbb{C}$ è autovalore di A se e solo se anche $\overline{\lambda}$ coniugato è autovalore di A. Equivalentemente, gli autovalori complessi arrivano sempre in coppie coniugate. Questo da:

$$p(\lambda) = 0 = (-1)^n\lambda^n + (-1)^{n-1}\sigma_1\lambda^{n-1} + ... - \sigma_{n-1}\lambda + \sigma_n$$

con $\lambda \in \mathbb{C} \setminus \mathbb{R}$, dovra essere che:

$$p(\lambda) = 0 = p(\lambda_n)$$

□

Definiamo infine il **raggio spettrale**, cioè il *massimo modulo* degli autovalori.

Definizione 5.8: Raggio spettrale

La quantità $\rho(A) = \max_{j=\{1,...,n\}} |\lambda_j|$ è detta raggio spettrale di A.

Questa definizione è utile sempre al calcolo di esponenti di matrici. Ad esempio, vale il teorema:

Teorema 5.3: Limiti e raggio spettrale

Per un matrice A con raggio spettrale $\rho(A)$, vale:

$$\lim_{x\to+\infty} A^x = 0 \Leftrightarrow \rho(A) < 1$$

6 Lezione del 14-03-25

6.1 Norme

Concludiamo il ripasso di algebra lineare parlando del concetto di **norma**.

6.1.1 Norme vettoriali

In molti casi è utile definire (quantificare) la "grandezza" di un vettore, prendendo il vettore 0 come nullo e ogni vettore come più grande di esso. Formalmente, quello che si fa è definire funzioni con proprietà specifiche dette **norme**.

Definizione 6.1: Norma

Una funzione $|\cdot| : \mathbb{C}^n \to \mathbb{R}^+$ è detta norma se verifica le 3 proprietà:

1. $|x| = 0 \Leftrightarrow x = 0$
2. $|\alpha x| = |\alpha| \cdot |x|, \quad \forall \alpha \in \mathbb{C}, \forall x \in \mathbb{C}^n$
3. $|x + y| \leq |x| + |y|, \quad \forall x, y \in \mathbb{C}^n$ (diseguaglianza del triangolo)

Possiamo dare alcuni esempi di norme:

- $|x|_1 = \sum_{i=1}^n |x_i|$ (norma di Manhattan)

- $|x|_2 = \sqrt{\sum_{i=1}^n |x_i|^2}$ (norma Euclidea)
- $|x|_p = (\sum_{i=1}^p |x_i|^p)^{\frac{1}{p}}$ (p-norma)
- $|x|_\infty = \max_{i=1,...,n} |x_i|$ (norma a infinito)

Notiamo che l'insieme dei vettori di norma 1 cambia geometria al variare della norma che scegliamo. Ad esempio, in $\mathbb{R}^2$, l'insieme dei vettori di norma 1 dà un quadrato ruotato di 45 gradi, dei vettori di norma 2 una circonferenza, e via via al crescere di p si approssima un quadrato unitario centrato sull'origine (che la norma a infinito effettivamente raggiunge).

Le norme sono equivalenti in quanto esiste il teorema:

Teorema 6.1: Norme equivalenti

Date due norme $|\cdot|_a$ e $|\cdot|_b$, si ha che $\exists \alpha, \beta \in \mathbb{R}^+$ tali che:

$$\alpha|x|_a \leq |x|_b \leq \beta|x|_a, \quad \forall x \in \mathbb{C}^n$$

Questa proprietà è sempre valida su spazi vettoriali finiti (non vale necessariamente lo stesso su spazi infiniti). Se vale il teorema 6.1, quindi, allora vale anche che:

$$\beta^{-1}|x|_b \leq |x|_a \leq \alpha^{-1}|x|_b, \quad \forall x \in \mathbb{C}^n$$

Per le norme più comuni ($|\cdot|_1$, $|\cdot|_2$, $|\cdot|_\infty$) valgono le diseguaglianze:

- $|x|_2 \leq |x|_1 \leq \sqrt{n}|x|_2$

  Dimostriamo entrambe le diseguaglianze.

  - $|x|_2 \leq |x|_1$

    Questo si ha da:

    $$\sqrt{\sum_{i=1}^n |x_i|^2} \leq \sum_{i=1}^n |x_i|$$

    prendendo i quadrati:

    $$\sum_{i=1}^n |x_i|^2 \leq \left(\sum_{i=1}^n |x_i|\right)^2 = \sum_{i=1}^n |x_i|^2 + 2\sum_{i<j}^n |x_i||x_j|$$

    che è chiaramente vero con $2\sum_{i<j}^n |x_i||x_j| > 0$.
  - $|x|_1 \leq \sqrt{n}|x|_2$

    Questo si nota prendendo:

    $$\sum_{i=1}^n |x_i| = (||x||, 1) = |x_1| \cdot 1 + ... + |x_n| \cdot 1$$

    dove $||x||$ rappresenta il modulo componente a componente di x. Varrà allora dalla diseguaglianza di Cauchy-Schwarz:

    $$(||x||, 1) \leq |\,||x||\,| \cdot |1| = \sqrt{\sum_{i=1}^n |x_i|^2} \cdot \sqrt{\sum_{i=1}^n |1|^2} = \sqrt{n}\sqrt{\sum_{i=1}^n |x_i|^2}$$

    che è la tesi. □

- $|x|_\infty \leq |x|_1 \leq n|x|_\infty$

  Questo risulta chiaro da:

$$\max_{i=1,\dots,n} |x_i| \leq \sum_{i=1}^{n} |x_i| \leq n \max_{i=1,\dots,n} |x_i|$$

□

- $|x|_\infty \leq |x|_2 \leq \sqrt{n}|x|_\infty$

  Anche questo risulta chiaro da:

$$\max_{i=1,\dots,n} |x_i| \leq \sqrt{\sum_{i=1}^{n} |x_i|^2} \leq \sqrt{n} \max_{i=1,\dots,n} |x_i|$$

  prendendo i quadrati:

$$\left(\max_{i=1,\dots,n} |x_i|\right)^2 \leq \sum_{i=1}^{n} |x_i|^2 \leq n \left(\max_{i=1,\dots,n} |x_i|\right)^2$$

□

6.1.2 Norme matriciali

Si possono definire norme anche per matrici.

Definizione 6.2: Norma matriciale

Una funzione $|\cdot| : \mathbb{C}^{m\times n} \to \mathbb{R}^+$ è detta norma matriciale se verifica le 3 proprietà:

1. $|A| = 0 \Leftrightarrow A = 0$
2. $|\alpha A| = |\alpha| \cdot |A|, \quad \forall \alpha \in \mathbb{C}, \forall A \in \mathbb{C}^{m\times n}$
3. $|A + B| \leq |A| + |B|, \quad \forall A, B \in \mathbb{C}^{m\times n}$
4. Se $m = n$ chiediamo inoltre, visto che è ben definito il prodotto matriciale:
   $|A \cdot B| \leq |A| \cdot |B|, \quad \forall A, B \in \mathbb{C}^{m\times n}$
   da qui in poi assumiamo solo questo caso.

Esiste un modo per trovare norme matriciali a partire da norme vettoriali.

Definizione 6.3: Norma indotta

Una norma matriciale $|\cdot|_m$ si dice indotta se esiste una norma vettoriale $|\cdot|_v$ tale che:

$$|A|_m = \sup_{|x|_v \neq 0} \frac{|Ax|_v}{|x|_v} = \sup_{|x|_v = 1} |Ax|_v$$

Intuitivamente, diciamo che una matrice è tanto più "grande" tanto più *allarga* i vettori a distanza unitaria.

Le norme matriciali indotte dalle norme vettoriali più comuni ($|\cdot|_1, |\cdot|_2, |\cdot|_\infty$) sono le seguenti:

- $|A|_1 = \max_{j=1,\dots,n} \sum_{i=1}^{n} |a_{ij}|$, cioè il massimo della somme delle colonne di A;
- $|A|_\infty = \max_{i=1,\dots,n} \sum_{j=1}^{n} |a_{ij}|$, cioè il massimo delle somme delle righe di A;
- $|A|_2 = \sqrt{\rho(A^H A)}$, cioè la radice dell'autovalore di modulo massimo di $A^H A$. Notiamo che questo metodo è sensibilmente più difficile di calcolare la somma delle entrate delle matrice in riga o in colonna.

  Prendiamo $A^H A$ anzichè A in quanto:

  $$|A|_2 = \sup_{|x|_2=1} |Ax|_2$$

  e:

  $$|Ax|_2 = < Ax, Ax > = < x, A^H Ax >$$

  il resto della dimostrazione non viene dato in quanto si basa su un risultato che esula dal corso (*quoziente di Rayleigh*), ma resta il fatto che ci serve una forma del tipo $A^H A$ anziché la sola matrice A.

Un esempio di norma matriciale non indotta è la norma di Frobenius:

$$|A|_F = \sqrt{\sum_{ij}^{n} |\alpha_{ij}|^2}$$

che possiamo assumere come una grande norma $|\cdot|_2$ su tutta la matrice.

Un modo per verificare che questa norma non è effettivamente indotta è osservare che:

$$|I|_F = \sqrt{n}$$

mentre per ogni norma indotta vale che:

$$|I|_i = \sup_{|x|\neq 0} \frac{|Ix|}{|x|} = \sup_{|x|\neq 0} \frac{|x|}{|x|} = 1 \neq \sqrt{n} \quad \forall n > 1$$

□

Notiamo che questa verifica vale in una sola direzione: se la norma dell'identità valesse effettivamente 1, potremmo dire che questa proprietà non è violata, ma non che di conseguenza la norma è indotta, cioè:

$$|I|_m \neq 1 \implies |\cdot|_m \text{ non è indotta}$$

$$|I|_m = 1 \not\implies |\cdot|_m \text{ è indotta (può non esserlo)}$$

Diamo quindi la definizione di compatibilità:

Definizione 6.4: Compatibilità

Una norma matriciale $|\cdot|_m$ si dice compatibile con una norma vettoriale $|\cdot|_v$ se vale:

$$|Ax|_v \leq |A|_m \cdot |x|_v, \quad \forall A \in \mathbb{C}^{n\times n}, \quad \forall x \in \mathbb{C}^n$$

Osserviamo che le norme matriciali indotto sono ovviamente compatibili con le norme vettoriali che le inducono. Infatti posto $x \neq 0$:

$$\frac{|Ax|_v}{|x|_v} \leq |A|_m \implies |Ax|_v \leq |A|_m \cdot |x|_v$$

□

6.1.3 Norme e autovalori

Le diseguaglianze precedenti e le proprietà delle norme ci dicono che gli autovalori non possono essere ovunque sul piano complesso, ma in modulo devono essere più piccoli di $|A|_m$ se $|\cdot|_m$ è compatibile (a maggior ragione indotta) con una norma vettoriale. Infatti, con (x, λ) autocoppia per A, cioè:

$$Ax = \lambda x, \quad \lambda \in \mathbb{C}, \quad x \neq 0$$

si ha:

$$|\lambda x| = |\lambda| \cdot |x| = |Ax|_v \leq |A|_m \cdot |x|_v \implies |\lambda| \leq |A|_m$$

□

Questo significa che se si prende il "disco" centrato sull'origine con raggio uguale alla norma della matrice A, allora tutti gli autovalori di A devono trovarsi all'interno di tale disco, cioè:

$$\lambda \text{ autovalore} \Leftrightarrow \lambda \in \{z \in \mathbb{C} : |z| \leq |A|_m\}$$

In realtà l'assunzione $|\cdot|_m$ compatibile può essere rimossa, ovvero vale il seguente teorema:

Teorema 6.2: di Hirsch

Data $A \in \mathbb{C}^{n \times n}$ allora $\forall |\cdot|$ norma matriciale vale:

$$\rho(A) = \max\{|\lambda| \text{ autovalore}\} \leq |A|$$

Questo si dimostra prendendo (λ, x) autocoppia per A e considerando la matrice:

$$B = \begin{pmatrix} x & | & 0 \end{pmatrix}$$

per cui vale:

$$A \cdot B = \begin{pmatrix} Ax & A0 & \dots & A0 \end{pmatrix} = \begin{pmatrix} \lambda x & 0 & \dots & 0 \end{pmatrix} = \lambda \begin{pmatrix} x & 0 & \dots & 0 \end{pmatrix} = \lambda B$$

allora vale:

$$|\lambda||B| = |\lambda B| = |A \cdot B| \leq |A| \cdot |B|$$

dividendo per B ($B \neq 0$ da $x \neq 0$ autovettore):

$$\frac{|\lambda||B|}{|B|} \leq \frac{|A| \cdot |B|}{|B|} \implies |\lambda| \leq |A|$$

e quindi $\rho(A) \leq |A|$. □

Questa misura potrebbe non essere molto utile in quanto la norma $|A|$ potrebbe essere aribtrariamente più grande del raggio spettrale $\rho(A)$

6.1.4 Teoremi di Gershgorin

Esistono alcuni teoremi più forti riguardo alle norme degli autovalori, che prendono il nome di **teoremi di Gershgorin**.

Iniziamo dalla definizione di *cerchio* di Gershgorin:

Definizione 6.5: Cerchio di Gershgorin

Data $A \in \mathbb{C}^{n \times n}$ definiamo gli insiemi:

$$\mathcal{F}_i(A) = \{z \in \mathbb{C} : |z - a_{ii}| \leq \rho_i\}, \quad \rho_i = \sum_{j \neq i}^{n} |a_{ij}| \geq 0$$

cerchi di Gershgorin di A associati alla riga i.

Questi sono effettivamente i cerchi di raggio uguale alla somma degli elementi fuori dalla diagonale, con centro sull'asse dei reali il valore presente sulla diagonale, per ogni riga.

