Università degli Studi di Firenze

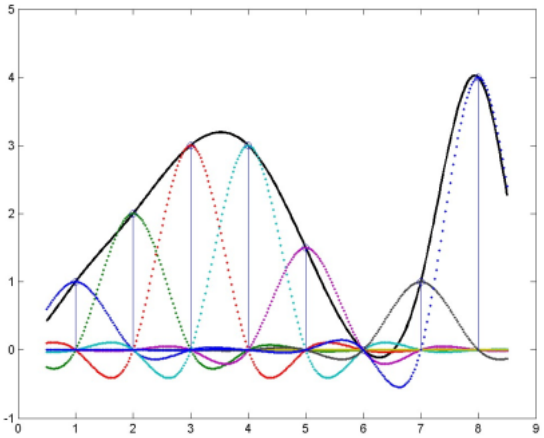
Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Triennale in Informatica (classe L-31)

Anno accademico 2011/2012

Appunti di Calcolo Numerico

Autore:
Tommaso Papini
tommaso.papini.unifi@gmail.com



25 giugno 2013

THE FIRST RULE *of any technology used in a business is that automation applied to an efficient operation will magnify the efficiency. The second is that automation applied to an inefficient operation will magnify the inefficiency*

BILL GATES

Indice

Elenco dei codici

Capitolo 1

Errori ed aritmetica finita

Indice

Quando si utilizza un metodo numerico per risolvere un problema matematico non sempre si ottiene un **risultato esatto**, ma un **risultato approssimato**, che differisce dal risultato esatto.
Sia $x \in \mathbb{R}$ il risultato esatto e $\tilde{x}$ il risultato approssimato corrispondente

Definizione 1.1. *La quantità*

$$\Delta x \equiv \tilde{x} - x$$

*rappresenta l'**errore assoluto** commesso.*

Tuttavia questa quantità non è un buon indice di valutazione dell'errore, in quanto l'errore non viene in alcun modo rapportato all'ordine di grandezza del risultato: un errore assoluto pari a 10^{-3} rappresenta un errore trascurabile se, ad esempio, $x = 10^8$ ma allo stesso tempo è un errore enorme per $x = 10^{-3}$.
Per poter meglio valutare il "peso" che l'errore ha sul risultato approssimato viene quindi introdotto un nuovo concetto

Definizione 1.2. *Se $x \neq 0$, si dice* ***errore relativo*** *la quantità*

$$\varepsilon_x \equiv \frac{\Delta x}{x} = \frac{\tilde{x} - x}{x}$$

Spesso l'errore relativo viene rapportato ad 1: valori di ε_x vicini a 0 significano un errore "piccolo", mentre un errore relativo pari ad 1 implica una perdita totale di informazione.
Le cause che portano al verificarsi di tale errore sul risultato sono principalmente tre:

- errori di discretizzazione, o troncamento;
- errori di convergenza;
- errori di *round-off*.

1.1 Errori di discretizzazione

Si ha un errore di **discretizzazione**, o di **troncamento**, quando il problema dato viene formulato nel continuo e, per poter definire un metodo numerico che lo risolva eseguibile su calcolatore, è necessario sostituire tale problema con uno discreto che lo approssimi.

1.2 Errori di convergenza

Se un metodo numerico utilizza una **funzione d'iterazione** $\Phi(x)$, allora si dice che il metodo è di tipo **iterativo**: esso non fornisce direttamente la soluzione al problema, ma genera una successione di risultati intermedi $\{x_n\}$, definita da

$$x_{n+1} = \Phi(x_n), \quad n = 0, 1, 2, \ldots, \tag{1.1}$$

con x_0 *approssimazione iniziale* fissata, che converge alla soluzione x^*, ovvero

$$\lim_{x \to +\infty} x_n = x^*. \tag{1.2}$$

È ovvio però che un calcolatore non può eseguire infinite iterazioni, quindi sarà necessario definire un opportuno **criterio d'arresto**.

Definizione 1.3. *Se l'iterazione (1.1) viene arrestata a $n = N - 1$, utilizzando come risultato x_N al posto di x^*, la quantità*

$$x_N - x^*$$

viene detta ***errore assoluto di convergenza****.*

1.3 Errori di *round-off*

Si ha un **errore di round-off**, e più precisamente **di rappresentazione**, quando si tenta di rappresentare una quantità numerica, che spesso ha bisogno di una quantità infinita di informazione per essere rappresentata esattamente (come ad esempio i numeri irrazionali), su un calcolatore, che per forza di cose è costretto ad utilizzare un'*aritmetica finita*.
Quindi qualunque numero, in un calcolatore, viene rappresentato mediante una quantità finita di informazione, dando luogo, appunto, agli *errori di rappresentazione*.
In un calcolatore, inoltre, ogni numero viene rappresentato utilizzando una *notazione posizionale* che utilizza le potenza di una base fissata $b \in \mathbb{N}$, con $b \geq 2$.
Quando si parla di *errore di rappresentazione* si distinguono due casi:

- numeri interi;
- numeri reali.

1.3.1 Numeri interi

Un numero intero viene rappresentato, con base $b \in \mathbb{N}$, mediante una stringa del tipo

$$\alpha_0 \alpha_1 \dots \alpha_N,$$

con

$$\alpha_0 \in \{+, -\}, \quad \alpha_i \in \{0, 1, \dots, b-1\}, \quad i = 1, \dots, N.$$

Questa stringa corrisponde al numero

$$n = \begin{cases} \sum_{i=1}^{N} \alpha_i b^{N-i}, & \text{se } \alpha_0 = +, \\ \sum_{i=1}^{N} \alpha_i b^{N-i} - b^N, & \text{se } \alpha_0 = -. \end{cases}$$

Quindi risulta che l'insieme dei numeri correttamente rappresentabili tramite questa rappresentazione è

$$[-b^N, b^N - 1].$$

1.3.2 Numeri reali

Fissata una base $b \in \mathbb{N}$, un numero reale viene rappresentato mediante una stringa del tipo

$$\alpha_0 \alpha_1 \dots \alpha_m \beta_1 \dots \beta_s, \tag{1.3}$$

dove

$$\alpha_0 \in \{+, -\}, \quad \alpha_i, \beta_j \in \{0, 1, \dots, b-1\}, \quad i = 1, \dots, m, \quad j = 1, \dots, s, \tag{1.4}$$

e tale che

$$\alpha_1 \neq 0,$$

cioè tale che il numero sia normalizzato. Questa rappresentazione viene detta **rappresentazione scientifica normalizzata** in base b.
Fissato lo **shift** $\nu \in \mathbb{N}$, la stringa (1.3) rappresenta il numero

$$r = \begin{cases} \rho b^{\eta} & \text{se } \alpha_0 = +, \\ -\rho b^{\eta} & \text{se } \alpha_0 = -. \end{cases} \tag{1.5}$$

dove ρ e η, rispettivamente **mantissa** ed **esponente**, sono le quantità

$$\rho = \pm \sum_{i=1}^{m} \alpha_i b^{1-i}, \qquad \eta = \left(\sum_{j=1}^{s} \beta_j b^{s-j} \right) - \nu.$$

Talvolta viene chiamato esponente anche la sola quantità $e = \sum_{j=1}^{s} \beta_j b^{s-j}$, ovvero la quantità tale che $\eta = e - \nu$.

Definizione 1.4. *L'**insieme dei numeri di macchina**, o numeri floating-point, è definito come*

$$\mathcal{M} = \{0\} \cup \{\textit{numeri della forma (1.3)}\}.$$

Teorema 1.1. $\mathcal{M}$ *ha un numero finito di elementi.*

Teorema 1.2. *Per ogni numero reale della forma (1.5) vale*

$$1 \leq |\rho| < b.$$

Teorema 1.3. *Il più piccolo ed il più grande (in valore assoluto), tra i numeri di macchina diversi da 0, sono rispettivamente dati da:*

$$r_{min} = b^{-\nu}, \tag{1.6}$$

$$r_{max} = (1 - b^{-m}) b^{\varphi}, \quad \varphi = b^s - \nu. \tag{1.7}$$

Solitamente lo *shift* ν viene scelto in modo che $r_{min} \approx r_{max}^{-1}$, ovvero $\nu \approx {}^{b^s}/_2$.
Risulta allora che i numeri di macchina appartengono all'insieme

$$\mathcal{I} = [-r_{max}, -r_{min}] \cup \{0\} \cup [r_{min}, r_{max}], \tag{1.8}$$

che ha un numero infinito di elementi, mentre, per il Teorema 1.1, $\mathcal{M}$ ha un numero finito di elementi.
Quindi si definisce una funzione

$$fl : \mathcal{I} \to \mathcal{M},$$

che associa numeri reali $x \in \mathcal{I}$ a numeri di macchina $fl(x) \in \mathcal{M}$. Quando $x \neq fl(x)$ si commette un *errore di rappresentazione*.
Sia

$$x = (\alpha_1.\alpha_2 \ldots \alpha_m \alpha_{m+1} \ldots) b^{\eta}$$

un elemento positivo di $\mathcal{I}$. È possibile rappresentare tale numero

- con troncamento
$$fl(x) = (\alpha_1.\alpha_2 \dots \alpha_m)b^\eta;$$
- con arrotondamento
$$fl(x) = (\alpha_1.\alpha_2 \dots \alpha_{m-1}\tilde{\alpha}_m)b^\eta,$$
con
$$\tilde{\alpha}_m = \begin{cases} \alpha_m, & \text{se } \alpha_{m+1} < b/2, \\ \alpha_m + 1, & \text{se } \alpha_{m+1} \geq b/2, \end{cases}$$
con eventuali riporti sulle cifre precedenti alla m-esima nel caso in cui $\tilde{\alpha}_m \geq b$.

Infine si impone che $fl(0) = 0$ e che $fl(x) = -fl(-x)$, se $x \in \mathcal{I}$ ed $x < 0$.

Teorema 1.4. *Se $x \in \mathcal{I}$ e $x \neq 0$, allora*
$$fl(x) = x(1 + \varepsilon_x), \quad |\varepsilon_x| \leq u,$$
dove
$$u = \begin{cases} b^{1-m}, & \textit{con troncamento}, \\ \frac{1}{2}b^{1-m}, & \textit{con arrotondamento}. \end{cases}$$

Definizione 1.5. *La quantità u, definita nel Teorema 1.4, è detta* ***precisione di macchina****.*

1.3.3 *Overflow* e *Underflow*

Quando si cerca di rappresentare un numero reale non contenuto nell'insieme $\mathcal{I}$ definito nella (1.8) si incorre in particolari tipi di errori:

- se $|x| > r_{max}$
allora l'errore viene chiamato **overflow** e spesso viene risolto rappresentando i numeri più alti in valore assoluto di r_{max} come una quantità infinita.
- se $0 < |x| < r_{min}$
si incorre in una condizione d'errore denominata di **underflow**. Si può risolvere o con la tecnica di **store 0**, che prevede di rappresentare questi numeri con uno 0, o con la tecnica denominata **gradual underflow**, che consiste nel denormalizzare la mantissa, includendo quindi nell'insieme $\mathcal{M}$ anche i *numeri di macchina denormalizzati*.

Riassumendo l'implementazione della funzione fl si ha che
$$fl(x) = \begin{cases} 0, & \text{se } x = 0, \\ \tilde{x} \equiv x(1 + \varepsilon_x), \quad |\varepsilon_x| \leq u, & \text{se } r_{min} \leq |x| \leq r_{max}, \\ underflow, & \text{se } 0 < |x| < r_{min}, \\ overflow, & \text{se } |x| > r_{max}. \end{cases}$$

1.3.4 Lo standard IEEE 754

Lo standard per la rappresentazione di numeri reali più diffuso è l'*IEEE 754*, che utilizza una reppresentazione in base $b = 2$ ed utilizza una tecnica di arrotondamento definita **round to even**, ovvero rappresentando $fl(x)$ come il numero di macchina "più vicino" ad x (nel caso vi fossero due numeri reali equidistanti viene scelto quello il cui ultimo bit della mantissa è *pari*, ossia 0).
Con questo standard viene guadagnato un bit della mantissa in quanto essa sarà sempre della forma $1.f$, per numeri normalizzati, o della forma $0.f$ per numeri denormalizzati. pertanto viene memorizzata soltanto la *frazione* f.
Lo standard si suddivide in due formati: *singola precisione* e *doppia precisione*.
Di seguito è riportato il numero di bit assegnato a ciascun formato

	Singola precisione	Doppia precisione
segno	1	1
frazione (mantissa)	23 (24)	52 (53)
esponente	8	11
totale	32	64

Mentre per quanto riguarda l'implementazione dei due formati si ha la seguente tabella:

Singola precisione	Doppia precisione	Interpretazione
$0 < e < 255$	$0 < e < 2047$	mantissa normalizzata ($\nu = 127$ in singola precisione, $\nu = 1023$ in doppia)
$e = 0$ e $f \neq 0$		mantissa denormalizzata ($\nu = 126$ in singola precisione, $\nu = 1022$ in doppia)
$e = f = 0$		zero (con segno)
$f = 0$		infinito
$e = 255$	$e = 2047$	(con segno)
$f \neq 0$		NaN
$e = 255$	$e = 2047$	(Not a Number)

1.3.5 Aritmetica finita

Quando si eseguono delle operazioni in aritmetica finita si deve tenere conto dell'insieme di rappresentabilità.
Per quanto riguarda i numeri interi, se entrambi gli operandi ricadono nell'insieme di rappresentabilità, allora l'implementazione non differisce dalla corrispettiva operazione algebrica.
Se invece si sta operando con numeri reali, allora si deve tener conto che l'implementazione opera tra numeri di macchina e restituisce un numero di macchina

come risultato.
Ad esempio l'implementazione della somma algebrica in aritmetica finita sarà:

$$x \oplus y = fl(fl(x) + fl(y)), \quad x, y \in \mathbb{R}. \tag{1.9}$$

Ovviamente, in questo secondo caso, proprietà come associatività o distributività possono non valere più.

1.3.6 Conversione tra tipi diversi

Spesso capita di dover convertire un numero intero in un reale e viceversa. La conversione da intero a reale è sempre possibile, introducendo al più un errore dell'ordine di u, se il numero di bit della mantissa non bastano a rappresentare il numero (a parità di bit nella rappresentazione come intero e come reale si ha che la mantissa ha meno bit a disposizione rispetto alla rappresentazione come intero). La conversione da reale ad intero è invece, generalmente, più problematica, in quanto il numero che si vuole convertire potrebbe non appartenere all'insieme degli interi rappresentabili $\{-b^N, \ldots, b^N - 1\}$, come ad esempio tutti i numeri con virgola.

1.4 Condizionamento di un problema

Sia

$$y = f(x) \tag{1.10}$$

la formalizzazione di un problema matematico che vogliamo risolvere, con $x \in \mathbb{R}$ dati di ingresso, $f : \mathbb{R} \to \mathbb{R}$ funzione che descrive formalmente il problema ed $y \in \mathbb{R}$ soluzione del problema.
Quando si decide di risolvere un problema del genere con un calcolatore viene definito un opportuno metodo numerico che risolva il problema in questione, ma di fatto il problema che ci troveremo a risolvere sarà del tipo

$$\tilde{y} = \tilde{f}(\tilde{x}),$$

dove $\tilde{x}$ rappresenta i dati in ingresso *perturbati* (cioè con errori di *round-off* o ottenuti sperimentalmente), $\tilde{f}$ indica che il metodo numerico è implementato in aritmetica finita (con eventuali errori di *discretizzazione* o *convergenza*) e $\tilde{y}$ denota i dati in uscita affetti dai precedenti errori.
Ciò che è interessante studiare è il **condizionamento del problema**, ovvero come gli errori sui dati in ingresso $\varepsilon_x = \dfrac{\tilde{x} - x}{x}$ si propagano attraverso l'esecuzione del metodo numerico fino a definire l'errore $\varepsilon_y = \dfrac{\tilde{y} - y}{y}$ commesso sulla soluzione finale. Per un'analisi più approfondita e completa dell'errore del metodo numerico utilizzato si dovrebbero considerare anche gli errori introdotti dall'utilizzo di $\tilde{f}$ al posto di f, tuttavia, per semplicità, ci limitiamo a studiare il problema

$$\tilde{y} = f(\tilde{x}).$$

Come accennato precedentemente, se si considerano gli errori relativi sui dati di ingresso e sulla soluzione finale si ha che

$$\tilde{x} = x(1+\varepsilon_x), \quad \tilde{y} = y(1+\varepsilon_y).$$

Ciò che ci interessa determinare è, quindi, la relazione che intercorre tra ε_x e ε_y. Sviluppando $f(\tilde{x})$ in x si ottiene

$$f(\tilde{x}) = f(x(1+\varepsilon_x)) = f(x+x\varepsilon_x) = P_1(x+x\varepsilon_x; x)+O((x\varepsilon_x)^2) = f(x)+f'(x)x\varepsilon_x+O(\varepsilon_x^2),$$

ovvero

$$\tilde{y} = y - y\varepsilon_y = f(x) + f'(x)x\varepsilon_x + O(\varepsilon_x^2).$$

Ricordando la (1.10) e considerando un'*analisi al primo ordine* (ovvero trascurando l'errore $O(\varepsilon_x^2)$), si ha che

$$y + y\varepsilon_y \approx y + f'(x)x\varepsilon_x$$

$$y\varepsilon_y \approx f'(x)x\varepsilon_x$$

$$\varepsilon_y \approx f'(x)\frac{x}{y}\varepsilon_x$$

$$|\varepsilon_y| \approx |f'(x)\frac{x}{y}| \cdot |\varepsilon_x| \equiv k|\varepsilon_x|.$$

Quindi si ha che

$$k \equiv |f'(x)\frac{x}{y}|. \tag{1.11}$$

Definizione 1.6. *Il fattore di amplificazione k, che misura di quanto gli errori sui dati iniziali si possono amplificare sull'errore della soluzione finale, è detto* ***numero di condizionamento del problema*** *(1.10).*

L'ordine di grandezza di k determina quindi il condizionamento del problema:

- se $k \approx 1$, allora l'ordine degli errori finali è uguale a quello degli errori sui dati in ingresso ed il problema si dice **ben condizionato**.
- se $k \gg 1$ significa che nell'eseguire il metodo numerico gli errori finali risultano essere molto più grandi rispetto agli errori sui dati in ingresso. Il problema si dice in questo caso **malcondizionato**.

Se un problema risulta avere numero di condizionamento pari a $k \approx u^{-1}$, allora qualunque risultato sarà privo di significato: infatti l'errore sui dati di ingresso sarà dell'ordine di u, il che vuol dire che risulterà $\varepsilon_y = 1$, che equivale ad una totale perdita di informazione. Se il problema risulta essere malcondizionato l'unica possibilità per ottenere risultati accettabili è quella di riformulare il problema tentando di ottenere un condizionamento migliore; se invece il problema è ben condizionato si deve utilizzare un metodo numerico che risolva il problema mantenendone le buone proprietà di condizionamento (tali metodi numerici sono detti *metodi numericamente stabili*).
Analizziamo il condizionamento delle operazioni algebriche elementari:

- **Somma algebrica**
  Il problema in questione è

$$y = x_1 + x_2, \qquad x_1, x_2 \in \mathbb{R}, \quad x_1 + x_2 \neq 0.$$

  Siano ε_1 e ε_2 gli errori sui dati in ingresso, si ha

$$y(1+\varepsilon_y) = x_1(1+\varepsilon_1) + x_2(1+\varepsilon_2) = x_1 + x_2 + x_1\varepsilon_1 + x_2\varepsilon_2.$$

  Si ottiene quindi

$$y\varepsilon_y = x_1 + x_2 + x_1\varepsilon_1 + x_2\varepsilon_2 - y$$

$$y\varepsilon_y = x_1 + x_2 + x_1\varepsilon_1 + x_2\varepsilon_2 - x_1 - x_2$$

$$|y\varepsilon_y| = |x_1\varepsilon_1 + x_2\varepsilon_2| \leq |x_1\varepsilon_1| + |x_2\varepsilon_2| \leq (|x_1| + |x_2|)\varepsilon_x, \quad \varepsilon_x = max\{|\varepsilon_1|, |\varepsilon_2|\}$$

$$|\varepsilon_y| \leq \frac{|x_1| + |x_2|}{|y|}\varepsilon_x = \frac{|x_1| + |x_2|}{|x_1 + x_2|}\varepsilon_x. \tag{1.12}$$

  Quindi risulta che

$$k \leq \frac{|x_1| + |x_2|}{|x_1 + x_2|}.$$

  Detto questo si possono avere due casi:

  - $x_1 x_2 > 0$: se gli addendi sono concordi, allora $k = 1$ e la somma è ben condizionata;
  - $x_1 \approx -x_2$: se invece gli addendi sono di segno opposto il numero di condizionamento k può essere arbitrariamente grande, perciò in questo caso la somma è malcondizionata. Questo particolare malcondizionamento, in aritmetica finita, prende il nome di *cancellazione numerica*.

- **Moltiplicazione**
  Il problema della moltiplicazione di due numeri reali è formulato come segue:

$$y = x_1 x_2, \qquad x_1, x_2 \in \mathbb{R}, \quad x_1 x_2 \neq 0$$

  Introducendo gli errori relativi

$$y(1+\varepsilon_y) = x_1(1+\varepsilon_1)x_2(1+\varepsilon_2) = x_1 x_2(1 + \varepsilon_1 + \varepsilon_2 + \varepsilon_1\varepsilon_2).$$

  Trascurando il termine quadratico si ha che

$$\begin{aligned} y\varepsilon_y &\approx x_1 x_2(1 + \varepsilon_1 + \varepsilon_2) - y = \\ &= x_1 x_2(1 + \varepsilon_1 + \varepsilon_2) - x_1 x_2 = \\ &= x_1 x_2(\varepsilon_1 + \varepsilon_2) \end{aligned}$$

$$\varepsilon_y \approx \frac{x_1 x_2(\varepsilon_1 + \varepsilon_2)}{x_1 x_2} = \varepsilon_1 + \varepsilon_2$$

$$|\varepsilon_y| \approx |\varepsilon_1 + \varepsilon_2| \leq 2\varepsilon_x, \qquad \varepsilon_x = \{|\varepsilon_1|, |\varepsilon_2|\}.$$

  Quindi il numero di condizionamento risulta essere $k = 2$, pertanto la moltiplicazione è sempre ben condizionata.

- **Divisione**
  Esaminiamo adesso il problema

$$y = \frac{x_1}{x_2}, \qquad x_1, x_2 \in \mathbb{R}, \quad x_1 x_2 \neq 0.$$

  Si ha che

$$y(1+\varepsilon_y) = \frac{x_1(1+\varepsilon_1)}{x_2(1+\varepsilon_2)} = \frac{x_1}{x_2} \cdot \frac{(1+\varepsilon_1)(1-\varepsilon_2)}{(1+\varepsilon_2)(1-\varepsilon_2)} = \frac{x_1}{x_2} \cdot \frac{(1+\varepsilon_1)(1-\varepsilon_2)}{(1-\varepsilon_2^2)} \approx$$
$$\approx \frac{x_1}{x_2}(1+\varepsilon_1)(1-\varepsilon_2) = \frac{x_1}{x_2}(1+\varepsilon_1-\varepsilon_2-\varepsilon_1\varepsilon_2),$$

  dove l'approssimazione è dovuta al fatto che si è trascurato il termine quadratico ε_2^2.
  Trascurando anche il termine quadratico $\varepsilon_1\varepsilon_2$ si ottiene quindi

$$y\varepsilon_y \approx \frac{x_1}{x_2}(1+\varepsilon_1-\varepsilon_2) - y$$
$$y\varepsilon_y \approx \frac{x_1}{x_2}(1+\varepsilon_1-\varepsilon_2) - \frac{x_1}{x_2}$$
$$\varepsilon_y \approx \frac{x_1}{x_2}(\varepsilon_1-\varepsilon_2) \cdot \frac{x_2}{x_1}$$
$$\varepsilon_y \approx \varepsilon_1 - \varepsilon_2$$
$$|\varepsilon_y| \approx |\varepsilon_1 - \varepsilon_2| \leq 2\varepsilon_x, \qquad \varepsilon_x = \{|\varepsilon_1|, |\varepsilon_2|\}.$$

  Anche in questo caso il numero di condizionamento del problema risulta essere $k = 2$, quindi la divisione, come la moltiplicazione, è sempre ben condizionata.

Esercizi

Esercizio 1.1. *Sia $x = \pi \approx 3.1415 = \tilde{x}$. Calcolare il corrispondente errore relativo ε_x. Verificare che il numero di cifre decimali corrette nella rappresentazione approssimata di x mediante $\tilde{x}$ è all'incirca dato da*

$$-\log_{10}|\varepsilon_x|.$$

Soluzione.
Per definizione di errore relativo si ha che

$$\varepsilon_x = \frac{\tilde{x}-x}{x} = \frac{3.1415-\pi}{pi} \approx -2.9493 \times 10^{-5}$$

Risulta allora che $-\log_{10}|\varepsilon_x| \approx 4.5303$; infatti 4 è il numero di cifre decimali esatte di 3.1415 come approssimazione di π in quanto l'errore assoluto

$$\Delta x = \tilde{x} - x = 3.1415 - \pi \approx -9.2654 \times 10^{-5}$$

risulta essere dell'ordine di 10^{-5}, ovvero ha le prime 4 cifre decimali pari a zero.