Possiamo quindi enunciare i teoremi di Gershgorin:

Teorema 6.3: Primo teorema di Gershgorin

Se $\lambda \in \mathbb{C}$ è autovalore vale che:

$$\lambda \in \bigcup_{i=1}^{n} \mathcal{F}_i(A)$$

cioè gli autovalori stanno nell'unione dei cerchi di Gershgorin. Per dimostrare questo risultato si prende l'autocoppia di A (λ, x) e si indica con k l'indice della componente di massimo modulo in x. Quindi $x \neq 0$ perchè x autovettore, e k è scelto tale che $|x_k| > |x_j|$, $\forall j = 1, ..., n$. Sappiamo quindi che vale $Ax = \lambda x$ dalla definizione di autovalore, e consideriamo la riga k nell'uguaglianza:

$$a_{k1}x_1 + a_{k2}x_2 + ... + a_{kn}x_n = \lambda x_k \Leftrightarrow \sum_{j=1}^{n} a_{kj}x_j = \lambda x_k \Leftrightarrow \sum_{j \neq k}^{n} a_{kj}x_j = \lambda x_k - a_{kk}x_k = (\lambda - a_{kk})x_k$$

$$\implies |\lambda - a_{kk}| \cdot |x_k| = \left| \sum_{j \neq k}^{n} a_{kj}x_j \right| \leq \sum_{j \neq n}^{n} |a_{kj}| \cdot |x_j|$$

Dato $|x_k| > 0$, si divide da entrambi i lati per $|x_k|$:

$$\implies |\lambda - a_{kk}| \leq \sum_{j \neq n}^{n} |a_{kj}| \frac{x_j}{x_k} \leq \sum_{j \neq k}^{n} |a_{kj}| = \rho_k$$

Quindi:

$$|\lambda - a_{kk}| \leq \rho_k \implies \lambda \in \mathcal{F}_k(A)$$

e, dato che l'argomento utilizzato vale per un qualsiasi autovalore, abbiamo la tesi del teorema:

$$\lambda \in \bigcup_{i=1}^{n} \mathcal{F}_i(A)$$

□

Osserviamo che in realtà avremmo dimostrato che $\lambda \in \mathcal{F}_k(A)$ con k indice dell'elemento di massimo modulo nell'autovettore. Tuttavia, nella situazione usuale in cui non conosciamo né λ ne x, sapere k non è plausibile.

Definizione 6.6: Matrice a predominanza diagonale forte

Una matrice A si dice a predominanza diagonale forte se vale:

$$|a_{ii}| > \sum_{j \neq i}^{n} |a_{ij}| = \rho_i, \quad \forall i = 1, ..., n$$

Un **corollario** del teorema 6.3 è che se A è a predominanza diagonale forte, allora A è non singolare ($\det(A) \neq 0$).

Questo si dimostra dal fatto che tutti i cerchi di Gershgorin di A non comprendono l'origine, e quindi:

$$0 \notin \bigcup_{i=1}^{n} \mathcal{F}_i(A)$$

e 0 non può essere autovalore di A.

Teorema 6.4: Secondo teorema di Gershgorin

Se M_1 è unione di s cerchi di Gershgorin ed M_2 è unione di $n - s$ cerchi di Gershgorin e vale $M_1 \cap M_2 = \emptyset$, allora M_1 contiene esattamente s autovalori e M_2 ne contiene esattamente altri $n - s$.

Osserviamo che il primo teorema di Gershgorin (6.3) non ci dice che necessariamente c'è un autovalore per ogni cerchio $\mathcal{F}_i(A)$, cioè può succedere che $\mathcal{F}_i(A)$ non ne contiene nemmeno uno. Il secondo teorema di Gershgorin (6.4) invece ci dice che se abbiamo un cerchio isolato, allora questo contiene esattamente un autovalore.

Un **corollario** del secondo teorema di Gershgorin è che se A ha n cerchi disgiunti, allora A ha n autovalori distinti ed è diagonalizzabile.

6.1.5 Matrici reali e Gershgorin

Ci sono delle note da fare sulle **matrici reali**. Osserviamo infatti che se $A \in \mathbb{R}^{n \times n}$, se λ è autovalore di A allora lo è anche $\overline{\lambda}$. Quindi, se un cerchio di Gershgorin contiene un autovalore complesso, dovrà necessariamente contenere anche il suo coniugio. Ora, visto che un cerchio isolato contiene necessariamente un solo autovalore, sarà che un cerchio isolato in una matrice a entrate reali contiene necessariamente un autovalore reale, e quindi la matrice ha necessariamente un autovalore reale.

6.1.6 Matrici trasposte e Gershgorin

Dato che A e A^T hanno gli stessi autovalori (dallo stesso polinomio caratteristico), si può prendere come regione dove stanno gli autovalori l'intersezione:

$$\left[\bigcup_{i=1}^{n} \mathcal{F}_i(A)\right] \cap \left[\bigcup_{i=1}^{n} \mathcal{F}_i(A^T)\right]$$

dove notiamo bene che:

$$\left[\bigcup_{i=1}^{n} \mathcal{F}_i(A)\right] \cap \left[\bigcup_{i=1}^{n} \mathcal{F}_i(A^T)\right] \neq \bigcup_{i=1}^{n} \left(\mathcal{F}_i(A) \cap \mathcal{F}_i(A^T)\right)$$

infatti solitamente il termine a sinistra è ben più grande. Osserviamo poi che:

$$\mathcal{F}_I(A^T) = \{z \in \mathbb{C} : |z - a_{ii}| \leq \rho_i\}, \quad \rho_i = \sum_{i \neq i}^{n} |a_{ij}| \geq 0$$

cioè semplicemente si prendono le colonne anziché le righe.

6.1.7 Note sul calcolo dei dischi di Gershgorin

Vediamo come si può usare MATLAB per il calcolo e la visualizzazione dei dischi di Gershgorin di una matrice.

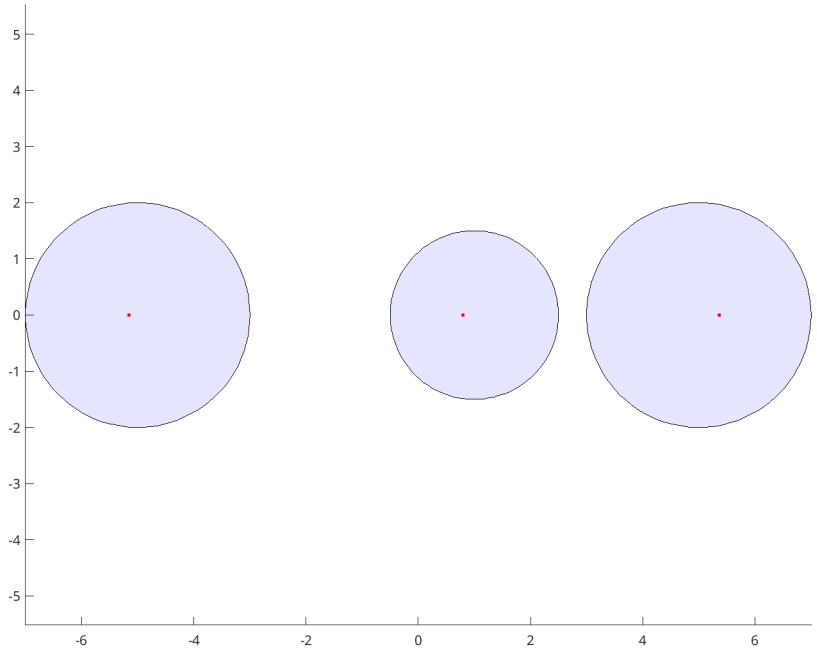
Potremmo pensare di scrivere una funzione per il calcolo e la rappresentazione grafica dei cerchi di Gershgorin di una matrice arbitraria (sia reale che complessa), ad esempio come:

```
function draw_gersh(A, draw_eigen)
    % draws a circle
    function circle(center, radius, col)
        t = linspace(0, 2*pi);
        patch(radius * cos(t) + real(center), ...
              radius * sin(t) + imag(center), col);
    end

    if nargin < 2
        draw_eigen = true;
    end

    n = size(A, 1);

    % graph setup
    close all;
    hold on;
    axis('equal');

    for k = 1:n
        center = A(k, k);
        radius = sum(abs(A(k, [1:k-1, k+1:end])));
        circle(center, radius, 'blue');
    end

    alpha(.1)

    if draw_eigen
        eigenvals = eig(A);
        plot(real(eigenvals), imag(eigenvals), 'red.');
    end

    hold off;
end
```

A questo punto potremo tracciare i cerchi di Gershgorin come:

```
>> A = [5, 1, 1; 1, -5, 1; 1, 0.5, 1];
>> draw_gersh(A)
```

da cui si ottiene il grafico:



Notiamo come, nel caso specifico, a 3 cerchi di Gershgorin distinti corrispondono 3 autovalori distinti, e quindi la matrice A è diagonalizzabile.

7 Lezione del 17-03-25

Avevamo visto i primi 2 teoremi di Gershgorin, cioè:

- **Primo teorema di Gershgorin:** ogni autovalore λ è contenuto in:

$$\lambda \in \bigcup_{i=1}^{n} \mathcal{F}_i(A)$$

  definiti i *cerchi di Gershgorin*:

$$\mathcal{F}_i(A) = \left\{ z \in \mathbb{C}, \quad |z - a_{ii}| < \sum_{j \neq i} a_{ij} \right\}$$

- **Secondo teorema di Gershgorin:** un M_1 è unione di s cerchi e M_2 unione dei restanti $n - s$, allora: $M_1 \cap M_2 = \emptyset \implies M_1$ contiene s autovalori e M_2 i restanti $n - s$.

Vediamo quindi il terzo teorema di Gershgorin. Per fare ciò, introduciamo il concetto di **matrice irriducibile**, e ancor prima di **grafo** associato alla matrice.

7.1 Matrici irriducibili

Diamo la definizione:

Definizione 7.1: Grafo associato ad una matrice

Data una matrice $A \in \mathbb{C}^{n\times n}$ si dice grafo associato ad A, indicato con $G(A) = (V, E)$, il grafo che ha come vertici l'insieme $V = \{1, ..., n\}$ e come archi l'insieme E così definito:

$$(i, j) \in E \Leftrightarrow a_{ij} \neq 0$$

Solitamente si ignora la diagonale (notiamo che in caso di entrate $\neq 0$ risultano semplicemente in anelli aperti e chiusi sulllo stesso nodo).

A noi interesseranno grafi **fortemente connessi**:

Definizione 7.2: Grafo fortemente connesso

Un grafo si dice fortemente connesso se per ogni scelta di vertici $i, j \in 1, ..., n$ esiste un cammino *orientato* sul grafo che parte da i e arriva a j.

A questo punto possiamo definire le matrici **irriducibili**:

Definizione 7.3: Matrice irriducibile

Una matrice $A \in \mathbb{C}^{n\times n}$ si dice irriducibile se il grafo $G(A)$ è fortemente connesso.

Possiamo quindi enunciare il terzo teorema di Gershgorin:

Teorema 7.1: Terzo teorema di Gershgorin

Data una matrice $A \in \mathbb{C}^{n\times n}$ irriducibile si ha che, se λ è un autovalore tale che:

$$\lambda \in \partial \left(\bigcup_{i=1}^{n} \mathcal{F}_i(A) \right)$$

dove con ∂ si indica il bordo, cioè esiste un indice, $\exists k \in \{1, ..., n\}$ tale che: $|\lambda - a_{kk}| = \sum_{j\neq k} a_{kj}$, allora dovrà essere che:

$$\lambda \in \bigcap_{i=1}^{n} \partial \left(\mathcal{F}_i(A) \right)$$

cioè per *tutti* gli indici, $\forall k \in \{1, ..., n\}$ vale che: $|\lambda - a_{kk}| = \sum_{j\neq k} a_{kj}$.

Questo significa che, nel caso di matrici irriducibili, un autovalore che sta sul bordo di un disco di Gershgorin dovrà necessariamente stare sul bordo di *tutti* i dischi di Gershgorin.

Avevamo visto la definizione di matrici a predominanza diagonale forte. Esistono anche matrici a predominanza diagonale *debole* (o semplicemente *diagonali deboli*):

Definizione 7.4: Matrice a predominanza diagonale debole

Una matrice A si dice a predominanza diagonale debole se vale:

$$|a_{ii}| \geq \sum_{j \neq i} |aij|$$

ed $\exists k \in \{1, ..., n\}$ tale per cui:

$$|a_{kk}| > \sum_{j \neq k} |akj|$$

Un **corollario** è che se A è a predominanza diagonale debole ed è irriducibile, allora A non è singolare.

Questo si dimostra direttamente dal fatto che il valore 0 appartiene ai cerchi per cui vale $|a_{ii}| \geq \sum_{j \neq i} |a_{ij}$, ma esiste almeno un cerchio per cui vale $|a_{kk}| > \sum_{j \neq k} |akj|$, cioè non contiene lo 0, e quindi per il terzo teorema di Gershgorin 0 non può essere autovalore (matrice non singolare). □

7.2 Matrici a blocchi

Talvolta è conveniente definire una matrice in termini di *sottomatrici*, o **blocchi**, al posto delle entrate scalari che la compongono, cioè si può scrivere:

$$A \in \mathbb{C}^{m \times n}, \quad A = \begin{pmatrix} A_{11} & ... & A_{1r} \\ ... & & ... \\ A_{s1} & ... & A_{sr} \end{pmatrix}$$

Si può sempre scrivere una matrice a blocchi, purché le sottomatrici abbiano dimensioni compatibili (sia le somme delle dimensioni devono corrispondere alle dimensioni effettive della matrice, sia le dimensioni delle matrici adiacenti devono corrispondere), cioè si deve avere:

$$A_{ij} \in \mathbb{C}^{m_i \times n_j}, \quad \sum_{i=1}^{s} m_i = m, \quad \sum_{j=1}^{r} n_j = n$$

Notiamo che per noi sarà il importante il caso di matrici quadrate poste come sottomatrici quadrate, cioè $m = n$, $s = r$ e $m_i = n_i \; \forall i = 1, ..., r \cdot s$.