Riferimenti MATLAB
Codice 1.1 (pagina 26)

Esercizio 1.2. *Dimostrare che, se $f(x)$ è sufficientemente regolare e $h > 0$ è una quantità "piccola", allora:*

$$\frac{f(x_0+h)-f(x_0-h)}{2h} = f'(x_0) + O(h^2),$$

$$\frac{f(x_0+h)-2f(x_0)+f(x_0-h)}{h^2} = f''(x_0) + O(h^2).$$

Soluzione.
Per queste due dimostrazioni si deve sviluppare la funzione $f(x)$ mediante un polinomio di Taylor di grado opportuno. Ricordiamo che lo sviluppo in polinomio di Taylor di grado n della funzione $f(x)$ centrato in x_0 è $P_n(x;x_0) = \sum_{k=0}^{n} \frac{(x-x_0)^k}{k!} f^{(k)}(x_0)$ e che il resto n-esimo vale $R_n(x;x_0) = O(x-x_0)^{n+1}$.
Per la prima uguaglianza si sviluppa $f(x)$ nel polinomio di Taylor al secondo ordine

$$\begin{aligned} f(x) &= P_2(x;x_0) + R_2(x;x_0) = \\ &= f(x_0) + (x-x_0)f'(x_0) + \frac{(x-x_0)^2}{2} f''(x_0) + O((x-x_0)^3). \end{aligned}$$

Calcolando questo sviluppo in $x = (x_0 + h)$ e $x = (x_0 - h)$ si ottiene

$$f(x_0+h) = f(x_0) + hf'(x_0) + \frac{h^2}{2} f''(x_0) + O(h^3),$$

$$f(x_0-h) = f(x_0) - hf'(x_0) + \frac{h^2}{2} f''(x_0) + O(h^3).$$

Quindi sostituendo nel rapporto incrementale di partenza si ha che

$$\begin{aligned} \frac{f(x_0+h)-f(x_0-h)}{2h} &= \frac{f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + O(h^3)}{2h} + \\ &\quad - \frac{f(x_0) - hf'(x_0) + \frac{h^2}{2}f''(x_0) + O(h^3)}{2h} = \\ &= \frac{2hf'(x_0) + o(h^3)}{2h} = f'(x_0) + O(h^2). \end{aligned}$$

Per la seconda uguaglianza è invece necessario utilizzare un'approssimazione con Taylor al terzo ordine

$$\begin{aligned} f(x) &= P_3(x;x_0) + R_3(x;x_0) = \\ &= f(x_0) + (x-x_0)f'(x_0) + \frac{(x-x_0)^2}{2}f''(x_0) + \\ &+ \frac{(x-x_0)^3}{6}f'''(x_0) + O((x-x_0)^4) \end{aligned}$$

e come prima si calcola tale sviluppo in $x = (x_0 + h)$ e $x = (x_0 - h)$

$$f(x_0+h) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \frac{h^3}{6}f'''(x_0) + O(h^4),$$

$$f(x_0-h) = f(x_0) - hf'(x_0) + \frac{h^2}{2}f''(x_0) - \frac{h^3}{6}f'''(x_0) + O(h^4).$$

Quindi sostituendo come nell'uguaglianza precedente si ottiene

$$\begin{aligned} \frac{f(x_0+h) - 2f(x_0) + f(x_0-h)}{h^2} &= \frac{h^2 f''(x_0) + O(h^4)}{h^2} = \\ &= f''(x_0) + O(h^2). \end{aligned}$$

Esercizio 1.3. *Dimostrare che il metodo iterativo (1.1), convergente a x^* (vedi (1.2)), deve verificare la condizione di consistenza*

$$x^* = \Phi(x^*).$$

Ovvero, la soluzione cercata deve essere un punto fisso per la funzione di iterazione che definisce il metodo.

Soluzione.
Essendo il metodo iterativo convergente, per definizione risulta che

$$\lim_{x\to+\infty} x_n = x^*.$$

Supponendo la funzione $\Phi(x_n)$ continua, si ha

$$\lim_{n\to+\infty} \Phi(x_n) = \Phi(\lim_{n\to+\infty} x_n) = \Phi(x^*),$$

inoltre, essendo per definizione $\Phi(x_n) = x_{n+1}$,

$$\lim_{n\to+\infty} \Phi(x_n) = \lim_{x\to+\infty} x_{n+1} = x^*,$$

da cui la tesi, $x^* = \Phi(x^*)$.

Esercizio 1.4. *Il metodo iterativo*

$$x_{n+1} = \frac{x_n x_{n+1} + 2}{x_n + x_{n-1}}, \quad n = 1, 2, \ldots, \quad x_0 = 2, x_1 = 1.5,$$

definisce una successione di approssimazioni convergente a $\sqrt{2}$*. Calcolare a quale valore si n bisogna arrestare l'iterazione, per avere un errore di convergenza* $\approx 10^{-12}$*. Comparare con i risultati nella seguente tabella*

n	x_n
0	2
1	1.5
2	1.416666666666 . . .
3	1.414215686274 . . .
4	1.414213562374 . . .

relativa all'iterazione

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{2}{x_n}\right), \quad n = 0, 1, 2, \ldots, \quad x_0 = 2.$$

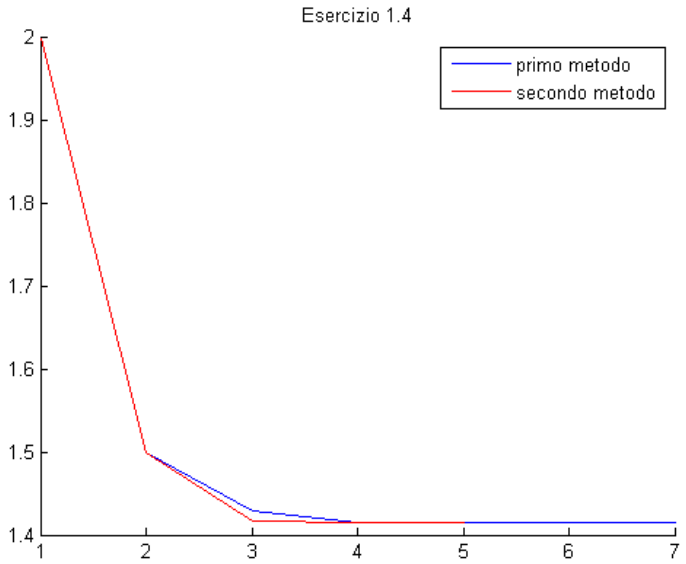
che approssima $\sqrt{2}$*.*

Soluzione.
Risulta che il primo metodo iterativo proposto (quello con i due punti iniziali x_0 ed x_1) approssima $\sqrt{2}$ con errore di convergenza assoluto dell'ordine di 10^{-12} per $i = 6$, come si può vedere dalla tabella seguente

i	x_i	Δx
0	2	$5.85786e-001$
1	1.5	$8.57864e-002$
2	1.428571428571 . . .	$1.43578e-002$
3	1.414634146341 . . .	$4.20583e-004$
4	1.414215686274 . . .	$2.12390e-006$
5	1.414213562688 . . .	$3.15774e-010$
6	1.414213562373 . . .	0

Per $i \geq 6$ l'errore riportato è 0, probabilmente a causa della precisione di macchina che non riesce a rappresentare numeri troppo "piccoli".
Il secondo metodo proposto (quello con il solo punto iniziale $x_0 = 2$) risulta invece

raggiungere un errore assoluto di convergenza per $i = 4$.



Riferimenti MATLAB
Codice 1.2 (pagina 26)

Esercizio 1.5. *Il codice Fortran*

```
    program INTERO
c——variabili intere da 2 byte
    integer*2 numero, i
    numero = 32765
    do i = 1, 10
        write(*,*) i, numero
        numero = numero +1
    end do
    end
```

produce il seguente output:

1	32765
2	32766
3	32767
4	-32768
5	-32767
6	-32766
7	-32765
8	-32764
9	-32763
10	-32762

Spiegarne il motivo.

Soluzione.
Essendo la variabile *numero* di 2 byte significa che il bit più significativo (α_0) rappresenta il segno (0 per un numero positivo ed 1 per un numero negativo), mentre i restanti 15 bit (da α_1 ad $\alpha_1 5$) rappresentano il modulo del numero espresso in complemento a 2.
Quando viene stampata la terza iterazione il numero vale 32767, ovvero

$$\underbrace{0}_{+}\underbrace{111111111111111}_{32767},$$

e ci si aspetta quindi che la quarta iterazione stampi il numero 32768. Tuttavia il numero 32768 avrebbe bisogno, per essere rappresentato in binario, di 16 bit per il modulo e di 1 per il segno, per un totale di 17. Infatti, eseguendo la somma in base due si ottiene

$$\underbrace{1}_{-}\underbrace{000000000000000}_{32768},$$

in quanto il riporto viene propagato fino al bit del segno.
Successivamente il numero viene correttamente incrementato.

Esercizio 1.6. *Dimostrare i Teoremi 1.1 e 1.3.*

Soluzione.

- Teorema 1.1:
  Essendo l'insieme $\mathcal{M}$

  $$\mathcal{M} = \{0\} \cup \{\text{numeri della forma (1.3)}\},$$

  risulta che

  $$|\mathcal{M}| = |\{0\}| + |\{\text{numeri della forma (1.3)}\}|.$$

  Per la (1.4) si vede che:

  - α_0 può assumere 2 valori;
  - α_1 può assumere $b-1$ valori;
  - α_i, β_j possono assumere b valori, per $i = 2, \dots, m$, $j = 1, \dots, s$.

  Quindi la cardinalità di $\mathcal{M}$ è data dalle disposizioni con ripetizione di questi elementi, ovvero

  $$|\mathcal{M}| = 2(b-1)b^{m+s} + 1 < +\infty.$$

- Teorema 1.3:
  Per quanto riguarda il minimo numero di macchina, positivo e diverso da 0, si ha che la mantissa minima vale $\rho_{min} = 1$, mentre l'esponente minimo vale $\eta_{min} = 0 - \nu = -\nu$; quindi, applicando la (1.5)

$$r_{min} = \rho_{min} b^{\eta_{min}} = 1 \cdot b^{-\nu} = b^{-\nu}.$$

  Invece, per quanto riguarda r_{max}, si ha che la mantissa è massima quando tutti i suoi bit valgono $(b-1)$, ovvero

$$\rho_{max} = (b-1)\sum_{i=1}^{m} b^{1-i} = (b-1)\sum_{k=0}^{m-1} b^{-k} = \\ = (b-1)\sum_{k=0}^{m-1}\left(\frac{1}{b}\right)^k,$$

  ricordando che $\left(1-\frac{1}{b}\right)\sum_{k=0}^{m-1}\left(\frac{1}{b}\right)^k = 1^m - \left(\frac{1}{b}\right)^m$, ovvero che $\sum_{k=0}^{m-1}\left(\frac{1}{b}\right)^k = b\frac{1-b^{-m}}{b-1}$, si ha che

$$\rho_{max} = b\left(1-b^{-m}\right).$$

  Per quanto riguarda l'esponente massimo, risulta invece

$$\eta_{max} = \left[(b-1)\sum_{i=1}^{s} b^{s-j}\right] - \nu = \left[(b-1)b^{s-1}\sum_{k=0}^{s-1} b^{-k}\right] - \nu = \\ = [b^s(1-b^{-s})] - \nu = (b^s - 1) - \nu.$$

  Quindi, applicando di nuovo la (1.5), si ottiene

$$r_{max} = b\left(1-b^{-m}\right) b^{b^s-1-\nu} = \left(1-b^{-m}\right) b^{b^s-\nu}.$$

Esercizio 1.7. *Dimostrare il Teorema 1.4 nel caso della rappresentazione con arrotondamento.*

Soluzione.
Si distinguono i due casi in cui si ha arrotondamento per difetto e per eccesso:

- per difetto:
  In questo caso si ha che $\tilde{\alpha}_m = \alpha_m$ in quanto $\alpha_{m+1} < b/2$.

$$|\varepsilon_x| = \frac{|x - fl(x)|}{|x|} = \frac{|(\alpha_1.\alpha_2 \ldots \alpha_m\alpha_{m+1}\cdots - \alpha_1.\alpha_2 \ldots \tilde{\alpha}_m)b^\eta|}{|(\alpha_1.\alpha_2\ldots)b^\eta|} = \\ = \frac{|0.\overbrace{0\ldots0}^{m-1}\alpha_{m+1}\ldots|}{|\alpha_1.\alpha_2\ldots|},$$

quindi, essendo il denominatore sicuramente ≥ 1, il reciproco sarà sicuramente ≤ 1. Se poi si trasla il numeratore di m posizioni otteniamo

$$|\varepsilon_x| \leq |(\alpha_{m+1}.\alpha_{m+2}\dots)b^{-m}| < \frac{b}{2}b^{-m} = \frac{1}{2}b^{1-m} \equiv u.$$

- per eccesso:
  In questo caso invece risulta che $\tilde{\alpha}_m = \alpha_m + 1$, essendo $\alpha_{m+1} \geq b/2$.

$$|\varepsilon_x| = \frac{|x - fl(x)|}{|x|} = \frac{|(\alpha_1.\alpha_2\dots\alpha_m\alpha_{m+1}\cdots - \alpha_1.\alpha_2\dots\tilde{\alpha}_m)b^\eta|}{|(\alpha_1.\alpha_2\dots)b^\eta|} =$$
$$= \frac{|0.\overbrace{0\dots0}^{m-1}\hat{\alpha}_{m+1}\dots|}{|\alpha_1.\alpha_2\dots|},$$

  con $\hat{\alpha}_{m+1} = \tilde{\alpha}_m 0 - \alpha_m\alpha_{m+1} \leq b/2$.
  In questo caso però è possibile che il valore assoluto al numeratore, traslato di m posizioni, sia $> b/2$, come ad esempio nel caso estremo in cui $\hat{\alpha}_{m+1} = b/2$ e tutte le cifre successive valgono $(b-1)$. Quando si verifica questo si nota, tuttavia, che dividendo tale quantità per il valore assoluto al denominatore si ottiene sicuramente una quantità $< b/2$.
  Quando invece il valore assoluto al numeratore (traslato di m posizioni) risulta già essere $\leq b/2$ si procede come nel caso precedente per difetto.
  Quindi in ogni caso si ha che

$$|\varepsilon_x| \leq \frac{b}{2}b^{-m} = \frac{1}{2}b^{1-m} \equiv u.$$

Esercizio 1.8. *Quante cifre binarie sono utilizzate per rappresentare, mediante arrotondamento, la mantissa di un numero, sapendo che la precisione di macchina è $u \approx 4.66 \cdot 10^{-10}$?*

Soluzione.
Applicando il Teorema 1.4 si ha che $u = \frac{1}{2}b^{1-m}$, ovvero che

$$m = 1 - \log_b 2u.$$

Ponendo $b = 2$ e $u = 4.66 \cdot 10^{-10}$ e arrotondando, eventualmente, per eccesso, risulta

$$m = -\log_2 4.66 \cdot 10^{-10} \approx 31.$$

Quindi servono almeno 31 cifre binarie per la mantissa per avere una precisione di macchina non superiore a $4.66 \cdot 10^{-10}$.

Riferimenti MATLAB
Codice 1.3 (pagina 27)

Esercizio 1.9. *Dimostrare che, detta u la precisione di macchina utilizzata,*

$$-\log_{10} u$$

fornisce, approssimativamente, il numero di cifre decimali correttamente rappresentate nella mantissa.

Soluzione.
Studiamo separatamente i casi in cui la rappresentazione avviene tramite troncamento e tramite arrotondamento.

- con troncamento:
  Per il Teorema 1.4, posto $b = 10$, si ha che

$$u = 10^{1-m} \Rightarrow \log_{10} u = 1 - m \Rightarrow m = 1 - \log_{10} u \approx -\log_{10} u.$$

- con arrotondamento:
  Sempre per il Teorema 1.4, con $b = 10$,

$$\begin{aligned} u = \frac{1}{2}10^{1-m} &\Rightarrow \log_{10} 2u = 1 - m \Rightarrow \\ \Rightarrow m = 1 - \log_{10} 2u &= 1 - \log_{10} 2 - \log_{10} u = \\ &= \log_{10} 5 - \log_{10} u \approx -\log_{10} u. \end{aligned}$$

Esercizio 1.10. *Con riferimento allo standard IEEE 754 (vedi Sezione 1.3.4) determinare, relativamente alla doppia precisione:*

1. *il più grande numero di macchina,*
2. *il più piccolo numero di macchina normalizzato positivo,*
3. *il più piccolo numero di macchina denormalizzato positivo,*
4. *la precisione di macchina.*

Confrontare le risposte ai primi due quesiti col risultato fornito dalle function MATLAB `realmax` *e* `realmin`.

Soluzione.

1. Si ha che il più grande numero di macchina in doppia precisione è dato dalla mantissa massima e l'esponente massimo. Per l'Esercizio 1.6 si ha che $\rho = b(1 - b^{-m}) = 2(1 - 2^{-53}) = 2 - 2^{-52}$, mentre $\eta = 2046 - \nu = 2046 - 1023 = 1023$, essendo il valore $e = 2047$ riservato.
   Quindi il massimo numero di macchina risulta essere
   $$r_{max} = (2 - 2^{-52})2^{1023} \approx 1.8 \cdot 10^{308},$$
   infatti la function `realmax` di MATLAB restituisce lo stesso risultato.

2. Per quanto riguarda il più piccolo numero di macchina normalizzato positivo basta applicare la (1.6), con l'accortezza che l'esponente $e = 0$ è riservato e quindi il più piccolo esponente risulta essere $e = 1$:
   $$r_{minN} = 2^{1-\nu} = 2^{1-1023} = 2^{-1022} \approx 2.2 \cdot 10^{-308}.$$
   Utilizzando la function MATLAB `realmin` si perviene allo stesso risultato.

3. Il più piccolo numero di macchina denormalizzato positivo è caratterizzato, per definizione, dall'esponente $e = 0$ e dalla mantissa minima diversa da 0, ovvero la mantissa che ha tutti i bit, tranne il meno significativo, a 0, che vale $\rho = 2^{-52}$.
   Quindi
   $$r_{minD} = 2^{-52}2^{0-\nu} = 2^{-52}2^{-1022} \approx 4.9 \cdot 10^{-324}.$$

4. Dal momento che lo standard *IEEE 754* utilizza la rappresentazione con arrotondamento, per il Teorema 1.4 si ha
   $$u = \frac{1}{2}b^{1-m} = \frac{1}{2}2^{1-53} = 2^{-53} \approx 1.1 \cdot 10^{-16}.$$

Riferimenti MATLAB
Codice 1.4 (pagina 27)

Esercizio 1.11. *Eseguire le seguenti istruzioni* MATLAB*:*

```
x = 0; delta = 0.1;
while x ~= 1, x = x+delta, end
```

Spiegarne il (non) funzionamento.

Soluzione.
Si nota che la rappresentazione in binario di 0.1 è infinita periodica, in particolare vale $0.0\overline{0011}$, in quanto la frazione corrispondente $1/10$ non ha soltanto 2 come fattore primo del denominatore.
Il valore memorizzato nel calcolatore della variabile *delta*, e quindi il valore di x alla prima iterazione, risulta essere $1\underbrace{0011\ldots0011}_{12volte}001$ per la mantissa, che ricordiamo dev'essere normalizzata, e $\eta = -4$, che corrisponde all'esponente $e = \nu + \eta = 1023 - 4 = 1019 = 01111111011_{[2]}$. Questa quantità vale esattamente

$$\begin{aligned} r = 0.&09999999999999999938596421336683465390818471271218498224152779334997546040964300101522745446966427383, \end{aligned}$$

ed è quindi minore di 0.1.
Risulta quindi che alla decima iterazione il valore memorizzato di x è $0.\underbrace{1\ldots1}_{52volte} < 1_{[10]}$ mentre all'undicesima si ha $x = 1.0\underbrace{0011\ldots0011}_{12volte}001 > 1_{[10]}$. Quindi x non assume mai il valore 1 e di conseguenza il ciclo è infinito.
Per far funzionare il codice si può utilizzare il comando MATLAB `eps`, che restituisce lo spazio presente tra i numeri di macchina, ovvero la precisione di macchina, come guardia per la terminazione del ciclo, imponendo che il ciclo termini quando `abs(x-1)<=eps`, che corrisponde all'impostare una tolleranza come criterio d'arresto.

Riferimenti MATLAB
Codice 1.5 (pagina 27)

Esercizio 1.12. *Individuare l'algoritmo più efficace per calcolare, in aritmetica finita, l'espressione* $\sqrt{x^2 + y^2}$.

Soluzione.
L'espressione risulta essere condizionata in quanto i due elevamenti a potenza si possono ricondurre a due moltiplicazioni ($x^2 = x \cdot x$ e $y^2 = y \cdot y$), che sono sempre ben condizionate (con $k = 2$); la somma risulta essere tra addendi concordi (entrambi positivi), quindi anch'essa ben condizionata (con $k = 1$); ed infine l'estrazione della radice quadrata risulta ben condizionata (con $k = 1/2$).
Il problema in cui si può incorrere quando si tenta di valutare quest'espressione in aritmetica finita è che si verifichi un *overflow*: infatti se prendiamo, ad esempio, $x = 10^{200}$ ed $y = 10^{200}$ si ha che il risultato è $\sqrt{2} \cdot 10^{200}$ il quale, se si utilizza lo standard *IEEE 754* con doppia precisione, rientra nell'insieme dei numeri di

macchina, essendo il massimo numero di macchina rappresentabile con questo formato $\approx 1.8 \cdot 10^{308}$ (vedi Esercizio 1.10). Tuttavia l'elevamento a potenza produce il valore 10^{400} che, non rientrando nell'insieme dei numeri di macchina, causa un *overflow*.
Se si suppone, senza perdita di generalità, che $x > y$, una soluzione consiste nel portare fuori dalla radice l'ordine di grandezza di x

$$\sqrt{x^2 + y^2} = \sqrt{x^2(1 + \frac{y^2}{x^2})} = |x|\sqrt{1 + \left(\frac{y}{x}\right)^2}.$$

In questo modo si ha che l'espressione è correttamente calcolata (utilizzando i valori precedenti) $\sqrt{2} \cdot 10^{200}$ anziché *Inf* e l'espressione rimane ben condizionata, in quanto la divisione è un'operazione sempre ben condizionata (con $k = 2$).
Per valori molto grandi di x e molto piccoli di y si può incorrere nel problema opposto, ovvero un *underflow*. Tuttavia questo problema è considerato meno grave in quanto può essere risolto denormalizzando il numero in floating point.

Riferimenti MATLAB
Codice 1.6 (pagina 28)

Esercizio 1.13. *Eseguire le seguenti istruzioni* MATLAB*:*

```
help eps
((eps/2+1)-1)*(2/eps)
(eps/2+(1-1))*(2/eps)
```

Concludere che la somma algebrica non gode, in aritmetica finita, della proprietà associativa.

Soluzione.
Il comando MATLAB `eps`, senza argomenti, restituisce la distanza tra 1 ed il primo numero maggiore di 1 in doppia precisione, ovvero $2^{-52} \approx 2.22 \cdot 10^{-16}$.
Quando, nella prima espressione, il valore di `eps` viene dimezzato e poi sommato ad 1, si ottiene un valore che è a metà tra 1 ed $1 + eps$, ovvero un numero non rappresentabile con la doppia precisione, che viene quindi interpretato come 1. Questo passaggio provoca l'errore, in quanto viene persa una quantità pari ad $eps/2$, restituendo come risultato 0.
La seconda espressione, invece, viene valutata correttamente 1 in aritmetica finita in quanto, dando priorità alla sottrazione $(1 - 1)$ si ha che il valore $eps/2$ viene correttamente moltiplicato per il suo reciproco, restituendo, appunto, 1.
Da questo semplice esempio si evince che, in aritmetica finita, la somma algebrica non gode, in generale, della proprietà associativa.

Esercizio 1.14. *Eseguire e discutere il risultato delle seguenti istruzioni* MATLAB*:*

```
(1e300-1e300)*1e300,     (1e300*1e300)-(1e300*1e300)
```

Soluzione.
Le due espressioni sono algebricamente equivalenti (per la proprietà distributiva del prodotto rispetto alla sottrazione), tuttavia la valutazione della prima restituisce, correttamente, il valore 0, mentre la seconda da come risultato *NaN*, che sta per *Not a Number*.
Questo è dovuto al fatto che mentre nella prima espressione viene subito calcolato il valore 0, che annullerà anche il secondo fattore della moltiplicazione, nella seconda espressione viene eseguita prima la moltiplicazione e poi la sottrazione. Risulta quindi che la valutazione di $10^{300} \cdot 10^{300}$ viene interpretata come *Inf*, ovvero un valore infinito, in quanto il valore effettivo 10^{600} non è rappresentabile in doppia precisione (più precisamente si verificano due overflow). La sottrazione degenera quindi nella forma indeterminata $[\infty - \infty]$, restituendo *NaN* come valore.
Da questo esempio si può dedurre che, in aritmetica finita, non vale la proprietà distributiva del prodotto rispetto alla sottrazione (e ragionevolmente la proprietà distributiva in generale).

Esercizio 1.15. *Eseguire l'analisi dell'errore (relativo), dei due seguenti algoritmi per calcolare la somma di tre numeri (vedi (1.9)):*

$$1) \quad (x \oplus y) \oplus z, \qquad 2) \quad x \oplus (y \oplus z).$$

Soluzione.
L'espressione in aritmetica esatta è equivalente nei due casi ed è data da $R = (x + y) + z = x + (y + z) = x + y + z$.

- Nel primo caso si ha che l'espressione in aritmetica finita è data da

$$\begin{aligned} F_1 &= (x \oplus y) \oplus z = \\ &= fl(fl(fl(fl(x) + fl(y))) + fl(z)) = \\ &= [(x(1+\varepsilon_x) + y(1+\varepsilon_y))(1+\varepsilon_A) + z(1+\varepsilon_z)](1+\varepsilon_B). \end{aligned}$$

Quindi l'errore relativo, tenendo conto che $u \geq u^2 \geq u^3$, è dato da

$$\begin{aligned}
\varepsilon_1 &= \frac{F_1 - R}{R} = \\
&= [(x(1+\varepsilon_x) + y(1+\varepsilon_y))(1+\varepsilon_A) + z(1+\varepsilon_z)](1+\varepsilon_B) = \\
&= \frac{x(1+\varepsilon_x)(1+\varepsilon_A)(1+\varepsilon_B) + y(1+\varepsilon_y)(1+\varepsilon_A)(1+\varepsilon_B)}{x+y+z} + \\
&\quad + \frac{z(1+\varepsilon_z)(1+\varepsilon_B) - x - y - z}{x+y+z} \leq \\
&\leq \frac{x(1+u)^3 + y(1+u)^3 + z(1+u)^2 - x - y - z}{x+y+z} = \\
&= \frac{x(3u+3u^2+u^3) + y(3u+3u^2+u^3) + z(2u+u^2)}{x+y+z} \leq \\
&\leq \frac{7ux + 7uy + 3uz}{x+y+z} = \\
&= u\left(3 + 4\frac{x+y}{x+y+z}\right).
\end{aligned}$$

- Analogamente per il secondo caso si ha che l'espressione in aritmetica finita è data da

$$\begin{aligned}
F_2 &= x \oplus (y \oplus z) = \\
&= [x(1+\varepsilon_x) + (y(1+\varepsilon_y) + z(1+\varepsilon_z))(1+\varepsilon_C)](1+\varepsilon_D),
\end{aligned}$$

  e quindi l'errore relativo corrispondente è dato da

$$\varepsilon_2 = \frac{F_2 - R}{R} = u\left(3 + 4\frac{y+z}{x+y+z}\right).$$

Si nota che l'unico modo per far diminuire l'errore relativo è di diminuire il numeratore, ovvero la quantità $(x+y)$, nel primo caso, e $y+z$ nel secondo. Quindi se ne deduce che, anche se in aritmetica esatta le due espressioni sono equivalenti, in aritmetica finita l'espressione migliore (ovvero quella che presenterà un minor errore sul risultato) sarà quella che somma per primi i due addendi con valore minore tra i tre presenti nell'espressione.