Vediamo quindi la definizione di matrici **triangolari a blocchi**:

Definizione 7.5: Matrice triangolare a blocchi

Nelle condizioni di cui sopra (in particolare, matrici quadrate) una matrice si dice triangolare a blocchi se sono $\neq 0$ tutti i blocchi lungo la diagonale, e sopra o sotto la diagonale (dove rispettivamente si parla di matrice triangolare superiore o inferiore), cioè:

$$A \in \mathbb{C}^{m \times n}, \quad A = \begin{pmatrix} A_{11} & ... & A_{1r} \\ 0 & ... & ... \\ 0 & 0 & A_{sr} \end{pmatrix} \text{ (superiore) o } A = \begin{pmatrix} A_{11} & 0 & 0 \\ ... & ... & 0 \\ A_{s1} & ... & A_{sr} \end{pmatrix} \text{ (inferiore)}$$

Vediamo anche le matrici **diagonali a blocchi**:

Definizione 7.6: Matrice diagonale a blocchi

Nelle condizoni della definizione 7.5, una matrice si dice diagonale a blocchi se sono $\neq 0$ i soli blocchi lungo la diagonale, cioè:

$$A \in \mathbb{C}^{m \times n}, \quad A = \begin{pmatrix} A_{11} & 0 & 0 \\ 0 & ... & 0 \\ 0 & 0 & A_{sr} \end{pmatrix} \text{ (diagonale)}$$

7.2.1 Proprietà delle matrici a blocchi

Vediamo alcune proprietà delle matrici a blocchi:

1. Se A è triangolare a blocchi, allora l'insieme degli autovalori di A corrisponde all'unione degli insiemi di autovalori associati ai blocchi sulla diagonale A_{jj};

2. Dalla scorsa proprietà valgono, a cascata, tutte le proprietà che avevamo visto su autovalori e determinanti. Ad esempio il determinante sarà:

   $$\det(A) = \prod_{j=1}^{s} \det(A_{jj})$$

   ricordando che:

   $$\det(A_{jj}) = \prod_{i=1}^{\frac{n}{s}} \lambda_i$$

   con i λ_i autovalori di A_{jj};

3. Se A è diagonale, a blocchi, vale:

   $$A^{-1} = \begin{pmatrix} A_{11}^{-1} & 0 & 0 \\ 0 & ... & 0 \\ 0 & 0 & A_{sr}^{-1} \end{pmatrix}$$

4. Se si hanno 2 matrici a blcchi con lo stesso partizionamento, allora calcolare $A + B$ e $A \cdot B$ si può fare trattando i sottoblocchi come entrate scalari, cioè date:

   $$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

   con A_{11} corrispondente in dimensioni a B_{11} e via dicendo, si ha che:

   $$A + B = \begin{pmatrix} A_{11} + B_{11} & A_{12} + B_{12} \\ A_{21} + B_{21} & A_{22} + B_{22} \end{pmatrix}$$

   riguardo alla somma, e considerazioni simili (che non riportiamo) riguardo al prodotto.

7.3 Matrici riducibili

Vediamo ora una definizione che è effettivamente *duale* a quella di matrice irriducibile: quella di **matrice riducibile**:

Definizione 7.7: Matrice riducibile

Una matrice $A \in \mathbb{C}^{n\times n}$, $n \geq 2$, si dice riducibile se $\exists \Pi$ matrice di permutazione tale che:

$$\Pi A \Pi^T = \begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix}$$

a blocchi, con $A_{11} \in \mathbb{C}^{k\times k}$, $A_{22} \in \mathbb{C}^{(n-k)\times(n-k)}$, $k \in \{1, ..., n\}$.

Osserviamo che ci possono essere più di una matrice di permutazione Π che portano A in una forma triangolare a blocchi, con diversi valori di k.

Osserviamo poi che la forma triangolare superiore non è imperativa, cioè se $\exists \Pi_1$ tale che:

$$\Pi_1 A \Pi_1^T = \begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix}$$

triangolare superiore, allora vale anche che $\exists \Pi_2$ tale che:

$$\Pi_2 A \Pi_2^T = \begin{pmatrix} A_{11} & 0 \\ A_{21} & A_{22} \end{pmatrix}$$

triangolare inferiore. Vediamo anzi che in verità queste due forme sono equivalenti.

7.3.1 Sistemi lineari con matrici riducibili

Supponiamo di voler risolvere $Ax = b$ con $A \in \mathbb{C}^n \times n$, con A riducibile, e conoscendo T. In questo caso conviene procedere sfruttando:

$$Ax = b \;\Leftrightarrow\; \Pi A x = \Pi b \;\Leftrightarrow\; \Pi A \Pi^T \Pi x = \Pi b \;\Leftrightarrow\; By = c$$

dove si è detto $B = \Pi A \Pi^T$, $c = \Pi b$, e $y = \Pi x$. $By = c$ sarà in forma:

$$\begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix}$$

con $c_1, y_1 \in \mathbb{C}^k$ e $c_2, y_2 \in \mathbb{C}^{n-k}$.

Una volta risolto $By = C$, si ritrova x mediante la relazione:

$$\Pi x = y \;\rightarrow\; x = \Pi^T y$$

Vediamo come risolvere effettivamente questo sistema. Scrivendo il sistema 2×2 a blocchi per esteso si ottiene:

$$\begin{cases} A_{11} y_1 + A_{12} y_2 = C_1 \\ A_{22} y_2 = c_2 \end{cases}$$

L'ultima equazione rappresenta un sistema $(n-k) \times (n-k)$, che risolviamo per trovare y_2 (notiamo non c'è dipendenza da y_1, dalla definizione come triangolare superiore a blocchi della matrice B). A questo punto si ottiene un altro sistema lineare:

$$A_{11} y_1 = c_1 - A_{12} y_2$$

con y_2 stavolta noto.

Il *guadagno* di questo metodo è che risolvere un sistema $n \times n$ costa generalmente $O(n^3)$. Effettuare il *preprocessing* reso disponibile dalla riducibilità della matrice ci permette di rendere la complessità uguale a:

$$O(k^3 + (n-k)^3 + k(n-k)) = O(k^3 + (n-k)^3)$$

dove $k(n-k)$ alla prima equazione rappresenta il costo della moltiplicazione matrice-vettore $A_{11}y_2$ nella seconda equazione di risoluzione del sistema a blocchi.

Se si hanno molti 0 da sfruttare, e quindi si può scegliere $k = \frac{n}{2}$, si ottiene un costo:

$$O\left(\left(\frac{n}{2}\right)^3 + \left(n - \frac{n}{2}\right)^3\right) = O\left(2\left(\frac{n}{2}\right)^3\right) = O\left(\frac{n^3}{4}\right)$$

cioè si guadagna di un fattore di 4 in termini di tempo di esecuzione.

Osserviamo inoltre che se i blocchi A_{11}, A_{22} sono a loro volta matrici riducibili, il processo si può applicare **ricorsivamente** sulle matrici della diagonale, in modo da ridurre ulteriormente la complessità del sistema (a patto di dover effettuare più passi di preprocessing).

8 Lezione del 21-03-25

8.1 Valutazione della riducibilità

Riprendiamo il discorso delle matrici riducibili, soffermandoci sul come capire quando una matrice è riducibile, e come ricavare, in caso affermativo, la marice di permutazione Π corrispondente.

Vale il seguente risultato (abbastanza banale guardando a quanto detto riguardo alle matrici irriducibili):

Teorema 8.1: Caratterizzazione di matrice riducibile

Una matrice A è riducibile quando non è irriducibile, cioè quando il suo grafo associato $G(A)$ non è fortemente connesso.

La dimostrazione avviene osservando innanzitutto questo **lemma**: se esiste una certa permutazione π degli elementi in riga, si può ricavare una matrice di permutazione Π e definire la matrice:

$$A' = \Pi A \Pi^T$$

Avremo allora che il grafo associato ad A' sarà lo stesso associato a $\Pi A \Pi^T$, solamente cambiando i nomi dei vertici, cioè:

$$G(A) \text{ fortemente connesso } \Leftrightarrow G(A') \text{ fortemente connesso}$$

La **dimostrazione** vera e propria dovrà quindi affermaredovrà quindi affermare che se una matrice è rducibile, allora il suo grafo non è fortemente connesso, cioè:

$$A \text{ riducibile } \Leftrightarrow G(A) \text{ non fortemente connesso}$$

$\Rightarrow$) Abbiamo che se A è riducibile allora $\exists \Pi$:

$$B = \Pi A \Pi^T = \begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix}$$

con $A_{11} \in \mathbb{C}^{k \times k}$ e $A_{22} \in \mathbb{C}^{(n-k)\times(n-k)}$. Che $G(B)$ non è fortemente connesso è chiaro dal blocco di 0 in basso a sinistra: significherà che non ci sono archi che collegano il blocco A_{22} al blocco A_{11}, cioè non esistono archi che vanno da $\{k+1, ..., n\}$ a $\{1, ..., n\}$.

$\Leftarrow$) Se $G(A)$ non è fortemente connesso allora $\exists (j, h)$ per cui da j non si raggiunge h. Dividamo allra $\{1, ..., n\}$ in due sottoinsiemi:

$$\begin{cases} \mathcal{P} = \{\text{vertici raggiungibili da } j\} \\ \mathcal{Q} = \{\text{vertici non raggiungibili da } j\} \end{cases}$$

con $\mathcal{P} \cup \mathcal{Q} = \{1, ..., n\}$ e $\mathcal{P} \cap \mathcal{Q} = \emptyset$. Non ci sarà quindi nessun arco che collega un elemento di $\mathcal{P}$ a un elemento di $\mathcal{Q}$. Se si considera una permutazione Π che manda in testa tutti gli elementi di $\mathcal{Q}$, si ritrova esattamente la forma della definizione 7.7, cioè quella di una matrice riducibile, notando $k = |\mathcal{Q}|$ e $n - k = \mathcal{P}$.

□

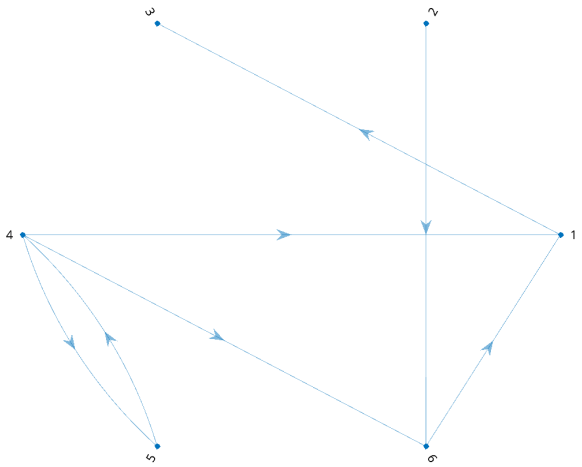
Prendiamo ad esempio la matrice:

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 3 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 3 & 0 & 0 & 2 & -1 & -2 \\ 0 & 0 & 0 & -1 & 2 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Disegnamo il grafo $G(A)$ associato alla matrice A. In MATLAB, questo si potrà fare come:

```
>> A = [ 1 0 1 0 0 0; ... 1 0 0 0 0 1 ]
>> A(eye(size(A)) == 1) = 0 % rimuovi gli elementi sulla diagonale
>> G = digraph(A)
>> p = plot(G)
>> layout(p, "circle") % usa il layout circolare
```

che dalla A dell'esempio dà:



Vediamo quindi che le adiacenze fra nodi (pensando alla matrice, le dipendenze riga-colonna) sono:

Nodo	Raggiungibile	Non raggiungibile
1	1 3	2 4 5 6
2	1 2 3 6	4 5
3	3	1 2 4 5 6
4	1 3 4 5 6	2
5	1 3 4 5 6	2
6	1 3 6	2 4 5

Notiamo che il nodo 6 ci fornisce la partizione migliore: 3×3 e 3×3. Decidiamo quindi di prendere:

$$\mathcal{P} = \{1, 3, 6\}, \quad \mathcal{Q} = \{2, 4, 5\}$$

per cui la permutazione che manda $\mathcal{Q}$ in testa è:

$$\Pi = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{matrix} 2 \to 1 \\ 4 \to 2 \\ 5 \to 3 \\ 1 \to 4 \\ 3 \to 5 \\ 6 \to 6 \end{matrix}$$

Applicando $PiA\Pi^T$ si ottiene quindi:

$$\Pi A \Pi^T = \begin{pmatrix} A_{11} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix} & A_{12} = \begin{pmatrix} 0 & 0 & 3 \\ 3 & 0 & -2 \\ 0 & 0 & 0 \end{pmatrix} \\ 0 & A_{21} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \end{pmatrix}$$

che è una forma più agile per la risoluzione della matrice originale A.

Nella sottodirectory `/code/matlab` si rende disponibile uno script `block_decomp.m` per la decomposizione in matrici a blocchi come nell'esempio. Un'esecuzione tipica dello script potrebbe avere l'aspetto:

```
p = block_decomp(A) % ottieni una permutazione
    Node       Reachable       Not reachable
    ----    -------------    -------------

     1      {[       1 3]}    {[  2 4 5 6]}
     2      {[  1 2 3 6]}    {[      4 5]}
     3      {[        3]}    {[1 2 4 5 6]}
     4      {[1 3 4 5 6]}    {[        2]}
     5      {[1 3 4 5 6]}    {[        2]}
     6      {[    1 3 6]}    {[    2 4 5]}

   Choose an index: 6 % chiesto dallo script
>> A(p, p) % permuta A
```

da cui si ottiene la stessa matrice a blocchi riportata sopra.

8.1.1 Riduzione iterata

Ricordiamo di poter iterare ricorsivamente il processo di riduzione, cioè di poter trovare per ogni blocco A_{ii} con $i \in \{1, 2\}$ un Π_i tale che:

$$\Pi_i A_{ii} \Pi_i^T = \begin{pmatrix} A_{11}^{(i)} & A_{12}^{(i)} \\ 0 & A_{22}^{(i)} \end{pmatrix}$$

così che valga:

$$\begin{pmatrix} \Pi_1 & 0 \\ 0 & \Pi_2 \end{pmatrix} \begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix} \begin{pmatrix} \Pi_1^T & 0 \\ 0 & \Pi_2^T \end{pmatrix} = \begin{pmatrix} \Pi_1 A_{11} \Pi_1^t & \Pi_1 A_{12} \Pi_2^T \\ 0 & \Pi_2 A_{22} \Pi_2^t \end{pmatrix}$$

$$= \begin{pmatrix} \begin{pmatrix} A_{11}^{(1)} & A_{12}^{(1)} \\ 0 & A_{22}^{(1)} \end{pmatrix} & * \\ 0 & \begin{pmatrix} A_{11}^{(2)} & A_{12}^{(2)} \\ 0 & A_{22}^{(2)} \end{pmatrix} \end{pmatrix}$$

e via dicendo, dove in $(*)$ comparrà qualcosa che al momento non ci interessa.

8.1.2 Problemi agli autovalori per riduzione

Notiamo che questo procedimento semplifica anche la risoluzione dei **problemi agli autovalori**: infatti iterando abbastanza, il problema si ridurrà a trovare i singoli autovalori di matrici sulla diagonale sempre più piccole, e quindi dal polinomio caratteristico di più facile risoluzione.

8.2 Sistemi lineari

Veniamo quindi alla trattazione dei sistemi lineari, che avevamo definito come forme $Ax = b$ con $A \in \mathbb{C}^{n \times n}, b \in \mathbb{C}^n$.

Studieremo 2 tipi di metodi risolutivi:

- **Metodi diretti:** esatti ma dispendiosi, se eseguiti in aritmetica esatta (cioè senza arrotondamenti) poterebbero in un numero n finito di passaggi alla soluzione esatta. Esempi di metodi diretti sono il **metodo di Cramer** (visto in 4.7.2) e l'**eliminazione di Gauss** (che vedremo fra poco);
- **Metodi iterativi:** meno accurati ma più efficienti computazionalmente, portano ad una successione $\{x_k\}_{k \in \mathbb{N}}$ di approssimazioni tali che $\lim_{k \to +\infty} x_k = x$ soluzione esatta. Notiamo però che, in generale, è impossibile trovare il valore esatto di x in un numero esatto di iterazioni. Per contropartita, risultano spesso molto più efficienti dei metodi diretti (esistono esempi di sistemi addirittura non risolvibili, nella pratica, con metodi diretti).

8.3 Metodi diretti per i sistemi lineari

Iniziamo a trattare i metodi diretti per la risoluzione dei sistemi lineari.

8.3.1 Sistemi triangolari

Diamo la definizione parallela a quella di matrice triangolare:

Definizione 8.1: Sistema triangolare

Si dice sistema triangolare un sistema della forma $Ux = c$ con U matrice triangolare.

Per un **sistema triangolare superiore**, si avrà la forma:

$$\begin{cases} u_{11}x_1 + u_{12}x_2 + ... + u_{1,n-1}x_{n-1} + u_{1n}x_n = c_1 \\ u_{22}x_2 + ... + u_{2,n-1}x_{n-1} + u_{2n}x_n = c_2 \\ \vdots \\ u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n = c_{n-1} \\ u_{nn}x_n = c_n \end{cases}$$

Il **metodo risolutivo** sarà allora la *sostituzione all'indietro*, definita ricorsivamente come:

$$\begin{cases} x_n = \frac{c_n}{u_{nn}} \\ x_i = \frac{c_i - \sum_{j=i+1}^{n} u_{ij}x_j}{u_{ii}} \end{cases}$$

che equivale all'algoritmo, in MATLAB:

```
function x = bck_subst(U, b)
    n = height(U);
    x = zeros(n, 1);

    for i = n:-1:1
        p = b(i);
        for j = (i + 1):n
            p = p - U(i, j) * x(j);
        end
        x(i) = p / U(i, i);
    end
end
```

Riguardo alla complessità, si potra dire che al passo i si eseguono $n - i + n - i + 2 = 2(n - 1) + 2$ passaggi, cioè 2 per la divisione per u_{jj} e la somma fra c_i e il termine accumulato a destra, $n - i$ per i prodotti nella sommatoria e di nuovo $n - i$ per la sommatoria stessa. Sarà allora che:

$$\sum_{i=1}^{n} (2(n - i) + 2) \sim O(n^2)$$

cioè si ha complessità quadratica.

Osserviamo poi che per **sistemi triangolari inferiori** la situazione è uguale, cioè si risolve la prima equazione, si sostituisce il risultato nella seconda, e via dicendo:

$$\begin{cases} x_1 = \frac{c_1}{u_{11}} \\ x_i = \frac{c_i - \sum_{j=1}^{i-1} u_{ij}x_j}{u_{ii}} \end{cases}$$

che equivale all'algoritmo, in MATLAB:

```
function x = fwd_subst(L, b)
    n = height(L);
    x = zeros(n, 1);

    for i = 1:n
        p = b(i);
        for j = 1:(i -1)
            p = p - L(i, j) * x(j);
        end
        x(i) = p / L(i, i);
    end
end
```

Il metodo ottenuto, speculare al quello di sostituzione all'indietro, viene detto *sostituzione in avanti*, di costo identico ($O(n^2)$).

8.3.2 Metodo di eliminazione di Gauss

L'idea del metodo di eliminazione di Gauss è quella di partire da un sistema $Ax = b$, trasformarlo in un sistema equivalente $Ux = c$ (quindi triangolare superiore), ed applicare la sostituzione all'indietro.

Per arrivare alla forma $Ux = c$ si sostituiscono le equazioni del sistema con loro combinazioni lineari scelte in modo da annullare gli elementi inferiori alla diagonale.

L'idea è quella di eliminare, per ogni elemento i-esimo sulla diagonale a partire da quello in alto a destra, gli $n - i$ elementi che stanno al di sotto, cioè:

Algoritmo 1 Eliminazione di Gauss

Input: un sistema lineare qualsiasi $Ax = b$
Output: un sistema lineare triangolare superiore $Ux = c$
for $i = 1$ to n **do**
  for $j = i$ to n **do**
    Calcola il **moltiplicatore** $l_{ji} = \frac{a_{ji}^{(i-1)}}{a_{ii}^{(i-1)}}$
    Aggiungi alla riga j la riga i moltiplicata per l_{ij}
  end for
end for

8.3.3 Implementazione MATLAB del metodo di eliminazione Gauss

Una semplice implementazione in MATLAB del suddetto algoritmo può essere la seguente:

```
function [A, b] = gauss_decomp(A, b)
    n = height(A);

    for i = 1:n % i itera sulle diagonali
        den = A(i, i);

        for j = (i + 1):n % j itera sulle righe
            mul = A(j, i) / den; % moltiplicatore

            A(j, :) = A(j, :) - A(i, :) * mul;
            b(j) = b(j) - b(i) * mul;
        end
    end
```

```
14 end
```

da cui si potrà ottenere una riduzione di Gauss semplicemente come:

```
1  >> [U, c] = gauss_decomp(A, b)
```

Dal punto di vista della complessità, la riduzione in forma triangolare costa $O(\frac{2}{3}n^3)$, e chiaramente domina sul termine $O(n^2)$ della risoluzione di $Ux = c$ con la sostituzione all'indietro.

Facciamo una nota sulla fattibiltà della riduzione di Gauss, definendo:

Definizione 8.2: Moltiplicatori di Gauss

I termini $l_{ji} = \frac{a_{ji}^{(i-1)}}{a_{ii}^{(i-1)}}$ vengono detti moltiplicatori.

Chiaramente, per poter eseguire l'eliminazione di Gauss serve che $a_{jjj-1} \neq 0 \; \forall j = 1, ..., n-1$. Inoltre, i casi $a_{jj}^{j-1} \approx 0$ possono causare problemi di instabilità numerica. Vedremo in seguito metodi per ovviare a questo problema.

8.3.4 Fattorizzazione LU

Il primo passo dell'algoritmo di Gauss si può vedere come equivalente a moltiplicare l'equazione $Ax = b$ a sinistra per una particolare matrice H_1 triangolare inferiore:

$$H_1 = \begin{pmatrix} 1 & ... & ... & 0 \\ -l_{21} & 1 & ... & 0 \\ ... & & & \\ -l_{n1} & ... & ... & 1 \end{pmatrix}$$

con la diagonale a 1 e i moltiplicatori sulla prima colonna, così che AH_1 risulti esattamente quello che volevamo per Gauss, cioè la combinazione lineare di ogni riga j con l'prima riga moltiplicata per il moltiplicatore l_{j1} (con $j > 2$).

Possiamo generalizzare questo processo a una serie di matrici H_i, per ogni elemento sulla diagonale, con la diagonale a 1 e i moltiplicatori corrispondenti a i sulla i-esima colonna:

$$H_i = \begin{pmatrix} 1 & ... & ... & 0 \\ 0 & 1 & ... & 0 \\ ... & -l_{ji} & ... & ... \\ 0 & -l_{ni} & ... & 1 \end{pmatrix}$$

Così, ancora una volta, AH_i risulterà quello che volevamo per Gauss, cioè la combinazione lineare di ogni riga j con l'i-esima riga moltiplicata per il moltiplicatore l_{ji}.

Varrà allora he il metodo di Gauss sarà equivalente a considerare:

$$H_{n-1}H_{n-2} ... H_1 Ax = H_{n-1}H_{n-2} ... H_1 b$$

e potremo quindi dire:

$$H_{n-1}H_{n-2} ... H_1 A = U, \quad L = H_1^{-1} ... H_{n-1}^1$$

da cui:

$$A = LU$$

Semplifichiamo i calcoli notando alcune proprietà delle matrici H_j:

1. Hanno l'inversa facile, in quanto basta invertire i segni:

$$H_i^{-1} = \begin{pmatrix} 1 & \dots & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \dots & l_{ji} & \dots & \dots \\ 0 & l_{ni} & \dots & 1 \end{pmatrix}$$

2. Sono facili da moltiplicare, in quanto si può dire:

$$H_{i_1} \cdot H_{i_2} = \begin{pmatrix} 1 & \dots & \dots & 0 \\ -l_{2i_1} & 1 & \dots & 0 \\ \dots & -l_{ji_2} & \dots & \dots \\ -l_{ni_i} & -l_{ni_2} & \dots & 1 \end{pmatrix}$$

   e:

$$H_{i_1}^{-1} \cdot H_{i_2}^{-1} = \begin{pmatrix} 1 & \dots & \dots & 0 \\ l_{2i_1} & 1 & \dots & 0 \\ \dots & l_{ji_2} & \dots & \dots \\ l_{ni_i} & l_{ni_2} & \dots & 1 \end{pmatrix}$$

   cioè semplicemente si somma sotto la diagonale.

Questo significa che una volta svolta la prima parte dell'eliminazione di Gauss si è gia calcolata la fattorizzazione LU come la matrice dei moltiplicatori:

$$H_{i_1}^{-1} \cdot H_{i_2}^{-1} = \begin{pmatrix} 1 & \dots & \dots & 0 \\ l_{21} & 1 & \dots & 0 \\ \dots & l_{3,2} & \dots & \\ l_{n1} & l_{n2} & \dots & 1 \end{pmatrix}$$

8.3.5 Implementazione MATLAB della riduzione LU

Modifichiamo il codice MATLAB della scorsa sezione per calcolare, oltre all'eliminazione di Gauss (cioè la matrice U), la matrice dei moltiplicatori L:

```
function [A, b, L] = gauss_decomp(A, b)
    n = height(A);

    L = eye(n); % prepara L

    for i = 1:n % i itera sulle diagonali
        den = A(i, i);

        for j = (i + 1):n % j itera sulle righe
            mul = A(j, i) / den; % moltiplicatore
            L(j, i) = mul;

            A(j, :) = A(j, :) - A(i, :) * mul;
            b(j) = b(j) - b(i) * mul;
        end
    end
end
```

A questo punto per calcolare la fattorizzazione LU di una matrice basterà eseguire:

```
>> [U, ~, L] = gauss_decomp(A, b)
>> L * U % idealmente dara' A
```

Si osserva quindi che se già si conoscono L ed U (magari di una matrice che dovremo usare spesso) risolvere $Ax = b$ costa $O(n^2)$, in quanto basta dire:

$$Ax = b \Rightarrow LUx = b \implies x = U^{-1}L^{-1}b$$

dove basta risolvere a cascata:

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

Questi sono due sistemi triangolari, uno **inferiore** (risolvibile per *sostituzione in avanti*), l'altro **superiore** (risolvibile per *sostituzione all'indietro*), da cui l'andamento complessivo $O(n^2)$.