Esercizio 1.16. *Dimostrare che il numero di condizionamento del problema del calcolo di $y = \sqrt{x}$ è $k = 1/2$.*

Soluzione.
Applicando la (1.11), in questo caso specifico si ha $y = f(x) = \sqrt{x}$ ed $f'(x) = \dfrac{1}{2\sqrt{x}}$.
Quindi risulta che

$$k = \left|\frac{1}{2\sqrt{x}} \cdot \frac{x}{\sqrt{x}}\right| = \left|\frac{1}{2} \cdot \frac{x}{x}\right| = \frac{1}{2}.$$

L'estrazione da radice quadrata risulta quindi essere sempre ben condizionata.

Esercizio 1.17 (Cancellazione Numerica)**.** *Si supponga di dover calcolare l'espressione*

$$y = 0.12345678 - 0.12341234 \equiv 0.00004444,$$

utilizzando una rappresentazione decimale con arrotondamento alla quarta cifra significativa. Comparare il risultato esatto con quello ottenuto in aritmetica finita, e determinare la perdita di cifre significative derivante dalla operazione effettuata. Verificare che questo risultato è in accordo con l'analisi di condizionamento (vedi (1.12)).

Soluzione.
Il problema in questione è una somma algebrica con addendi discordi, in particolare

$$x_1 = 0.12345678, \qquad x_2 = -0.12341234, \qquad y = x_1 + x_2 = 0.00004444.$$

Arrotondando x_1 ed x_2 alla quarta cifra decimale si ottengono i dati di ingresso perturbati

$$\tilde{x_1} = 0.1235, \qquad \tilde{x_2} = -0.1234,$$

e la corrispondente soluzione del problema in aritmetica finita è calcolata come somma dei due addendi perturbati

$$\tilde{y} = \tilde{x_1} + \tilde{x_2} = 0.0001.$$

Calcoliamo adesso gli errori relativi

$$\varepsilon_1 = \frac{\tilde{x_1} - x_1}{x_1} = \frac{0.1235 - 0.12345678}{0.12345678} \approx 3.5 \cdot 10^{-4}$$

$$\varepsilon_2 = \frac{\tilde{x_2} - x_2}{x_2} = \frac{-0.1234 + 0.12341234}{-0.12341234} \approx -9.9 \cdot 10^{-5}$$

$$\varepsilon_y = \frac{\tilde{y} - y}{y} = \frac{0.0001 - 0.00004444}{0.00004444} \approx 1.25.$$

Risulta quindi che il risultato ottenuto è privo di significato in quanto l'errore relativo sulla soluzione maggiore di 1 implica una perdita totale di informazione. Ne consegue che sono state perse tutte le cifre significative della soluzione.
Applicando la (1.12) si ha che

$$k = \frac{|x_1| + |x_2|}{|x_1 + x_2|} = \frac{0.12345678 + 0.12341234}{0.00004444} \approx 5.6 \cdot 10^3$$

$$\varepsilon_x = max\{|\varepsilon_1|, |\varepsilon_2|\} = |\varepsilon_1| \approx 3.5 \cdot 10^{-4}.$$

Si vede quindi che, essendo il numero di condizionamento del problema $k \gg 1$, il problema risulta malcondizionato.
Se si prova a moltiplicare l'errore relativo massimo sui dati di ingresso ε_x per

il fattore di amplificazione calcolato k, si ottiene una maggiorazione dell'errore relativo sulla soluzione, che non si discosta di molto dal valore effettivamente calcolato

$$k\varepsilon_x \approx 5.6 \cdot 10^3 \cdot 3.5 \cdot 10^{-4} \approx 1.9 \approx 1.25 \approx \varepsilon_y.$$

Riferimenti MATLAB
Codice 1.7 (pagina 28)

Esercizio 1.18 (Cancellazione Numerica)**.** *Eseguire le seguenti istruzioni in* MATLAB*:*

```
format long e
a = 0.1
b = 0.099999999999
a-b
```

Valutare l'errore relativo sui dati di ingresso e l'errore relativo sul risultato ottenuto.

Soluzione.
Essendo gli addendi discordi e circa uguali in modulo, si ha che il problema è malcondizionato. Infatti il numero di condizionamento del problema vale

$$k = \frac{|a| + |-b|}{|a-b|} = \frac{0.1 + 0.099999999999}{0.000000000001} \approx 2 \cdot 10^{11}.$$

Se si considera che i dati in ingresso sono affetti da un errore relativo dell'ordine della precisione di macchina, ovvero $\varepsilon_x = u \approx 1.1 \cdot 10^{-16}$ (con doppia precisione, vedi Esercizio 1.10), allora l'errore relativo sulla soluzione finale sarà dell'ordine di

$$\varepsilon_y \approx k\varepsilon_x \approx 2 \cdot 10^{11} \cdot 1.1 \cdot 10^{-16} \approx 2.2 \cdot 10^{-5}.$$

Infatti si ha che eseguendo il codice MATLAB proposto risultano corrette (cioè 0) soltanto le prime 5 cifre della soluzione, come si evince anche dal fatto che

$$\lceil -\log_{10} |\varepsilon_y| \rceil \approx \lceil -\log_{10}(2.2 \cdot 10^{-5}) \rceil = 5.$$

Riferimenti MATLAB
Codice 1.8 (pagina 29)

Codice degli esercizi

Codice 1.1: Esercizio 1.1.

```
% Esercizio 1.1
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:04 CET

format short
e = (3.1415 - pi)/pi
cifre = fix(-log10(abs(e)))
```

Codice 1.2: Esercizio 1.4.

```
% Esercizio 1.4
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:04 CET

hold on
format long e;
tol = 1e-11;

x = zeros(1, 2);
delta = 2 - sqrt(2);
i = 1;
while(delta > tol)
    if(i==1)
        x(1, 1) = 2;
    elseif (i==2)
        x(2, 1) = 1.5;
    else
        x(i, 1) = (x(i-1, 1)*x(i-2, 1) +2)/(x(i-1, 1) + x(i-2, 1));
    end
    delta = x(i, 1) - sqrt(2);
    x(i, 2) = delta;
    i = i+1;
end
x
plot(1:i-1, x(1:i-1, 1), 'b');

x = zeros(1, 2);
delta = 2 - sqrt(2);
i = 1;
while (delta > tol)
   if (i==1)
       x(1, 1) = 2;
```

```
    else
        x(i, 1) = 1/2*(x(i-1, 1) + 2/x(i-1, 1));
    end
    delta = x(i, 1) - sqrt(2);
    x(i, 2) = delta;
    i = i+1;
end
x
plot(1:i-1, x(1:i-1, 1), 'r');

legend('primo metodo', 'secondo metodo')
title('Esercizio 1.4')
hold off
```

Codice 1.3: Esercizio 1.8.

```
% Esercizio 1.8
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:04 CET

m = ceil(-log2(4.66*10^-10))
```

Codice 1.4: Esercizio 1.10.

```
% Esercizio 1.10
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:04 CET

rMax = (2-2^-52)*2^1023
rMax - realmax
rMinN = 2^-1022
rMinN - realmin
rMinD = 2^-52*2^-1022
u=2^-53
```

Codice 1.5: Esercizio 1.11.

```
% Esercizio 1.11
```

```
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:10 CET

x=0; delta=0.1;
while abs(x-1)>esp, x=x+delta, end
```

Codice 1.6: Esercizio 1.12.

```
% Esercizio 1.12
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:10 CET

x=10^200
y=10^200

ris = sqrt(x^2 + y^2)
ris = abs(x)*sqrt(1+(y/x)^2)
```

Codice 1.7: Esercizio 1.17.

```
% Esercizio 1.17
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:10 CET

x1=0.12345678
x2=-0.12341234
y=0.00004444

x1t=0.1235
x2t=-0.1234
yt=x1t+x2t

e1=(x1t-x1)/x1
e2=(x2t-x2)/x2
ey=(yt-y)/y

k=(abs(x1)+abs(x2))/abs(x1+x2)
ex=e1

ey_max=k*ex
```

Codice 1.8: Esercizio 1.18.

```
% Esercizio 1.18
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:10 CET

a=0.1
b=0.099999999999

k=(abs(a)+abs(-b))/abs(a-b)
ex = eps/2
ey = k*ex
c=ceil(-log10(abs(ey)))
```

Capitolo 2

Radici di una equazione

Indice

In questo capitolo studieremo il problema del calcolo degli *zeri* reali di una funzione, ovvero risolvere l'equazione

$$f(x) = 0, \qquad x \in \mathbb{R}, \quad f : \mathbb{R} \to \mathbb{R}, \tag{2.1}$$

determinandone le *radici* reali.
In generale, questo tipo di problema, o ammette un numero *finito* di soluzioni, o non ammette *alcuna* soluzione, oppure ammette un numero *infinito* di soluzioni. Di seguito supporremo che la funzione ammetta almeno una radice reale, presupposto sempre verificato nel caso in cui la funzione risulti continua in un intervallo $[a, b]$ con $f(a)f(b) < 0$, per il Teorema di Bolzano (o Teorema degli Zeri).

2.1 Il metodo di bisezione

Il metodo di bisezione, per il calcolo di un'approssimazione di una radice di una funzione, si basa sul *Teorema di Bolzano* poc'anzi accennato. Si suppone

quindi che la funzione $f(x)$ considerata sia continua in un intervallo $[a, b]$ e che $f(a)f(b) < 0$: in questo caso si dice che l'intervallo $[a, b]$ costituisce un **intervallo di confidenza** per la radice dell'equazione (2.1).
Se denotiamo con x^* la radice ricercata, il miglior risultato che possiamo ottenere in un primo momento è di stimare che

$$x^* \approx x_1 \equiv \frac{a+b}{2},$$

che corrisponde al punto medio dell'intervallo $[a, b]$. In questo modo si ha che l'errore commesso sarà maggiorato dalla semi-ampiezza dell'intervallo di confidenza, ovvero

$$|x^* - x_1| \leq \frac{b-a}{2}.$$

A questo punto, per quanto riguarda il punto appena calcolato x_1, ci possiamo trovare in tre situazioni distinte e mutuamente esclusive:

- $f(x_1) = 0$: in questo caso (assai raro) si ha che il punto trovato è effettivamente uno zero della funzione e costituisce, quindi, una soluzione al problema;

- $f(a)f(x_1) < 0$: il procedimento precedente viene ripetuto sul nuovo intervallo di confidenza $[a, x_1]$;

- $f(x_1)f(b) < 0$: viene ripetuto il procedimento precedente sull'intervallo $[x_1, b]$.

Quindi si ha che, se non viene trovata la radice cercata, allora l'intervallo di confidenza viene dimezzato, reiterando il procedimento: per questo motivo questo metodo prende il nome di **Metodo di Bisezione**.
Come qualsiasi metodo iterativo è necessario definire un opportuno criterio d'arresto: in un primo momento si potrebbe pensare di terminare l'esecuzione quando viene trovato un punto x_i tale che $f(x_i) = 0$, ma tale criterio d'arresto risulta molto inefficace in quanto molto probabilmente quella condizione, specialmente in aritmetica finita, non si verificherà mai.

2.2 Criteri di arresto e condizionamento

Spesso si ha che non interessa determinare esattamente la radice della (2.1), ma una sua approssimazione entro una certa tolleranza tol_x, ovvero determinare un punto $\tilde{x}$ tale che

$$|x^* - \tilde{x}| \leq tol_x.$$

Tuttavia, non conoscendo il valore della radice x^*, si deve utilizzare tol_x come criterio d'arresto facendo una stima sul numero minimo di passi d'iterazione da eseguire per ottenere un errore sicuramente minore o uguale a tol_x.

Ricordiamo che l'errore commesso al primo passo è maggiorato dalla semi-ampiezza dell'intervallo di confidenza, ovvero che

$$|x^* - x_1| \leq \frac{b-a}{2}.$$

Analogamente si ha che al passo i-esimo $x_i = \frac{a_i + b_1}{2}$, con $[a_i, b_i]$ intervallo di confidenza al passo i-esimo, e l'errore commesso è maggiorato da

$$|x^* - x_i| \leq \frac{b_i - a_i}{2} = \frac{b_{i-1} - a_{i-1}}{2^2} = \cdots = \frac{b-a}{2^i}.$$

Quindi il numero di passi da eseguire per ottenere un errore sicuramente minore o uguale a tol_x è dato da

$$\frac{b-a}{2^i} \leq tol_x$$

$$2^i \geq \frac{b-a}{tol_x}$$

$$i \geq \lceil \log_2 \frac{b-a}{tol_x} \rceil = \lceil \log_2(b-a) - \log_2 tol_x \rceil$$

$$i_{max} \equiv \lceil \log_2(b-a) - \log_2 tol_x \rceil.$$

Inserendo quindi un limite superiore al numero di passi che si può effettuare si ha che l'algoritmo che descrive il metodo di bisezione terminerà sempre (cosa che non veniva garantita con il semplice controllo $f(x_i) = 0$). Tuttavia rimane il fatto che il controllo $f(x_i) = 0$ spesso risulta, specialmente in aritmetica finita, inefficace. Si può pensare allora di sfruttare una certa tolleranza anche sul valore che la funzione assume in un determinato punto, ricordando che, essendo $f(x)$ continua ed essendo $f(x^*) = 0$, in un opportuno intorno di x^* si avrà che i valori della funzione risulteranno "piccoli".
Pertanto si sostituirà il controllo $f(x_i) = 0$ con un controllo del tipo $|f(x_i)| \leq tol_f$, dove tol_f rappresenta la tolleranza sul valore della funzione scelta in modo tale che $|x_i - x^*| \leq tol_x$. Supponendo $f(x) \in C^{(1)}$, si sviluppa la funzione in x^* (trascurando l'errore di ordine quadratico commesso)

$$\begin{aligned} f(x) &= P_1(x; x^*) + O((x-x^*)^2) = \\ &= f(x^*) + (x-x^*)f'(x^*) + O((x-x^*)^2) \approx \\ &\approx f'(x^*)(x-x^*), \end{aligned}$$

da cui segue che

$$x - x^* \approx \frac{f(x)}{f'(x^*)}$$

$$|x - x^*| \approx \frac{|f(x)|}{|f'(x^*)|}. \tag{2.2}$$

Quindi, per avere $|x - x*| \approx tol_x$, occorre utilizzare una tolleranza sulla $f(x)$ che vale circa

$$tol_f \approx |f'(x^*)| \cdot tol_x.$$

Quindi si ha che

- $tol_f \ll tol_x$, se $f'(x^*) \approx 0$;
- $tol_f \gg tol_x$, se $f'(x^*) \gg 1$.

Ovvero si ha che, fissata tol_x, tanto più grande sarà la derivata $|f'(x^*)|$, tanto più grande sarà l'intorno di tol_f e quindi meno stringente tol_f stessa; viceversa fissata tol_f, tanto più grande sarà $|f'(x^*)|$, tanto più piccolo sarà l'intervallo di confidenza definito dall'intorno di tol_x.
Se al passo i-esimo si ha l'intervallo di confidenza $[a_i, b_i]$, un'approssimazione della derivata $f'(x^*)$ è data da

$$f'(x^*) \approx \frac{f(b_i) - f(a_i)}{b_i - a_i}.$$

Vediamo di seguito un'implementazione in MATLAB del metodo di bisezione, utilizzando i criteri d'arresto appena enunciati:

Codice 2.1: Metodo di bisezione.

```
% bisezione(f, a, b, tolx, s, g, gs)
% Metodo di bisezione.
%
% Input:
%   -f: la funzione;
%   -a: estremo sinistro dell'intervallo di confidenza;
%   -b: estremo destro dell'intervallo di confidenza;
%   -tolx: la tolleranza desiderata;
%   -s: true per visualizzare ogni singolo step, false altrimenti;
%   -g: true per abilitare la parte grafica, false altrimenti;
%   -gs: true per vedere ogni step del grafico, false per vedere subito
%   il grafico finale.
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:04 CET

function [] = bisezione(f, a, b, tolx, s, g, gs)
    tStart = tic;
    if g
        hax=axes; hold on
        fplot(f, [a-0.5, b+0.5], 'b');
        fplot(@(x) 0, get(hax, 'XLim'), 'black');
        line([a a], get(hax, 'YLim'), 'Color', [0 0 0])
        line([b b], get(hax, 'YLim'), 'Color', [0 0 0])
        if gs, pause, end
    end
    imax = ceil( log2(b-a) - log2(tolx) );
```

```
    fa = feval(f, a);
    fb = feval(f, b);
    i = 0;
    while ( i<imax )
        x = (a+b)/2;
        if s, str = sprintf('x%d =', i); disp(str), disp(x), end
        fx = feval(f, x);
        f1x = abs( (fb-fa)/(b-a) );
        if abs(fx)<=tolx*f1x
            break
        elseif fa*fx<0
            if g
                YLim=get(hax,'YLim'); rectangle('Position', [x YLim(1)
                    abs(b-x) abs(YLim(2)-YLim(1))], 'FaceColor', [0 0
                    0])
                if gs, pause, end
            end
            b = x; fb = fx;
        else
            if g
                YLim=get(hax,'YLim'); rectangle('Position', [a YLim(1)
                    abs(x-a) abs(YLim(2)-YLim(1))], 'FaceColor', [0 0
                    0])
                if gs, pause, end
            end
            a = x; fa = fx;
        end
        i = i+1;
    end
    if g, line([x x], get(hax, 'YLim'), 'Color', [1 0 0]), end
    str=sprintf('Converge a %5.4f in %d passi', x, i); disp(str)
    tElapsed = toc(tStart); str=sprintf('Tempo medio/step: %5.4f ms;
        tempo totale %5.4f ms', tElapsed*1000/i, tElapsed*1000); disp(
        str)
```

Un altro importante aspetto dell'analisi dei criteri d'arresto per il metodo di bisezione è il condizionamento della radice x^*. Se si interpreta nella (2.2) il valore $f(x)$ della funzione come perturbazione del valore nullo e $|x-x^*|$ l'errore commesso sulla soluzione finale del problema, allora tale perturbazione sui dati di ingresso risulta amplificata di un fattore

$$k = \frac{1}{f'(x^*)}$$

durante l'esecuzione del metodo.
Se ne deduce che k può essere definito come **numero di condizionamento della radice** x^*.

Definizione 2.1. *Una radice x^* ha **molteplicità esatta** $m \geq 1$, se*

$$f(x^*) = f'(x^*) = \cdots = f^{(m-1)}(x^*) = 0, \quad f^{(m)}(x^*) \neq 0.$$

*Se $m = 1$ la radice si dice **semplice**; altrimenti, se $m \geq 2$, si dice **multipla**.*

Quindi si ha che il problema delle radici multiple è sempre malcondizionato in quanto, essendo $f'(x^*) = 0$, risulta che k assume un valore virtualmente infinito. Si ha, infine, che se $f(x)$ è sviluppabile in serie di Taylor in x^* (radice di $f(x)$ di molteplicità m), allora tale funzione si può scrivere come

$$f(x) = (x - x^*)^m g(x),$$

con $g(x)$ funzione ancora sviluppabile in serie di Taylor in x^* e tale che $g(x^*) \neq 0$.

2.3 Ordine di convergenza

Supponendo di utilizzare un generico metodo iterativo per l'approssimazione di una radice x^* dell'equazione (2.1), denotiamo con x_i l'approssimazione fornita dal metodo al passo i-esimo e e_i l'errore commesso corrispondente

$$e_i \equiv x_i - x^*.$$

Quindi, come visto nella Sezione 1.2, tale metodo si dirà *convergente* se

$$\lim_{i \to \infty} e_i = 0.$$

Ovviamente la convergenza è un requisito fondamentale per un metodo numerico iterativo, ma è interessante studiare anche la "velocità" con la quale il metodo converge verso la soluzione.
Ad esempio, come già osservato nelle sezioni precedenti, per il metodo di bisezione risulta che ad ogni passo d'iterazione l'errore viene dimezzato, infatti si può dimostrare che

$$\lim_{i \to \infty} \frac{|e_{i+1}|}{|e_i|} = \frac{1}{2}.$$

Definizione 2.2. *Se per un certo metodo iterativo si ha che p è il più grande valore reale per cui vale*

$$\lim_{i \to \infty} \frac{|e_{i+1}|}{|e_i|^p} = p < \infty, \tag{2.3}$$

allora si dice che il metodo ha ***ordine di convergenza*** *p con* ***costante asintotica dell'errore*** *pari a c.*
Nel caso di $p = 1$ si parla di ***convergenza lineare****, nel caso $p = 2$ di* ***convergenza quadratica****, ecc.*

Quindi si ha che il metodo di bisezione ha ordine di convergenza lineare ($p = 1$) con costante asintotica dell'errore $c = 1/2$.
L'ordine di convergenza p può assumere anche valori non interi, anche se deve necessariamente risultare $p \geq 1$ affinché il metodo converga.
Si nota che, per $p = 1$ ed i sufficientemente grande, vale

$$|e_{i+1}| \approx c|e_i|,$$
$$|e_{i+k}| \approx c^k|e_i|,$$

pertanto un metodo iterativo con $p = 1$ converge se e solo se $0 \leq c < 1$.
In generale si ha che tanto più elevato è il l'ordine di convergenza di un metodo iterativo, tanto più velocemente esso convergerà verso una radice x^*.

2.4 Il metodo di Newton

Supponiamo di disporre di un'approssimazione iniziale x_0 della radice x^*. La retta tangente al grafico nel punto $(x_0, f(x_0))$ è data dall'equazione

$$y = f(x_0) + f'(x_0)(x - x_0).$$

La nuova approssimazione x_1 della radice x^* è data dall'intersezione tra la suddetta retta tangente e la retta delle ascisse, ovvero

$$0 = f(x_0) + f'(x_0)(x_1 - x_0)$$

$$x_1 f'(x_0) = x_0 f'(x_0) - f(x_0)$$

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)},$$

che è definita se $f'(x_0) \neq 0$.
Quindi al passo $(i+1)$-esimo si ha che l'espressione funzionale del **metodo di Newton** è

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}, \qquad i = 0, 1, 2, \ldots \tag{2.4}$$

Quindi il metodo di Newton consiste nella risoluzione successiva di equazioni lineari: ovviamente nel caso in cui la $f(x)$ sia una funzione lineare, il metodo fornisce il risultato in un solo passo.
Rispetto al metodo di bisezione, che richiedeva come requisito la sola continuità della funzione, il metodo di Newton ne richiede anche la derivabilità. Inoltre, ad ogni passo, oltre alla valutazione della $f(x_i)$ si deve calcolare anche la derivata prima $f'(x_i)$.
Questi svantaggi del metodo di Newton rispetto al metodo di bisezione vengono tuttavia ripagati dall'ordine di convergenza del metodo di Newton: infatti vale

Teorema 2.1. *Se $f(x)$ è sufficientemente regolare (cioè $f \in C^{(1)}$), il metodo di Newton converge quadraticamente verso radici semplici.*

Teorema 2.2. *Se $f(x)$ è sufficientemente regolare, il metodo di Newton converge linearmente verso una radice di molteplicità $m > 1$, con costante asintotica dell'errore pari a $(m-1)/m$.*

2.5 Convergenza locale

Studiamo adesso quali sono le condizioni *necessarie* affinché un metodo iterativo converga.
Si è visto che per il metodo di bisezione basta assumere che la $f(x)$ sia continua in un intervallo $[a, b]$, con $f(a)f(b) < 0$, per poter affermare che il metodo converge *sempre* verso una radice: in questo caso si parla di **convergenza globale**.
In generale la convergenza globale è una prerogativa del metodo di bisezione. Per tutti gli altri metodi si cerca invece di garantire la convergenza in un opportuno

intorno della radice: in questo caso si parla di **convergenza locale**. Come visto in Sezione 1.2, se denotiamo con $\Phi(x)$ la funzione d'iterazione del metodo, allora il metodo iterativo può essere formalizzato come

$$x_{i+1} = \Phi(x_i), \qquad i = 0, 1, 2, \ldots.$$

Ovviamente (si veda l'Esercizio 1.3) deve valere che lo zero x^* è un *punto fisso* della funzione d'iterazione, ovvero che

$$\Phi(x^*) = x^*.$$

Le proprietà di convergenza vengono studiate attraverso lo studio della stabilità del punto fisso.

Teorema 2.3 (Teorema del punto fisso)**.** *Sia $\Phi(x)$ la funzione d'iterazione che definisce il metodo numerico. Si supponga che esistano $\delta > 0$ e $0 \leq L < 1$ tali che, definendo l'intervallo $I = (x^* - \delta, x^* + \delta)$,*

$$|\Phi(x) - \Phi(y)| \leq L|x - y|, \qquad \forall x, y \in I.$$

Allora

- *x^* è l'unico punto fisso di $\Phi(x)$ in I;*
- *se $x_0 \in I$, allora $x_i \in I$, $i = 0, 1, 2, \ldots$;*
- *$\lim_{i\to\infty} x_i = x^*$.*

Supponendo $f \in C^{(2)}$ si può utilizzare il Teorema del punto fisso per dimostrare che il metodo di Newton converge localmente.