In MATLAB, la soluzione si può quindi avere usando le funzioni definite finora:

```
>> [U, ~, L] = gauss_decomp(A, b)
>> y = fwd_subst(L, b)
>> x = bck_subst(U, y) % x e' la soluzione del sistema
```

Notiamo che questo vale se L ed U sono note (o come nell'esempio vengono calcolate), quindi tolto il prezzo dato dal doverle calcolare (come avevamo notato, conviene per matrici che magari dobbiamo usare spesso).

8.3.6 Metodo di Gauss per variabili matriciali

Se si vuole risolvere $AX = B$ con X e B matrici di vettori colonna di s colonne:

$$X = \begin{pmatrix} x_1 & \dots & x_s \end{pmatrix}, \quad B = \begin{pmatrix} b_1 & \dots & b_s \end{pmatrix}$$

cioè se si vogliono risolvere s sistemi lineari con la stessa matrice A:

$$Ax_1 = b_1, \quad \dots, \quad Ax_s = b_s$$

si può modificare l'algoritmo di Gauss, effettuando le mosse di Gauss sulla matrice aumentata $\left(A \,|\, B\right)$. Alla fine troveremo una matrice $\left(U \,|\, B^{(n-1)}\right)$, dove gli apici $(n-1)$ rappresentano che è la B che si ottiene all'$n-1$-esimo passaggio, che risolverà i sistemi triangolari superiori:

$$Ux_1 = b_1^{(n-1)}, \quad \dots, \quad Ux_s = b_s^{(n-1)}$$

Vediamo un esempio numerico. Partiamo dalle matrici A e B:

$$A = \begin{pmatrix} -2 & 1 & 1 \\ -1 & 3 & 1 \\ 1 & 1 & 2 \end{pmatrix}, \quad B = \begin{pmatrix} -2 & 0 \\ -3 & 0 \\ 2 & 1 \end{pmatrix}$$

Avremo che le due colonne di B rappresentano due vettori "termini noti" diversi, e che quindi la soluzione X di $AX = B$ rappresenterà in verità due vettori soluzione:

$$X = \begin{pmatrix} x_1^{(1)} & x_1^{(2)} \\ x_2^{(1)} & x_2^{(2)} \\ x_3^{(1)} & x_3^{(2)} \end{pmatrix}$$

Costruiamo quindi la matrice AB ed applichiamo il metodo di riduzione di Gauss:

$$AB = \left(\begin{array}{ccc|cc} -2 & 1 & 1 & -2 & 0 \\ -1 & 3 & 1 & -3 & 0 \\ 1 & 1 & 2 & 2 & 1 \end{array}\right) \xrightarrow{\text{Gauss}} \left(\begin{array}{ccc|cc} -2 & 1 & 1 & -2 & 0 \\ 0 & \frac{5}{2} & \frac{1}{2} & -2 & 0 \\ 0 & 0 & \frac{11}{5} & \frac{11}{5} & 1 \end{array}\right) = UB^{(n-1)}$$

Avremo quindi due sistemi dati dalla forma $UX = B^{(n-1)}$:

$$x^{(1)}: \begin{cases} -2x_1^{(1)} + x_2^{(1)} + x_3^{(1)} = -2 \\ \frac{5}{2}x_2^{(1)} + \frac{1}{2}x_3^{(1)} = -2 \\ \frac{11}{5}x_3^{(1)} = \frac{11}{5} \end{cases} \qquad x^{(2)}: \begin{cases} -2x_1^{(2)} + x_2^{(2)} + x_3^{(2)} = 0 \\ \frac{5}{2}x_2^{(2)} + \frac{1}{2}x_3^{(2)} = 0 \\ \frac{11}{5}x_3^{(2)} = 1 \end{cases}$$

da cui le soluzioni:

$$x^{(1)} = \begin{pmatrix} 1 \\ -1 \\ 1 \end{pmatrix} \quad x^{(2)} = \begin{pmatrix} \frac{2}{11} \\ -\frac{1}{11} \\ \frac{5}{11} \end{pmatrix}$$

e quindi la matrice incognita X:

$$X = \begin{pmatrix} 1 & \frac{2}{11} \\ -1 & -\frac{1}{11} \\ 1 & \frac{5}{11} \end{pmatrix}$$

8.3.7 Implementazione MATLAB del metodo di Gauss

Implementiamo un risolutore per sistemi lineari, per comprendere a pieno questo risultato. Vogliamo calcolare X come la soluzione del sistema $AX = B$, e capiamo quindi che quello che cerchiamo sono i vettori colonna $x_1, ..., x_n$ tali per cui:

$$Ax_1 = B_1, \quad ..., \quad Ax_n = b_n$$

con b_i l'i-esimo vettore colonna di B. Sfruttiamo allora l'eliminazione di Gauss per ricavare due matrici, U e $B^{(n-1)}$, tali che $Ux = B^{(n-1)}$, con il vantaggio che U è triangolare superiore e quindi risolvibile per sostituzione all'indietro. Possiamo fare questo in MATLAB come:

```
AI = [A, B]
UB1 = gauss_decomp(AB) % modificando gauss_decomp per un solo argomento
U = UB1(1:n, 1:n)
B1 = UB1(1:n, (n + 1):(n + s)
```

A questo punto potremo trovare le colonne x_i come:

```
>> x1 = bck_subst(U, B1(1:n, 1))
>> x2 = bck_subst(U, B1(1:n, 2))
...
```

e infine concatenare le colonne come:

```
>> X = [x1, ..., xn]
```

che realizzato in un unico script risulta:

```
function X = gauss_solve(A, B)
    n = height(A);
    s = width(B);

    AB = [A, B];

    UB1 = gauss_decomp(AB);
    U = UB1(:, 1:n);
    B1 = UB1(:, (n + 1):(n + s));

    X = zeros(n, s);

```

```
    for i = 1:s
        X(:, i) = bck_subst(U, B1(:, i));
    end
end
```

8.3.8 Metodo di Gauss per il calcolo dell'inversa

Un caso particolare è il **calcolo dell'inversa** con l'algoritmo di **Gauss-Jordan**. Infatti, scegliendo:

$$AX = I$$

con $n = s$, le colonne in X diventeranno l'inversa di A (basti vedere che $AA^{-1} = I$ per definizione).

Facciamo un esempio numerico. Prendiamo la stessa matrice di prima:

$$A = \begin{pmatrix} -2 & 1 & 1 \\ -1 & 3 & 1 \\ 1 & 1 & 2 \end{pmatrix}$$

e calcoliamone l'inversa come soluzione a $AX = I$. Impostiamo quindi la matrice AI e riduciamola:

$$AI = \left(\begin{array}{ccc|ccc} -2 & 1 & 1 & 1 & 0 & 0 \\ -1 & 3 & 1 & 0 & 1 & 0 \\ 1 & 1 & 2 & 0 & 0 & 1 \end{array}\right) \xrightarrow{\text{Gauss}} \left(\begin{array}{ccc|ccc} -2 & 1 & 1 & 1 & 0 & 0 \\ 0 & \frac{5}{2} & \frac{1}{2} & -\frac{1}{2} & 1 & 0 \\ 0 & 0 & \frac{11}{5} & \frac{4}{5} & -\frac{3}{5} & 1 \end{array}\right) = UB^{(n-1)}$$

da cui i 3 sistemi lineari:

$$x^{(1)} : \begin{cases} -2x_1^{(1)} + x_2^{(1)} + x_3^{(1)} = 1 \\ \frac{5}{2}x_2^{(1)} + \frac{1}{2}x_3^{(1)} = -\frac{1}{2} \\ \frac{11}{5}x_3^{(1)} = \frac{4}{5} \end{cases} \qquad x^{(2)} : \begin{cases} -2x_1^{(2)} + x_2^{(2)} + x_3^{(2)} = 0 \\ \frac{5}{2}x_2^{(2)} + \frac{1}{2}x_3^{(2)} = 1 \\ \frac{11}{5}x_3^{(2)} = -\frac{3}{5} \end{cases}$$

$$x^{(3)} : \begin{cases} -2x_1^{(3)} + x_2^{(3)} + x_3^{(3)} = 0 \\ \frac{5}{2}x_2^{(3)} + \frac{1}{2}x_3^{(3)} = 0 \\ \frac{11}{5}x_3^{(3)} = 1 \end{cases}$$

che hanno come soluzione:

$$X = \begin{pmatrix} x^{(1)} & x^{(2)} & x^{(3)} \end{pmatrix} = A^{-1} = \begin{pmatrix} -\frac{5}{11} & \frac{1}{11} & \frac{2}{11} \\ -\frac{3}{11} & \frac{5}{11} & -\frac{1}{11} \\ \frac{4}{11} & -\frac{3}{11} & \frac{5}{11} \end{pmatrix}$$

8.3.9 Implementazione MATLAB del metodo di Gauss per il calcolo dell'inversa

Facciamo un ultimo esempio MATLAB di questo procedimento. Vogliamo calcolare A^{-1} come la soluzione del sistema $AX = I$, e capiamo quindi che quello che cerchiamo sono i vettori colonna $i_1, ..., i_n$ tali per cui:

$$Ai_1 = I_1, \quad ..., \quad Ai_n = I_n$$

con I_i il vettore di zeri con un 1 all'i-esima riga. Anche allora vogliamo sfruttare l'eliminazione di Gauss per ricavare due matrici, U e $B^{(n-1)}$, tali che $Ux = B^{(n-1)}$, con il vantaggio che U è triangolare superiore e quindi risolvibile per sostituzione all'indietro.

Questo è esattamente quello che ci permette di fare la funzione `gauss_solve()` definita prima, fissando il secondo argomento all'identità I (in matlab `eye(n)`):

```
function A_inv = gauss_inv(A)
    n = height(A);
    A_inv = gauss_solve(A, eye(n));
end
```

9 Lezione del 24-03-25

9.1 Pivoting

L'algoritmo di eliminazione di Gauss che abbiamo definito alla scorsa lezione ha un punto di fallimento nel caso uno degli elementi $a_{ii}^{(i-1)}$ sia $= 0$, o comunque ≈ 0, in quanto vorremmo a quel punto calcolare un moltiplicatore $l_{ji} = \frac{a_{ji}}{a_{ii}} \to$ non ben definito.

In tal caso si può modificare l'algoritmo sfruttando una matrice di permutazione che porti un elemento diverso da zero nella stessa posizione di a_{ii}. Vorremo quindi cercare un indice h tal per cui $a_{hi}^{(i-1)}$ sia di modulo massimo nella sua colonna al di sotto di i, cioè:

$$a_{hi}^{(i-1)} \geq \max_{j=i,\dots,n} |a_{ji}^{(i-1)}|$$

e scambiare la riga i con la riga h. Infatti, se $\det(A) \neq 0$, allora necessariamente esiste un $a_{ji}^{(i-1)} \neq 0$ (altrimenti si ha uno 0 obbligato sulla diagonale, che con la matrice triangolare a blocchi dà $\det(A^{(i-1)}) = 0$). □

Un altra conseguenza di questo approccio è che tutti i moltiplicatori l_{ji} diventeranno ≤ 1. L'algoritmo di Gauss con questa modifica si chiama **eliminazione di Gauss con pivoting parziale** (*parziale* perché ne esistono versioni più sofisticate, che non vedremo).

Osserviamo che ogni scambio di righe equivale a moltiplicare a sinistra per una matrice di permutazione Π_i. Quindi il metodo di Gauss con pivoting può essere rappresentato come:

Algoritmo 2 Eliminazione di Gauss con pivoting parziale

Input: un sistema lineare qualsiasi $Ax = b$
Output: un sistema lineare triangolare superiore $Ux = c$
for $i = 1$ to n **do**
  Trova la matrice Π_i che porta l'elemento di modulo massimo in testa
  $A \leftarrow \Pi_i A$
  for $j = i$ to n **do**
    Calcola il **moltiplicatore** $l_{ji} = \frac{a_{ji}^{(i-1)}}{a_{ii}^{(i-1)}}$
    Aggiungi alla riga j la riga i moltiplicata per l_{ij}
  end for
end for

Vediamo un esempio pratico dell'algoritmo prima di procedere all'implementazione MATLAB. Prendiamo la matrice A e il vettore b:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix}, \quad b = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

Nel ridurre la matrice aumentata Ab:

$$\left(\begin{array}{ccc|c} 1 & 2 & 3 & 1 \\ 4 & 5 & 6 & 2 \\ 7 & 8 & 0 & 3 \end{array}\right)$$

ci accorgiamo che alla prima colonna l'entrata di modulo massimo è 7, di indice 3. Si permutano quindi la prima e la terza riga:

$$\xrightarrow{\Pi_1} \left(\begin{array}{ccc|c} 7 & 8 & 0 & 3 \\ 4 & 5 & 6 & 2 \\ 1 & 2 & 3 & 1 \end{array}\right) \xrightarrow{H_1\Pi_1} \left(\begin{array}{ccc|c} 7 & 8 & 0 & 3 \\ 0 & \frac{3}{7} & 6 & \frac{2}{7} \\ 0 & \frac{6}{7} & 3 & \frac{4}{7} \end{array}\right)$$

nuovamente, l'entrata di modulo massimo è all'indice 3. Si permutano quindi la seconda e la terza riga:

$$\xrightarrow{\Pi_2 H_1 \Pi_1} \left(\begin{array}{ccc|c} 7 & 8 & 0 & 3 \\ 0 & \frac{6}{7} & 3 & \frac{4}{7} \\ 0 & \frac{3}{7} & 6 & \frac{2}{7} \end{array}\right) \xrightarrow{H_2\Pi_2 H_1 \Pi_1} \left(\begin{array}{ccc|c} 7 & 8 & 0 & 3 \\ 0 & \frac{6}{7} & 3 & \frac{4}{7} \\ 0 & 0 & \frac{9}{2} & 0 \end{array}\right)$$

9.1.1 Implementazione MATLAB del metodo di eliminazione di Gauss con pivoting

Modifichiamo quindi la funzione gauss_decomp() per introdurre il meccanismo di pivoting appena visto:

```
function [A, b] = gauss_decomp(A, b)
    n = height(A);

    if nargin < 2
        b = zeros(n, 1);
    end

    for i = 1:n % i itera sulle diagonali
        % qui fai il pivot
        max_abs = max(abs(A(i:n, i)));
        h = find(abs(A(i:n, i)) == max_abs, 1);
        h = h + i - 1; % max abs si conta da i in poi

        A([i, h], :) = A([h, i], :); % permuta A
        b([i, h]) = b([h, i]); % permuta b

        den = A(i, i);

        for j = (i + 1):n % j itera sulle righe
            mul = A(j, i) / den; % moltiplicatore
            L(j, i) = mul;

            A(j, :) = A(j, :) - A(i, :) * mul;
            b(j) = b(j) - b(i) * mul;
        end
    end
end
```

9.1.2 Fattorizzazione LU con pivoting

Vediamo come ricavare una fattorizzazione LU dal metodo di Gauss modificato con il pivoting. Si ha quindi che la matrice U si evolve come:

$$A \to \Pi_1 A \to H_1\Pi_1 A \to ... \to H_{n-1}\Pi_{n-1}...H_1\Pi_1 A = U$$

mentre per la L dovremo notare che:

$$LU = \Pi A$$

dove la matrice Π rappresenta tutte le permutazioni fatte sulle righe di A.

Si ha quindi che:

- U è la matrice triangolare superiore trovata alla fine del metodo di Gauss con pivoting;
- L è la matrice dei moltiplicatori, a cui però si devono applicare gli scambi delle righe, come segue: se al passo i applico la matrice Π_i, devo applicare lo stesso cambio nelle prime $i-1$ colonne di L, sotto la diagonale.

Vediamo un esempio numerico spieghi il processo di formazione della matrice L e della matrice di permutazione Π. Presa la stessa matrice dell'esempio precedente:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix}$$

abbiamo che le permutazioni sono, in sequenza:

$$\Pi_1 : \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} \to \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix}, \quad \Pi_2 : \begin{pmatrix} 3 \\ 2 \\ 1 \end{pmatrix} \to \begin{pmatrix} 3 \\ 1 \\ 2 \end{pmatrix}$$

o, in forma matriciale:

$$\Pi_1 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}, \quad \Pi_2 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Calcoliamo quindi L. Π_1 è irrilevante al calcolo di L, quindi la ignoriamo. Vediamo che i primi due moltiplicatori sono $l_{21} = \frac{4}{7}$ e $l_{31} = \frac{1}{7}$, da cui si imposta $L^{(1)}$:

$$L^{(1)} = \begin{pmatrix} 1 & 0 & 0 \\ \frac{4}{7} & 1 & 0 \\ \frac{1}{7} & 0 & 1 \end{pmatrix}$$

Notiamo quindi che dalla Π_2 dobbiamo scambiare gli elementi sotto la diagonale della prima colonna, quindi:

$$L^{(1)} = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{7} & 1 & 0 \\ \frac{4}{7} & 0 & 1 \end{pmatrix}$$

Infine, l'ultimo moltiplicatore $l_{32} = \frac{1}{2}$ non ha ambiguità:

$$L^{(2)} = L = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{7} & 1 & 0 \\ \frac{4}{7} & \frac{1}{2} & 1 \end{pmatrix}$$

Il calcolo di Π deriva invece direttamente studiando la permutazione complessiva data dalle $\Pi_1, ..., \Pi_{n-1}$, in questo caso:

$$\Pi : \begin{pmatrix}1\\2\\3\end{pmatrix} \xrightarrow{\Pi_1} \begin{pmatrix}3\\2\\1\end{pmatrix} \xrightarrow{\Pi_2} \begin{pmatrix}3\\1\\2\end{pmatrix}$$

da cui:

$$\Pi = \begin{pmatrix}0 & 0 & 1\\1 & 0 & 0\\0 & 1 & 0\end{pmatrix}$$

Con brevi calcoli si verifica che:

$$LU = \Pi A$$

9.1.3 Implementazione MATLAB completa del metodo di eliminazione di Gauss con pivoting

Vediamo quindi l'implementazione completa, che calcola anche la matrice L e la matrice Π. Notiamo inoltre l'argomento condizionale b, che viene ignorato se non fornito (abbiamo constatato che spesso è così).

```
function [A, b, L, P] = gauss_decomp(A, b)
    n = height(A);

    if nargin < 2
        b = zeros(n, 1);
    end

    L = eye(n); % prepara L
    P = eye(n); % prepara P

    for i = 1:n % i itera sulle diagonali
        % qui fai il pivot
        max_abs = max(abs(A(i:n, i)));
        h = find(abs(A(i:n, i)) == max_abs, 1);
        h = h + i - 1; % max abs si conta da i in poi

        A([i, h], :) = A([h, i], :); % permuta A
        b([i, h]) = b([h, i]); % permuta b
        if i > 1
            L([i, h], 1:(i - 1)) = L([h, i], 1:(i - 1)); % permuta L
        end
        P([i, h], :) = P([h, i], :); % permuta P

        den = A(i, i);

        for j = (i + 1):n % j itera sulle righe
            mul = A(j, i) / den; % moltiplicatore
            L(j, i) = mul;

            A(j, :) = A(j, :) - A(i, :) * mul;
            b(j) = b(j) - b(i) * mul;
        end
    end
end
```

9.1.4 Determinante con pivoting

Possiamo sfruttare la matrice Π per il calcolo del determinante. Si ha infatti dal teorema di Binet-Cauchy (4.1) che:

$$\det(\Pi)\det(A) = \det(\Pi A) = \det(LU) = \det(L)\det(U)$$

e quindi:

$$\det(A) = \det(\Pi)^{-1}\det(L)\det(U)$$

dove $\det(L) = 1$ (triangolare inferiore con diagonale di 1). Si nota poi che $\det(\Pi)^{-1}$ è $(-1)^s$ è il numero di pivot che effettuiamo. A questo punto $\det(U)$ è semplicemente il prodotto degli elementi sulla diagonale (triangolare superiore), cioè:

$$\prod_{i=1}^{n} a_{ii}^{(i-1)} = \prod_{i=1}^{n} u_{ii}$$

e quindi:

$$\det(A) = (-1)^s \prod_{i=1}^{n} a_{ii}^{(i-1)} = (-1)^s \prod_{i=1}^{n} u_{ii}$$

Si può quindi usare il metodo di Gauss, ancora una volta, per il calcolo del determinante di una matrice, con costo pari al costo dell'eliminazione di Gauss ($O(\frac{2}{3}n^3)$), molto meglio dello sviluppo di Laplace! ($O(n!)$).

9.1.5 Implementazione MATLAB del metodo di Gauss per il determinante

In MATLAB si può calcolare il prodotto delle diagonali come `prod(diag(A))` e il segno di una permutazione come `det(P)` (anche se sicuramente esistono approcci più efficienti). Si può quindi realizzare uno script simile al seguente per il calcolo del determinante sfuttando `gauss_decomp()` con permutazioni:

```
function d = gauss_det(A)
    function s = perm_sign(P)
        s = det(P); % ci sono modi piu' efficienti, vale l'esempio
    end

    [U, ~, ~, P] = gauss_decomp(A);
    d = prod(diag(U)) * perm_sign(P);
end
```

9.2 Condizionamento di un sistema lineare

Date $A \in \mathbb{C}^{n\times n}$ e $b \in \mathbb{C}^n$, supponiamo di voler trovare $Ax = b$ ma a causa di errori nei dati o errori di arrotondamento troviamo (con un qualunque metodo) un vettore perturbato $x + \delta x \in \mathbb{C}^n$ che risolve un sistema lineare di per sé perturbato:

$$(A + \delta A)(x + \delta x) = (b + \delta b)$$

con δA e δB perturbazioni "piccole" della matrice A e del vettore b, quindi $A+\delta A \in \mathbb{C}^{n\times n}$ e $b + \delta b \in \mathbb{C}^n$

La domanda è, se δA e δb sono piccole, posso concludere che anche δx è relativamente piccolo? Si scopre che la risposta a questa domanda è generalmente no. Prendiamo ad esempio il sistema 2×2:

$$\begin{pmatrix} 1 & -1 \\ 1 & 1.000001 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

da cui x esatto è $\begin{pmatrix}1 & 0\end{pmatrix}$. Perturbando b a $\begin{pmatrix}0.999999 & 1\end{pmatrix}$, si ha x perturbato a $\begin{pmatrix}-10^{-6} & -1\end{pmatrix}$, che è chiaramente un cambiamento drastico. Viene da sé che agendo sulla matrice A potremo ottenere effetti anche più drammatici.

9.2.1 Condizionamento in δb

Riprendiamo quindi la definizione di errore relativo:

$$\epsilon = \frac{|\delta x|}{|x|}$$

Pirma di valutare questo errore, diamo la definizione di **numero di condizionamento**:

Definizione 9.1: Numero di condizionamento

Chiamiamo numero di condizionamento di una matrice A, data una certa norma $|\cdot|$, il valore:

$$\mu(A) = |A| \cdot |A^{-1}|$$

Si potrà allora dare, rispetto alla sola deviazione in b (δb), il seguente risultato:

Teorema 9.1: Condizionamento in δb

Se $\delta A = 0$, si ha:

$$\frac{|\delta x|}{|x|} \leq \mu(A) \cdot \frac{|\delta b|}{|b|}$$

Assumendo $\delta A = 0$ con $\det(A) \neq 0$, come nell'esempio precedente, e quindi perturbazioni solo del termine noto, si ha:

$$A(x + \delta x) = (b + \delta b) \iff A\delta x = \delta b$$

visto che $Ax = b$. Passando alle norme, si ha che:

$$|\delta x| \leq |A^{-1}| \cdot |\delta b|$$

e inoltre:

$$|Ax| = |b| \implies |A||x| \geq |b| \implies |x| \geq \frac{|b|}{|A|}$$

quindi:

$$\frac{|\delta x|}{|x|} \leq \frac{|A^{-1}| \cdot |\delta b| \cdot |A|}{|b|} = \frac{|\delta b|}{|b|} \cdot |A| \cdot |A^{-1}|$$

dove ci interessa il valore $|A| \cdot |A^{-1}|$, cioè il numero di condizionamento $\mu(A)$, l'unico che non dipende dall'errore assoluto δb. □

Abbiamo quindi che se $\mu(A) >> 1$, allora l'errore relativo può essere molto più grande dell'errore relativo dei dati e il problema si dice *mal condizionato*.

10 Lezione del 28-03-25

Riprendiamo il discorso sull'errore inerente dei sistemi lineari.

10.0.1 Condizionamento in $\delta \mathbf{A}$

Avevamo preso delle perturbazioni sulle matrici A e b (dovute a vari effetti reali, quali errori di arrotondamento, di misura, ecc...) nella forma:

$$(A + \delta A)(x + \delta x) = (b + \delta b)$$

e volevamo capire quanto può essere grande l'errore relativo $\frac{|\delta x|}{|x|}$, da:

$$\frac{\text{sol. perturbata} - \text{sol. reale}}{\text{sol. reale}} = \frac{|x + \delta x - x|}{|x|} = \frac{|\delta x|}{|x|}$$

Nel caso di $\delta A = 0$, abbiamo visto di poter maggiorare tale quantità come:

$$\frac{|\delta x|}{|x|} \leq \mu(A) \cdot \frac{|\delta b|}{|b|}$$

con $\mu(A) = |A| \cdot |A^{-1}|$ *numero di condizionamento* (definizione 9.1).

Riguardo a $\mu(A)$, si ha che è ≥ 1, cioè chiaramente non si può ridurre l'errore oltre la perfezione, e se $\mu(A) \approx 10^k$, k è il numero di cifre significative che si *perdono* nel risultato $x + \delta x$.

Possiamo quindi reintrodurre il termine δA ed enunciare il seguente teorema:

Teorema 10.1: Condizionamento in $\delta \mathbf{A}$

Se $|\delta A| \cdot |A^{-1}| < 1$ allora si ha:

$$\frac{|\delta x|}{|x|} \leq \frac{\mu(A)}{1 - \mu(A) \cdot \frac{|\delta A|}{|A|}} \cdot \left(\frac{|\delta A|}{|A|} + \frac{|\delta b|}{|b|} \right)$$

dove osserviamo che se $\delta A = 0$ si ottiene la stessa diseguaglianza che abbiamo dimostrato col teorema 9.1.

La dimostrazione del teorema non è immediata. Ci arriviamo in 3 iterazioni successive, restringendo man mano le approssimazioni fatte.

1. Una prima stima si potrebbe avere calcolando direttamente:

   $$(A + \delta A)(x + \delta x) = (b + \delta b) \implies Ax + A\delta x + \delta A x + \delta A \delta x = b + \delta b$$

   $Ax = b$ dalle ipotesi, mentre il termine $\delta A \delta b$, il prodotto di due errori, è trascurabile, quindi:

   $$A\delta x + \delta A x \approx \delta b \implies \delta x \approx A^{-1} (\delta b - \delta A x)$$

   Passando alle norme si ha:

   $$|\delta x| \leq |A^{-1}| (|\delta b| + |\delta A||x|) \tag{1}$$

   da cui dividendo per $|x| \geq \frac{|A|}{|b|}$ e moltiplicando e dividendo il termine a destra per $|A|$, si ha:

   $$\frac{|\delta x|}{|x|} \leq |A^{-1}| \left(\frac{|\delta b|}{|b|} |A| + |\delta A| \frac{|A|}{|A|} \right) = |A^{-1}||A| \left(\frac{|\delta b|}{|b|} + \frac{|\delta A|}{|A|} \right) = \mu(A) \left(\frac{|\delta b|}{|b|} + \frac{|\delta A|}{|A|} \right)$$

   Da cui otteniamo essenzialmente la forma del teorema.