2.6 Ancora sul criterio d'arresto

Cerchiamo adesso di valutare un criterio d'arresto efficace per il metodo di Newton.
A causa della convergenza locale del metodo di Newton non è possibile stabilire a priori il numero minimo di passi entro il quale si otterrà un'approssimazione della radice con l'accuratezza richiesta. Tuttavia si ha che per metodi con ordine di convergenza $p > 1$ (come il metodo di Newton), in prossimità della radice x^*

$$|x_{i+1} - x_i| = |x_{i+1} - x^* + x^* - x_i| = |e_i - e_{i+1}| \approx |e_i|,$$

essendo, in prossimità della radice, e_{i+1} trascurabile rispetto a e_i. Quindi un criterio d'arresto efficace potrebbe essere

$$|x_{i+1} - x_i| \leq tol_x.$$

Nel caso del metodo di Newton, per la (2.4), il criterio d'arresto risulta

$$|f(x_i)| \leq |f'(x_i)| \cdot tol_x.$$

Di seguito, il codice del metodo di Newton, tenendo conto del criterio d'arresto appena illustrato, in MATLAB:

Codice 2.2: Metodo di Newton.

```
% newton(f, f1, x0, imax, tolx, s, g, gs)
% Metodo di Newton generico.
%
% Input:
%   -f: la funzione;
%   -f1: la derivata della funzione;
%   -x0: l'approssimazione iniziale;
%   -imax: il numero massimo di iterazioni;
%   -tolx: la tolleranza desiderata;
%   -s: true per visualizzare ogni singolo step, false altrimenti;
%   -g: true per abilitare la parte grafica, false altrimenti;
%   -gs: true per vedere ogni step del grafico, false per vedere subito
%   il grafico finale.
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:16 CET

function [] = newton(f, f1, x0, imax, tolx, s, g, gs)
    tStart = tic;
    format long e
    if g
        hax=axes; hold on
        fplot(f, [x0-5, x0+5], 'b');
        fplot(@(x) 0, get(hax, 'XLim'), 'black');
        if gs, pause(tStart), pause, start(tStart), end
    end
    if s, disp('x0 ='), disp(x0), end
    i = 0;
    vai = true;
    while ( i<imax ) && vai
        i = i+1;
        fx = feval(f, x0);
        f1x = feval(f1, x0);
        if f1x==0, vai=false; i=i-1; break, end
        x1 = x0 - fx/f1x;
        if s, str = sprintf('x%d =', i); disp(str), disp(x1), end
        if g
            plot([x0 x1], [fx 0], 'r');
            plot([x0 x0], [0 fx], 'black');
            if gs, pause(tStart), pause, start(tStart), end
        end
        vai = abs(x1-x0)>tolx;
        x0 = x1;
    end
    if vai, disp('NON CONVERGE entro la tolleranza ed il numero di
        passi richiesti')
    else str=sprintf('CONVERGE a %5.4f dopo %d passi', x1, i); disp(str
        ), end
    if g, hold off, end
    tElapsed = toc(tStart); str=sprintf('Tempo medio/step: %5.4f ms;
        tempo totale %5.4f ms', tElapsed*1000/i, tElapsed*1000); disp(
        str)
```

Spesso si assegna anche una tolleranza relativa $Rtol_x$, in tal caso si ha che (considerando le approssimazioni $x^* \approx x_{i+1}$ e $|e_i| \approx |x_{i+1} - x_i|$)

$$\frac{|e_i|}{tol_x + Rtol_x \cdot |x^*|} \leq 1.$$

Inoltre spesso si considera la scelta $Rtol_x = tol_x$ per la tolleranza relativa: in questo caso il criterio d'arresto diventa

$$\frac{1}{tol_x} \cdot \frac{|x_{i+1} - x_i|}{1 + |x_{i+1}|} \leq 1$$

$$\frac{|x_{i+1} - x_i|}{1 + |x_{i+1}|} \leq tol_x.$$

In questo modo viene controllato l'errore assoluto quando $x_{i+1} \approx 0$, e l'errore relativo quando $|x_{i+1}| \gg 1$.

Per quanto riguarda metodi con ordine di convergenza lineare si ha che

$$|x_{i+1} - x_i| = |e_i - e_{i+1})| \approx |e_i|(1 - c),$$

con c costante asintotica dell'errore e approssimando $\frac{e_{i+1}}{e_i} \approx c$ (vedi (2.3)). Quindi si ha che

$$|e_{i+1}| \approx c|e_i| \approx c \cdot \frac{1}{1-c}|x_{i+1} - x_i|,$$

e quindi, in questo caso, un criterio d'arresto appropriato risulta essere

$$|x_{i+1} - x_i| \leq \frac{1-c}{c} tol_x.$$

Per stimare la costante asintotica dell'errore c si consideri che

$$|x_1 - x_0| \approx (1-c)|e_0, \qquad |x_2 - x_1| \approx (1-c)|e_1| \approx (1-c)c|e_0|,$$

da cui si ottiene una prima stima di c:

$$c \approx \frac{|x_2 - x_1|}{|x_1 - x_0|}.$$

Questa stima richiede che siano eseguite almeno due iterazioni del metodo, ma può essere tenuta costantemente aggiornata ad ogni passo considerando le tre iterazioni più recenti.

2.7 Il caso di radici multiple

Si è visto che nel caso di radici multiple il problema del calcolo di un'approssimazione della radice diventa malcondizionato e che il metodo di Newton ha

ordine di convergenza lineare (anziché quadratico). Con opportuni accorgimenti sarà possibile ripristinare la convergenza quadratica.
Si distinguono quindi due casi:

La molteplicità della radice è nota

Supponiamo per semplicità che la funzione presa in analisi sia della forma

$$f(x) = (x - x^*)^m.$$

Applicando il metodo di Newton per determinare la radice si ottiene

$$x_{i+1} = x_i - \frac{(x_i - x^*)^m}{m(x_i - x^*)^{m-1}} = x_i - \frac{x_i - x^*}{m}, \quad i = 0, 1, 2, \ldots.$$

Moltiplicando per m il *termine di correzione* a x_i si ottiene

$$x_{i+1} = x_i - m\frac{x_i - x^*}{m} = x_i - x_i + x^* = x^*,$$

ottenendo così la soluzione esatta in solo passo.
Più in generale si può dimostrare che, conoscendo la molteplicità m della radice da approssimare, lo schema iterativo

$$x_{i+1} = x_i - m\frac{f(x_i)}{f'(x_i)}, \qquad i = 0, 1, 2, \ldots, \tag{2.5}$$

ripristina la convergenza quadratica al metodo di Newton.

Codice 2.3: Metodo di Newton modificato.

```
% newtonMod(f, f1, x0, m, imax, tolx, s, g, gs)
% Metodo di Newton modificato.
%
% Input:
%   -f: la funzione;
%   -f1: la derivata della funzione;
%   -x0: l'approssimazione iniziale;
%   -m: la molteplicità della radice;
%   -imax: il numero massimo di iterazioni;
%   -tolx: la tolleranza desiderata;
%   -s: true per visualizzare ogni singolo step, false altrimenti;
%   -g: true per abilitare la parte grafica, false altrimenti;
%   -gs: true per vedere ogni step del grafico, false per vedere subito
%   il grafico finale.
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:18 CET

function [] = newtonMod(f, f1, x0, m, imax, tolx, s, g, gs)
    tStart = tic;
    format long e
```

```
    if g
        hax=axes; hold on
        fplot(f, [x0-5, x0+5], 'b');
        fplot(@(x) 0, get(hax, 'XLim'), 'black');
        if gs, pause, end
    end
    if s, disp('x0 ='), disp(x0), end
    i = 0;
    vai = true;
    while ( i<imax ) && vai
        i = i+1;
        fx = feval(f, x0);
        f1x = feval(f1, x0);
        if f1x==0, vai=false; i=i-1; break, end
        x1 = x0 - m*(fx/f1x);
        if s, str = sprintf('x%d =', i); disp(str), disp(x1), end
        if g
            plot([x0 x1], [fx 0], 'r');
            plot([x0 x0], [0 fx], 'black');
            if gs, pause, end
        end
        vai = abs(x1-x0)>tolx;
        x0 = x1;
    end
    if vai, disp('NON CONVERGE entro la tolleranza ed il numero di
        passi richiesti')
    else str=sprintf('CONVERGE a %5.4f dopo %d passi', x1, i); disp(str
        ), end
    if g, hold off, end
    tElapsed = toc(tStart); str=sprintf('Tempo medio/step: %5.4f ms;
        tempo totale %5.4f ms', tElapsed*1000/i, tElapsed*1000); disp(
        str)
```

La molteplicità della radice è incognita

Per il Teorema 2.2 sappiamo che, nel caso di radici multiple, il metodo di Newton converge soltanto linearmente. Quindi, per la (2.3) si ha che

$$\frac{e_i}{e_{i-1}} \approx c, \qquad \frac{e_{i+1}}{e_i} \approx c$$

$$e_i \approx c e_{i-1}, \qquad e_{i+1} \approx c e_i.$$

Combinando queste due si può eliminare la costante asintotica dell'errore c, ottenendo

$$e_{i+1} e_{i-1} \approx e_i^2$$

$$(x_{i+1} - x^*)(x_{i-1} - x^*) \approx (x_i - x^*)^2.$$

Supponendo esatta l'uguaglianza si ottiene la seguente approssimazione della radice

$$x_{i-1}x_{i+1} - x_{i+1}x^* - x_{i-1}x^* + x^{*^2} \approx x_i^2 - 2x_i x^* + x^{*^2}$$

$$(x_{i-1} - 2x_i + x_{i+1})x^* \approx x_{i-1}x_{i+1} - x_i^2$$

$$x^* \approx x_i^* \equiv \frac{x_{i-1}x_{i+1} - x_i^2}{x_{i-1} - 2x_i + x_{i+1}}. \tag{2.6}$$

La procedura viene reiterata a partire dall'approssimazione ottenuta x_i^*.
Quindi questa procedura, denominata **metodo di accelerazione di Aitken**, è divisa, per ogni passo d'iterazione, in due parti:

1. una prima parte nella quale vengono eseguiti due passi del metodo di Newton, calcolando x_i ed x_{i+1} (l'approssimazione x_{i-1} è quella calcolata alla fine del passo precedente);

2. una seconda parte di *accelerazione* dove viene calcolata l'approssimazione fornita dalla (2.6), che fornisce un'approssimazione più accurata della radice, la quale costituirà il punto di partenza per l'iterazione successiva.

Si può dimostrare che la successione delle approssimazioni ottenute mediante l'accelerazione di Aitken converge quadraticamente verso la radice x^*. L'unico svantaggio è che ogni iterazione, rispetto al metodo di Newton, ha un costo doppio.
Proponiamo, di seguito, un'implementazione in MATLAB del metodo di Aitken:

Codice 2.4: Metodo di Aitken.

```
% aitken(f, f1, x0, imax, tolx, s, g, gs)
% Metodo di accelerazione di Aitken.
%
% Input:
%   -f: la funzione
%   -f1: la derivata della funzione;
%   -x0: l'approssimazione iniziale;
%   -imax: il numero massimo di iterazioni;
%   -tolx: la tolleranza desiderata;
%   -s: true per visualizzare ogni singolo step, false altrimenti;
%   -g: true per abilitare la parte grafica, false altrimenti;
%   -gs: true per vedere ogni step del grafico, false per vedere subito
%   il grafico finale.
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:04 CET

function [] = aitken(f, f1, x0, imax, tolx, s, g, gs)
    tStart = tic;
    format long e
    if g
        hax=axes; hold on
        fplot(f, [x0-5, x0+5], 'b');
        fplot(@(x) 0, get(hax, 'XLim'), 'black');
        if gs, pause, end
    end
    if s, disp('x0 ='), disp(x0), end
    i = 0;
    vai = true;
```

```
    while ( i<imax ) && vai
31      i = i+1;
        fx = feval(f, x0);
33      f1x = feval(f1, x0);
        if f1x==0, vai=false; i=i-1; break, end
35      x1 = x0 - fx/f1x;
        if g
37          plot([x0 x1], [fx 0], 'r');
            plot([x0 x0], [0 fx], 'black');
39          if gs, pause, end
        end
41      fx = feval(f, x1);
        f1x = feval(f1, x1);
43      if f1x==0, vai=false; i=i-1; break, end
        x2 = x1 - fx/f1x;
45      if g
            plot([x1 x2], [fx 0], 'r');
47          plot([x1 x1], [0 fx], 'black');
            if gs, pause, end
49      end
        if (x2-2*x1+x0)==0, vai=false; i=i-1; break, end
51      x3 = (x2*x0-x1^2)/(x2-2*x1+x0);
        if s, str = sprintf('x%d =', i); disp(str), disp(x3), end
53      if g
            fx = feval(f, x2);
55          plot([x2 x3], [fx 0], 'g');
            plot([x2 x2], [0 fx], 'black');
57          if gs, pause, end
        end
59      vai = abs(x3-x0)>tolx;
        x0 = x3;
61  end
    if vai, disp('NON CONVERGE entro la tolleranza ed il numero di
        passi richiesti')
63  else str=sprintf('CONVERGE a %5.4f dopo %d passi', x3, i); disp(str
        ), end
    if g, hold off, end
65  tElapsed = toc(tStart); str=sprintf('Tempo medio/step: %5.4f ms;
        tempo totale %5.4f ms', tElapsed*1000/i, tElapsed*1000); disp(
        str)
```

2.8 Metodi quasi-Newton

I metodi quasi-Newton sono varianti del metodo di Newton che non richiedono il calcolo della derivata prima (che computazionalmente rappresenta il costo maggiore per iterazione).
Lo schema generale sarà

$$x_{i+1} = x_i - \frac{f(x_i)}{\varphi_i}, \qquad i = 0, 1, 2, \ldots, \quad \varphi_i \approx f'(x_i),$$

dove φ_i è un'approssimazione che sostituisce la derivata prima $f'(x_i)$. A seconda dell'approssimazione φ_i utilizzata si avranno diversi metodi quasi-Newton che,

essendo varianti del metodo di Newton, godono della proprietà di convergenza locale.