2. Per capire il denominatore del numero di condizionamento, che migliora il maggiorante nei casi dove A è quasi singolare, adottiamo un procedimento diverso che evidenzia l'errore fatto sull'inversa $(A+\delta A)^{-1}$ (nel calcolo precedente, starebbe nel termine $\delta A \delta x$ che abbiamo trascurato). Avremo quindi:

$$(A+\delta A)(x+\delta x) = (b+\delta b) \implies (x+\delta x) = (A+\delta A)^{-1}(b+\delta b)$$

$(A+\delta A)^{-1}$ risulta di difficile approssimazione. Riscriviamolo come:

$$(A+\delta A)^{-1} = (I + A^{-1}\delta A)^{-1}A^{-1}$$

e consideriamo la serie di Von Neumann:

$$(I-M)^{-1} = I + M + M^2 + ...$$

troncando al primo termine, si ha che possiamo riscrivere la prima inversa come:

$$(I + A^{-1}\delta A)^{-1} \approx I - A^{-1}\delta A$$

e quindi:

$$(A+\delta A)^{-1} \approx (I - A^{-1}\delta A)A^{-1} = A^{-1} - A^{-1}\delta A A^{-1} \tag{2}$$

Possiamo convincerci che l'approssimazione è valida sostituendo nella formula per $x+\delta x$ e trovando:

$$x+\delta x = (A^{-1} - A^{-1}\delta A A^{-1})(b+\delta b) = A^{-1}b + A^{-1}\delta b - A^{-1}\delta A A^{-1}b - A^{-1}\delta A A^{-1}\delta b$$

$x = A^{-1}$ dalle ipotesi, mentre il termine $A^{-1}\delta A A^{-1}\delta b$ contiene il prodotto di due errori, ed è trascurabile, quindi:

$$\delta x \approx A^{-1}\delta b - A^{-1}\delta A A^{-1}b = A^{-1}\delta b - A^{-1}\delta A x = A^{-1}(\delta b - \delta A x)$$

che è esattamente la forma della (1) che avevamo al primo metodo.

3. Riprendiamo la forma in (2) dell'inversa:

$$(A+\delta A)^{-1} \approx (I - A^{-1}\delta A)A^{-1}$$

per introdurre il denominatore che vediamo nell'enunciato del teorema. Passando alle norme, si avrà:

$$|I - A^{-1}\delta A||A^{-1}| = \frac{|A^{-1}|}{1 + |A^{-1}\delta A|}$$

Prendendo questa come la norma dell'inversa per il passaggio alle norme della (1), si ha:

$$\frac{|\delta x|}{|x|} \leq \frac{|A^{-1}|}{1+|A^{-1}\delta A|}\left(\frac{|\delta b|}{|b|}|A| + |\delta A|\frac{|A|}{|A|}\right) \leq \frac{|A^{-1}||A|}{1+|A^{-1}||\delta A|}\left(\frac{|\delta b|}{|b|} + \frac{|\delta A|}{|A|}\right)$$

dove l'$|A^{-1}\delta A|$ al denominatore si può scrivere come $\mu(A)\frac{|\delta A|}{|A|}$, in quanto:

$$|A^{-1}||\delta A| = \mu(A)\cdot\frac{|\delta A|}{|A|} = |A||A^{-1|}\cdot\frac{|\delta A|}{|A|}$$

da cui la forma del teorema. □

10.0.2 Stima di μ

In genere è abbastanza costoso calcolare il numero di condizionamento, in quanto bisogna calcolare un inversa e quindi la sua norma. Quello che si può fare è cercarne una stima.

Ad esempio, se A è hermitiana ($A = A^H$) e si considera la norma euclidea $|\cdot|_2$, si ha che:

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

cioè prendiamo il rapporto fra l'autovalore più grande e l'autovalore più piccolo di A, per cui si possono usare i metodi per gli autovalori (che vedremo verso la fine del corso).

10.0.3 Stime a posteriori

Supponiamo di aver calcolato $\tilde{x} \in \mathbb{C}^n$ con un qualunque metodo di approssimazione di x, prese A e b per buone. Per valutare se $\tilde{x}$ è una *buona* approssimazione basta guardare al **vettore residuo**, cioè:

$$r = b - A\tilde{x}$$

Potremmo chiederci se, con $|r|$ piccolo, si hanno anche $|x - \tilde{x}|$ piccoli. Sottraiamo allora $Ax = b$ da r:

$$A(x - \tilde{x}) = r \implies x - \tilde{x} = A^{-1}r$$

e quindi vale la diseguaglianza:

$$|x - \tilde{x}| \leq |A^{-1}||r|$$

Usando:

$$|x| \geq \frac{|b|}{|A|}$$

si otterrà allora che:

$$\frac{|x - \tilde{x}|}{|x|} \leq \frac{|A||A^{-1}||r|}{|b|} = \mu(A) \cdot \frac{|r|}{|b|}$$

Allora, in problemi ben condizionati, avremo che $\frac{|x-\tilde{x}|}{|x|}$ errore relativo e $|r|$ sono comparabili, mentre in problemi con condizionamento $\mu(A) >> 1$ si potrebbe avere:

$$\frac{\frac{|x-\tilde{x}|}{|x|}}{|r|} \approx \mu(A)$$

10.0.4 Tecniche per l'approssimazione delle inverse

Come nota conclusiva della sezione sull'errore nei sistemi lineari, notiamo che ogni volta che c'è bisogno di risolvere una forma del tipo:

$$Ax = b \implies x = A^{-1}b$$

il metodo naive sarebbe quello di calcolare A^{-1} e moltiplicare per b. Vediamo se si può fare di meglio.

Preso $n = 1$, la forma sarà quella di una semplice equazione lineare:

$$ax = b. \implies x = \frac{b}{a}$$

cioè basta fare una sola divisione, mentre l'approccio naive risulterebbe nel:

1. Calcolare il reciproco a^{-1};
2. Moltiplicare il reciproco per $b \implies x = a^{-1} \cdot b$.

che chiaramente risulta in più passaggi, e quindi più approssimazioni intermedie e in definitiva maggiore errore.

Nel caso matriciale il metodo di divisione equivale a fare una divisione matrice-vettore (che in MATLAB si effettua come `A \ b`), di complessità $O(\frac{2}{3}n^3)$. Di contro, il metodo naive (che in MATLAB si effettua come `inv(A) * b`) avrà complessità intorno ad $O(\frac{8}{3}n^3)$, in quanto calcola la fattorizzazione LU e risolve 2n sistemi triangolari.

Inoltre, l'approccio naive è anche meno accurato, in quanto se il numero di condizionamento $\mu(A)$ è alto, l'errore di approssimazione nei passaggi intermedi potrebbe accumularsi molto di più che rispetto all'approccio della semplice divisione matrice-vettore (tanto che la stessa documentazione di MATLAB suggerisce di evitarlo).

10.1 Metodi iterativi per i sistemi lineari

Veniamo quindi a trattare i metodi iterativi per la risoluzione dei sistemi lineari.

L'idea è quella di approssimare la soluzione di un sistema lineare $Ax = b$ generando una successione di vettori $\{x^{(k)}\}_{k\in\mathbb{N}}$ tali che:

$$\lim_{k\to+\infty} x^{(k)} = x$$

La motivazione è chiaramente quella di eludere l'alta complessità di $\sim O(n^3)$ che ha la risoluzione con metodi diretti. Altra motivazione potrebbe essere quella di non conoscere direttamente A, ma solo l'applicazione:

$$v \to A \cdot v$$

(si pensi, anche se rappresenta un caso *non* lineare, ai metodi di discesa a gradiente che ottimizzano funzioni non immediatamente calcolabili o anche solo esprimibili).

Abbiamo quindi che un buon metodo iterativo dovrà:

1. Costare meno di $O(n^3)$ ad ogni passaggio (altrimenti sarebbe inutile rispetto ai metodi diretti), e quindi richiedere solo prodotti di matrici con vettori, o risoluzione di sistemi lineari favorevoli (triangolari, diagonali, ecc...);
2. Data una certa **accuratezza** posta come obiettivo, impiegare un numero ragionevole di iterazioni per raggiungerla.

10.1.1 Metodi di punto fisso

L'idea è quella di partire da $Ax - b = 0$ e di riscrivere come un'equazione equivalente in forma:

$$x = Hx + c, \quad H \in \mathbb{C}^{n\times m},\ c \in \mathbb{C}^n$$

Facciamo alcuni esempi:

1. Si sceglie una matrice $G \in \mathbb{C}^{n\times n}$ invertibile e si considera:

   $$x = x \cdot G(Ax - b) = (I - GA)x + Gb$$

   cioè:

   $$H = I - GA, \quad c = Gb$$

2. Si scompone A come:
$$A = A_1 + A_2$$
per cui:
$$Ax = b \Leftrightarrow (A_1 + A_2)c = b \Leftrightarrow A_1 x = -A_2 x + b \Leftrightarrow x = -A_1^{-1} A_2 x + A_1^{-1} b$$
cioè ancora:
$$H = -A_1^{-1} A_2, \quad c = A_1^{-1} b$$

Una volta trovata un'equazione di punto fisso $x = Hx + c$, quindi, si considera il seguente metodo iterativo:

$$\begin{cases} x^{(0)} \text{ dato} \\ x^{(k+1)} = Hx^{(k)} + c, \quad k = 1, 2, 3, ... \end{cases}$$

partendo da $x^{(0)}$ come dato a priori (sarà la soluzione che vorremo raffinare).

Vediamo che infatti:

Definizione 10.1: Matrice di iterazione

La matrice H di un equazione di punto fisso $x = Hx + c$ viene detta matrice di iterazione.

Per avere una verifica della validità dei metodi di punto fisso, enunciamo il seguente teorema:

Teorema 10.2: Validità dei metodi di punto fisso

Il metodo di punto fisso dato da:

$$\begin{cases} x^{(0)} \text{ dato} \\ x^{(n+1)} = Hx^{(k)} + c, \quad k = 1, 2, 3, ... \end{cases}$$

converge $\forall x^{(0)} \in \mathbb{C}^n$ se e solo se $\rho(H) < 1$.

Questo si dimostra prendendo l'equazione di punto fisso $x = Hx + c$, soddisfatta da x soluzione esatta, per cui:

$$x^{(k+1)} - x = Hx^{(k)} + c - (Hx + c) = H(x^{(k)} - x)$$

Possiamo chiamare $e^{(k)} = x^{(k)} - x$, cioè l'errore al passo k, e quindi l'errore al passo $k + 1$ sarà:
$$e^{(k+1)} = He^{(k)} = H^k e^{(0)}$$

Basterà allora prendere il limite:

$$\lim_{k \to +\infty} e^{(k+1)} = \lim_{k \to +\infty} H^k e^{(0)}$$

Perchè questo tenda a 0, basterà imporre $\rho(H) < 1$, in quanto in tal caso:

$$\lim_{k \to +\infty} H^k e^{(0)} = 0$$

qualsiasi sia l'errore iniziale $e^{(0)}$ (e quindi la scelta di soluzione iniziale $x^{(0)}$) □

In particolare, possiamo ricordare che se $|H| < 1 \implies \rho(H) < 1$, quindi può risultare verificare questa condizione, che è sì più stretta ma anche più facile da verificare. Ricordiamo che non vale assolutamente il contrario, cioè $|H| > 1 \not\implies \rho(H) > 1$.

Di contro, vale anche la condizione $|\det(H)| \geq 1 \implies \rho(H) > 1$, che come prima non si inverte in $|\det(H)| < 1 \not\implies \rho(H) < 1$.

10.1.2 Velocità di convergenza

Guardando alla dimostrazione del teorema 10.2, possiamo osservare che il raggio spettrale $\rho(H)$ ci dà un informazione anche riguardo alla **velocità di convergenza** del metodo di punto fisso.

Infatti, si avrà che vale:

$$\frac{|e^{(k)}|}{|e^{(0)}|} \leq |H^k|$$

Dall'algebra lineare, si ha che quando k è abbastanza grande, $H^k \approx \rho(H)^k$, almeno per norme indotte, in quanto:

$$\lim_{k \to +\infty} \sqrt[k]{|H^k|} = \rho(H)$$

Questo ci dice che se si hanno 2 metodi di punto fisso con matrici di iterazione H_1 e H_2 tali che:

$$\rho(H_1) < \rho(H_2) < 1$$

allora il primo metodo converge più velocemente del secondo, è dall'ulteriore ipotesi < 1, entrambi convergono.

Inoltre, si può stimare il numero di iterazioni k necessarie a raggiungere un certo valore dell'errore:

$$\frac{|e^{(k)}|}{|e^{(0)}|} = \frac{|x^{(k)} - x|}{|e^{(0)}|} \leq \delta$$

infatti, in tal caso basterà imporre:

$$\rho(H)^k \leq \delta \implies k \geq \frac{\log(\delta)}{\log(\rho(H))}$$

10.1.3 Criteri di stop

Molto spesso non è pratico decidere un errore e fare le k iterazioni che dovrebbero portare a tale errore (in quanto non è scontato conoscere $\rho(H)$), ma bensì si preferisce definire un **criterio di stop** per la terminazione dell'algoritmo raggiunte determinate condizioni.

Queste condizioni possono essere:

1. Si può restringere il residuo:
$$\frac{|r^{(k)}|}{|b|} < \delta$$
2. Si può restringere direttamente la variazione di errore:
$$|x^{(k+1)} - x^{(k)}| < \delta$$

10.2 Metodi di Jacobi e Gauss-Seidel

Vediamo i metodi più famosi di questo tipo, detti metodi di **Jacobi** e **Gauss-Seidel**.

L'idea è di scomporre la matrice A come:

$$A = D - E - F =$$

$$\begin{pmatrix} d_1 & 0 & ... & 0 \\ 0 & d_2 & ... & 0 \\ 0 & ... & d_{n-1} & 0 \\ 0 & ... & 0 & d_n \end{pmatrix} - \begin{pmatrix} 0 & 0 & ... & 0 \\ -e_{21} & 0 & ... & 0 \\ -e_{n-1,1} & ... & 0 & 0 \\ -e_{n1} & ... & -e_{n,n-1} & 0 \end{pmatrix} - \begin{pmatrix} 0 & -f_{12} & ... & -f_{1n} \\ 0 & 0 & ... & -f_{2n} \\ 0 & ... & 0 & -f_{n-1,n} \\ 0 & ... & 0 & 0 \end{pmatrix}$$

con D diagonale, E l'opposta della triangolare inferiore a diagonale nulla e F l'opposta della triangolare superiore a diagonale nulla. Varrà quindi:

$$Ax = b \Leftrightarrow (D - E - F)x = b$$

10.2.1 Metodo di Jacobi

Il metodo di Jacobi consiste nel riscrivere quanto trovato come:

$$Dx = (E + F)x + b \implies x = D^{-1}(E + F)x + D^{-1}b$$

cioè trovare un equazione di punto fisso con:

$$H = D^{-1}(E + F), \quad c = D^{-1}b$$

e quindi applicare l'algoritmo:

$$\begin{cases} x^{(0)} \text{ dato} \\ x^{(k+1)} = D^{-1}(E + F)x^{(k)} + D^{-1}b = D^{-1}((E + F)x^{(k)} + b) \end{cases}$$

Osserviamo che ad ogni iterazione si calcola un prodotto matrice-vettore e si risolve un sistema diagonale ($O(n)$), in quanto la D si inverte facilmente (è diagonale):

$$H_J = D^{-1}(E + F) = \begin{pmatrix} \frac{1}{d_1} & 0 & ... & 0 \\ 0 & \frac{1}{d_2} & ... & 0 \\ 0 & ... & \frac{1}{d_{n-1}} & 0 \\ 0 & ... & 0 & \frac{1}{d_n} \end{pmatrix} \begin{pmatrix} 0 & -f_{12} & ... & -f_{1n} \\ -e_{21} & 0 & ... & -f_{2n} \\ -e_{n-1,1} & ... & 0 & -f_{n-1,n} \\ -e_{n1} & ... & -e_{n,n-1} & 0 \end{pmatrix}$$

per cui:

$$(H_J)_{ij} = \begin{cases} -\frac{a_{ij}}{a_{ii}}, & i \neq j \\ 0 & i = j \end{cases}$$

Le matrici che descriveranno il passo di Jacobi saranno allora:

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

osservando che per calcolare $x_i^{(k+1)}$ serviranno *tutte* le componenti di $x^{(k)}$, cioè in codice bisogna mantenere due vettori, in quanto non si può sovrascrivere $x^{(k)}$ prima di aver finito di calcolare $x^{(k+1)}$.

10.2.2 Implementazione MATLAB del metodo di Jacobi

In MATLAB, una possibile implementazione che sfrutta le funzionalità di calcolo matriciale può essere la seguente:

```
function x_old = jacobi(A, b, k, x_old)
    n = height(A);

    if nargin < 4
        % stima iniziale
        x_old = zeros(n, 1);
    end

    % imposta le matrici D E F
    D = diag(A);
    EF = -A + diag(D); % D e' un vettore

    % imposta c
    c = b ./ D;

    for j = 1:k
        x_new = (EF * x_old) ./ D + c;
        x_old = x_new;

        % stampa informazioni
        fprintf("it = %d, res = %.3f\n", j, norm(b - A * x_old));
        disp(x_old);
    end
end
```

10.2.3 Implementazione C++ del metodo di Jacobi

Possiamo sfruttare la forma scalare del metodo di Jacobi per realizzarne un'implementazione direttamente in C++. Definiamo inizialmente alcune funzioni helper in un header, `mat.h`, che useremo anche nell'esempio successivo a questo:

```
#ifndef MAT_H
#define MAT_H

#include <iostream>

inline double** get_matrix(unsigned n) {
  double** A = new double*[n];

  for(int r = 0; r < n; r++) {
    A[r] = new double[n];

    for(int c = 0; c < n; c++) {
      double e;
      std::cout << "Inserisci l'entrata " << r + 1 << ", " << c + 1 << "
    di A" << std::endl;
      std::cin >> e;

      A[r][c] = e;
    }
  }

  return A;
}

```

```
24 inline double* get_vector(unsigned n) {
25   double* b = new double[n];
26 
27   for(int r = 0; r < n; r++) {
28     double e;
29     std::cout << "Inserisci l'entrata " << r + 1 << " di b" << std::endl;
30     std::cin >> e;
31 
32     b[r] = e;
33   }
34 
35   return b;
36 }
37 
38 inline void print_matrix(double** A, unsigned n) {
39   for(int r = 0; r < n; r++) {
40     for(int c = 0; c < n; c++) {
41       std::cout << A[r][c] << "\t";
42     }
43     std::cout << std::endl;
44   }
45 }
46 
47 inline void print_vector(double* b, unsigned n) {
48   for(int r = 0; r < n; r++) {
49     std::cout << b[r] << std::endl;
50   }
51 }
52 
53 inline void init_env(unsigned &n, unsigned &k, double** &A, double* &b) {
54   // prendi dati
55   std::cout << "Inserisci la dimensione n" << std::endl;
56   std::cin >> n;
57   std::cout << std::endl;
58 
59   std::cout << "Inserisci il numero di iterazioni k" << std::endl;
60   std::cin >> k;
61   std::cout << std::endl;
62 
63   std::cout << "Inserzione della matrice A" << std::endl;
64   A = get_matrix(n);
65   std::cout << std::endl;
66 
67   std::cout << "Inserzione del vettore b" << std::endl;
68   b = get_vector(n);
69   std::cout << std::endl;
70 
71   // conferma dati
72   std::cout << "La matrice A e':" << std::endl;
73   print_matrix(A, n);
74   std::cout << std::endl;
75 
76   std::cout << "Il vettore b e':" << std::endl;
77   print_vector(b, n);
78   std::cout << std::endl;
79 }
80 
81 inline void clean_env(unsigned n, double** A, double* b) {
82   // ripulisci
83   for(int r = 0; r < n; r++) {
```

```
    delete[] A[r];
  }
  delete[] A;

  delete[] b;
}

inline void vec_copy(unsigned n, double* v1, double* v2) {
  for(int r = 0; r < n; r++) {
    v2[r] = v1[r];
  }
}

#endif
```

E definiamo quindi l'algoritmo come segue:

```
double* jacobi(unsigned n, unsigned k, double** A, double* b) {
  // inizializza vettori
  double* v1, *v2;
  v1 = new double[n];
  v2 = new double[n];

  for(int r = 0; r < n; r++) {
    v1[r] = 0;
  }

  for(int i = 0; i < k; i++) {
    std::cout << "Iterazione " << i + 1 << std::endl;

    std::cout << "Il vettore vecchio e':" << std::endl;
    print_vector(v1, n);

    for(int r = 0; r < n; r++) {
      v2[r] = b[r];

      for(int c = 0; c < n; c++) {
        if(r == c) continue;

        v2[r] -= A[r][c] * v1[c];
      }

      v2[r] /= A[r][r];
    }

    vec_copy(n, v2, v1);

    std::cout << "Il vettore nuovo e':" << std::endl;
    print_vector(v1, n);
    std::cout << std::endl;
  }

  // ripulisci
  delete v2;

  return v1;
}
```

A questo punto un semplice programma dovrà semplicemente usare le funzioni per il caricamento dell'ambiente di `mat.h` e chiamare la funzione `jacobi()`:

```
int main() {
```

```
  // inizializza ambiente
  unsigned n, k; double** A; double* b;
  init_env(n, k, A, b);

  double* x = jacobi(n, k, A, b);

  std::cout << "Il risultato finale e':" << std::endl;
  print_vector(x, n);
  std::cout << std::endl;

  delete[] x;

  // ripulisci
  clean_env(n, A, b);
  return 0;
}
```

10.2.4 Metodo di Gauss-Seidel

Il metodo di Gauss-Seidel consiste nel riscrivere quanto trovato come:

$$(D-E)x = Fx + b \implies x = (D-E)^{-1}Fx + (D-E)^{-1}b$$

cioè trovare un equazione di punto fisso con:

$$H = (D-E)^{-1}F, \quad c = (D-E)^{-1}b$$

e quindi applicare l'algoritmo:

$$\begin{cases} x^{(0)} \text{ dato} \\ x^{(k+1)} = (D-E)^{-1}Fx^{(k)} + (D-E)^{-1}b \end{cases}$$

Avremo che la matrice di iterazione è:

$$H_{GS} = (D-E)^{-1}F$$

di cui osserviamo che la prima colonna è il vettore di zeri in quanto F ha anch'essa la prima colonna al vettore di zeri.

In ogni caso, notiamo che non è necessario esplicitare H_{GS}, ma ad ogni iterazione basta calcolare il prodotto matrice-vettore:

$$x^{(k)} \to Fx^{(k)}$$

e risolvere col metodo di sostituzione all'indietro il sistema lineare:

$$(D-E)y = Fx^{(k)}$$

Stesso discorso per c, che si trova sempre risolvendo una sola volta, col metodo di sostituzione all'indietro, il sistema lineare:

$$(D-E)c = b$$

10.2.5 Implementazione MATLAB del metodo di Gauss-Seidel

In MATLAB, una possibile implementazione che sfrutta le funzionalità di calcolo matriciale (in particolare la funzione per la sostituzione all'indietro `bck_subst()` che abbiamo già implementato) può essere la seguente:

```
function x_old = seidel(A, b, k, x)
    n = height(A);

    if nargin < 4
        % stima iniziale
        x_old = zeros(n, 1);
    end

    % imposta le matrici D E F
    DE = triu(A);
    F = -A + DE;

    % imposta c
    c = bck_subst(DE, b);

    for j = 1:k
        x_old = bck_subst(DE, F * x_old) + c;
        x_new = x_old;

        % stampa informazioni
        fprintf("it = %d, res = %.3f\n", j, norm(b - A * x_old));
        disp(x_old);
    end
end
```

10.2.6 Ottimizzazione del metodo di Gauss-Seidel

L'implementazione vista adesso del metodo di Gauss-Seidel non è la più efficiente. Vediamo infatti che, preso:

$$x^{(k+1)} = (D-E)^{-1}Fx^{(k)} + (D-E)^{-1}b$$

e moltiplicando per $D^{-1}(D-E)$ si ha:

$$x^{(k+1)} = D^{-1}Ex^{(k+1)} + D^{-1}Fx^{(k)} + D^{-1}b$$

Esplicitare una dipendenza di $x^{(k+1)}$ da sé stessa potrebbe sembrare poco intuitivo, ma è invece conveniente in quanto ogni elemento di $x^{(k+1)}$ dipenderà dai soli elementi precedenti, cioè si avrà:

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

In questo modo si hanno due vantaggi:

1. Non occorre risolvere sistemi triangolari;
2. Per calcolare $x_i^{(k+1)}$ non si ha bisogno di x_h^k per $h < i$, e quindi si possono sovrascrivere le entrate di $x^{(k)}$, e mantenere un unico vettore.

10.2.7 Implementazione C++ del meotod di Gauss-Seidel

Vediamo come si può usare quest'ultima versione (scalare) del metodo di Gauss-Seidel per realizzarne un'implementazione C++, sfruttando la stessa libreria `mat.h` dell'esempio 10.2.3:

```
double* jacobi(unsigned n, unsigned k, double** A, double* b) {
  // inizializza vettori
  double* v1;
  v1 = new double[n];

  for(int r = 0; r < n; r++) {
    v1[r] = 0;
  }

  for(int i = 0; i < k; i++) {
    std::cout << "Iterazione " << i + 1 << std::endl;

    std::cout << "Il vettore vecchio e':" << std::endl;
    print_vector(v1, n);

    for(int r = 0; r < n; r++) {
      v1[r] = b[r];

      for(int c = 0; c < n; c++) {
        if(r == c) continue;

        v1[r] -= A[r][c] * v1[c];
      }

      v1[r] /= A[r][r];
    }

    std::cout << "Il vettore nuovo e':" << std::endl;
    print_vector(v1, n);
    std::cout << std::endl;
  }

  return v1;
}
```

Come vediamo, la proprietà dimostrata comporta modifiche minime al codice (la semplice rimozione del vettore `v2`, con prodotti scalari che vengono fatti tutti su `v1`).

La realizzazione di un programma che esegue questo algoritmo è ancora analoga all'esempio 10.2.3 (implementazioni complete si trovano in `../matlab/code`).

10.2.8 Jacobi/Gauss-Seidel e matrici di permutazione

Un'ultima osservazione è che sia Jacobi che Gauss-Seidel hanno come condizione che $a_{ii} \neq 0, \forall i$, in quanto altrimenti D diagonale non sarebbe invertibile. Se questa condizione non è soddisfatta, si possono fare delle trasformazioni "indolori" sul sistema per ritrovare la diagonale non nulla (applicare matrici di permutazione e applicare il metodo sul sistema permutato, per trovare poi una soluzione che al limite andrà *de*-permutata).

Ad esempio, si può pensare di applicare Jacobi a:

$$\Pi A x = \Pi b$$

per trovare una qualche permutazione Πx di x.