Metodo delle corde

Il **metodo delle corde** parte dal presupposto che la funzione $f(x)$ sia sufficientemente regolare e che l'approssimazione iniziale x_0 sia prossima alla radice. Quando si verifica ciò, intuitivamente, si ha che la derivata della $f(x)$ varia molto poco in prossimità della radice. Quindi si può convenientemente utilizzare l'approssimazione

$$f'(x_i) \approx f'(x_0) \equiv \varphi_i.$$

In questo modo si ottiene l'iterazione

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_0)}, \qquad i = 0, 1, 2, \ldots.$$

Ad ogni iterazione si deve quindi calcolare soltanto la $f(x_i)$, pertanto il costo per iterazione è uguale a quello del metodo di bisezione e, come quest'ultimo, si può dimostrare che il metodo delle corde ha ordine di convergenza lineare.
Il principale vantaggio del metodo delle corde è il basso costo computazionale rispetto ad altri metodi.

Codice 2.5: Metodo delle corde.

```
% corde(f, f1, x0, imax, tolx, s, g, gs)
% Metodo delle corde.
%
% Input:
%   -f: la funzione;
%   -f1: la derivata della funzione;
%   -x0: l'approssimazione iniziale;
%   -imax: il numero massimo di iterazioni;
%   -tolx: la tolleranza desiderata;
%   -s: true per visualizzare ogni singolo step, false altrimenti;
%   -g: true per abilitare la parte grafica, false altrimenti;
%   -gs: true per vedere ogni step del grafico, false per vedere subito
%   il grafico finale.
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:04 CET

function [] = corde(f, f1, x0, imax, tolx, s, g, gs)
    tStart = tic;
    format long e
    if g
        hax=axes; hold on
        fplot(f, [x0-5, x0+5], 'b');
        fplot(@(x) 0, get(hax, 'XLim'), 'black');
        if gs, pause, end
    end
```

```
    if s, disp('x0 ='), disp(x0), end
    f1x0 = feval(f1, x0);
    i = 0;
    vai = true;
    while ( i<imax ) && vai
        i = i+1;
        fx = feval(f, x0);
        if f1x0==0, vai=false; i=i-1; break, end
        x1 = x0 - fx/f1x0;
        if s, str = sprintf('x%d =', i); disp(str), disp(x1), end
        if g
            plot([x0 x1], [fx 0], 'r');
            plot([x0 x0], [0 fx], 'black');
            if gs, pause, end
        end
        vai = abs(x1-x0)>tolx;
        x0 = x1;
    end
    if vai, disp('NON CONVERGE entro la tolleranza ed il numero di
        passi richiesti')
    else str=sprintf('CONVERGE a %5.4f dopo %d passi', x1, i); disp(str
        ), end
    if g, hold off, end
    tElapsed = toc(tStart); str=sprintf('Tempo medio/step: %5.4f ms;
        tempo totale %5.4f ms', tElapsed*1000/i, tElapsed*1000); disp(
        str)
```

Metodo delle secanti

Supponiamo di essere al passo i-esimo d'iterazione ed aver già calcolato x_{i-1}, x_i, $f(x_{i-1})$ ed $f(x_i)$. Si considera la seguente approssimazione della derivata prima di $f(x_i)$:

$$f(x_i) \approx \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}} \equiv \varphi_i,$$

che corrisponde geometricamente alla retta secante (da qui il nome, **metodo delle secanti**) il grafico passante per i punti $(x_{i-1}, f(x_{i-1}))$ ed $(x_i, f(x_i))$.
Quindi l'iterazione corrispondente sarà

$$\begin{aligned} x_{i+1} &= x_i - f(x)\frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})} = \\ &= \frac{f(x_i)x_i - f(x_{i-1}x_i - f(x_i)x_i + f(x_i)x_{i-1})}{f(x_i) - f(x_{i-1})} = \\ &= \frac{f(x_i)x_{i-1} - f(x_{i-1})x_i}{f(x_i) - f(x_{i-1})}, \qquad i = 0, 1, 2, \ldots, \end{aligned}$$

con x_0 ed x_1 approssimazioni iniziali assegnate (spesso x_1 viene calcolata applicando un passo del metodo di Newton con x_0 come punto iniziale).
Si può dimostrare che l'ordine di convergenza del metodo delle secanti è

$$p = \frac{1 + \sqrt{5}}{2} \approx 1.618,$$

per radici semplici: quest'ordine di convergenza, più grande di quello lineare e più piccolo di quello quadratico, viene detto *superlineare* ed il valore $\frac{1+\sqrt{5}}{2}$ è detto *sezione aurea*. Per radici multiple, invece, la convergenza è lineare.
Il costo per iterazione corrisponde alla sola valutazione della $f(x_i)$, quindi risulta identico a quello del metodo delle corde e del metodo di bisezione, sebbene il metodo delle secanti converga più velocemente di questi ultimi due.
Di seguito, l'implementazione in MATLAB del metodo delle secanti:

Codice 2.6: Metodo delle secanti.

```
% secanti(f, f1, x0, imax, tolx, s, g, gs)
% Metodo delle secanti.
%
% Input:
%   -f: la funzione;
%   -f1: la derivata della funzione;
%   -x0: l'approssimazione iniziale;
%   -imax: il numero massimo di iterazioni;
%   -tolx: la tolleranza desiderata;
%   -s: true per visualizzare ogni singolo step, false altrimenti;
%   -g: true per abilitare la parte grafica, false altrimenti;
%   -gs: true per vedere ogni step del grafico, false per vedere subito
%   il grafico finale.
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:23 CET

function [] = secanti(f, f1, x0, imax, tolx, s, g, gs)
    tStart = tic;
    format long e
    if g
        hax=axes; hold on
        fplot(f, [x0-5, x0+5], 'b');
        fplot(@(x) 0, get(hax, 'XLim'), 'black');
        if gs, pause, end
    end
    if s, disp('x0 ='), disp(x0), end
    i = 1;
    fx0 = feval(f, x0);
    f1x = feval(f1, x0);
    if f1x==0
        vai=false; i=i-1;
    else
        x1 = x0 -fx0/f1x;
        if s, disp('x1 ='), disp(x1), end
        if g
            plot([x0 x1], [fx0 0], 'r');
            plot([x0 x0], [0 fx0], 'black');
            if gs, pause, end
        end
        vai = abs(x1-x0)>tolx;
    end
```

```
    while ( i<imax ) && vai
        i = i+1;
        fx1 = feval(f, x1);
        if (fx1-fx0)==0, vai=false; i=i-1; break, end
        x2 = (fx1*x0 - fx0*x1)/(fx1-fx0);
        if s, str = sprintf('x%d =', i); disp(str), disp(x1), end
        if g
            plot([x1 x2], [fx1 0], 'r');
            plot([x1 x1], [0 fx1], 'black');
            if gs, pause, end
        end
        vai = abs(x2-x1)>tolx;
        fx0 = fx1; x0 = x1; x1 = x2;
    end
    if vai, disp('NON CONVERGE entro la tolleranza ed il numero di
        passi richiesti')
    else str=sprintf('CONVERGE a %5.4f dopo %d passi', x1, i); disp(str
        ), end
    if g, hold off, end
    tElapsed = toc(tStart); str=sprintf('Tempo medio/step: %5.4f ms;
        tempo totale %5.4f ms', tElapsed*1000/i, tElapsed*1000); disp(
        str)
```

Riassumendo:

Metodo	**Richiede il calcolo della**		**Ordine di convergenza**			
	$f(x)$	$f'(x)$	**radici semplici**		**radici multiple**	
Bisezione	✓	✗	1	lineare	1	lineare
Newton	✓	✓	2	quadratico	1	lineare
Aitken	✓	✓	2	quadratico	2	quadratico
Corde	✓	✗	1	lineare	1	lineare
Secanti	✓	✗	$\frac{1+\sqrt{5}}{2} \approx 1.618$	superlineare	1	lineare

Metodo	**Funzione d'iterazione**
Bisezione	$x_{i+1} = \frac{a_i + b_i}{2}$
Newton	$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$
Aitken	$x_i^* = \frac{x_{i-1}x_{i+1} - x_i^2}{x_{i-1} - 2x_i + x_{i+1}}$
Corde	$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_0)}$
Secanti	$x_{i+1} = \frac{f(x_i)x_{i-1} - f(x_{i-1})x_i}{f(x_i) - f(x_{i-1})}$

Esercizi

Esercizio 2.1. *Definire una procedura iterativa basata sul metodo di Newton per determinare $\sqrt{\alpha}$, per un assegnato $\alpha > 0$. Costruire una tabella delle approssimazioni relative al caso $\alpha = x_0 = 2$ (comparare con la tabella dell'Esercizio 1.4).*

Soluzione.
Essendo $\sqrt{\alpha}$ la radice ricercata, dobbiamo innanzitutto trovare una funzione $f(x)$ che abbia uno zero in $x = \sqrt{\alpha}$. La funzione più semplice di questo tipo è $f(x) = x - \sqrt{\alpha}$, ma ovviamente, dato che si sta tentando di approssimare $\sqrt{\alpha}$ stessa, non è verosimile utilizzare il valore esatto per il calcolo dell'approssimazione. Quindi si utilizza la funzione $f(x) = x^2 - \alpha$, che ha radici semplici in $x = \sqrt{\alpha}$ e in $x = -\sqrt{\alpha}$, ovvero $f(\pm\sqrt{\alpha}) = 0$. La derivata prima di questa funzione è $f'(x) = 2x$.
L'iterazione del metodo di Newton utilizzando questa funzione diventa

$$\begin{aligned} x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{x_i^2 - \alpha}{2x_i} = \\ = \frac{2x_i^2 - x_i^2 + \alpha}{2x_i} = \frac{x_i^2 + \alpha}{2x_i} = \\ = \frac{1}{2}\left(x_i + \frac{\alpha}{x_i}\right), \quad i = 0, 1, 2, \ldots. \end{aligned}$$

Di seguito, l'implementazione in MATLAB di questa particolare istanza del metodo di Newton:

Codice 2.7: Metodo di Newton per il calcolo di $\sqrt{\alpha}$.

```
% newtonSqrt(alpha, x0, imax, tolx)
% Metodo di Newton ottimizzato per l'approssimazione della radice
% quadrata.
%
% Input:
%   -alpha: l'argomento della radice quadrata;
%   -x0: l'approssimazione iniziale;
%   -imax: il numero massimo di iterazioni;
%   -tolx: la tolleranza desiderata.
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:19 CET

function [] = newtonSqrt(alpha, x0, imax, tolx)
    format long e
    disp('x0 ='), disp(x0)
    x = (x0+alpha/x0)/2;
    disp('x1 ='), disp(x)
    i = 1;
    while( i<imax ) && ( abs(x-x0)>tolx )
       i = i+1;
       x0 = x;
```

```
        x = (x0+alpha/x0)/2;
24      str = sprintf('x%d =', i);
        disp(str), disp(x)
26  end
    if abs(x-x0)>tolx
28      disp('NON CONVERGE entro la tolleranza ed il numero di passi
            richiesti')
    else
30      disp('CONVERGE entro la tolleranza ed il numero di passi
            richiesti')
    end
```

Eseguendo questa procedura con $\alpha = 2$ ed approssimazione iniziale $x_0 = 2$ otteniamo la seguente successione di approssimazioni:

i	x_i
0	2
1	1.5
2	1.416666666666667...
3	1.414215686274510...
4	1.414213562374690...
5	1.414213562373095...
6	1.414213562373095...

Si vede quindi, comparando le approssimazioni con quelle viste nella tabella dell'Esercizio 1.4, che già alla quarta iterazione (cioè per $i = 4$) l'errore di convergenza è dell'ordine di 10^{-12}.

Riferimenti MATLAB
Codice 2.10 (pagina 58)

Esercizio 2.2. *Generalizzare il risultato del precedente esercizio, derivando una procedura iterativa basata sul metodo di Newton per determinare* $\sqrt[n]{\alpha}$*, per un assegnato* $\alpha > 0$.

Soluzione.
Per argomentazioni molto simili a quelle viste nel precedente esercizio, non sceglieremo una funzione che utilizzi esplicitamente la radice ricercata. Utilizzeremo invece la funzione $f(x) = x^n - \alpha$, che si annulla sempre per $x = \sqrt[n]{\alpha}$. La derivata di questa funzione vale $f'(x) = nx^{n-1}$.

Andando a sostituire nella formula generica del metodo di Newton otteniamo

$$\begin{aligned} x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} = x_i - \frac{x_i^n - \alpha}{n x_i^{n-1}} = \\ = \frac{n x_i^n - x_i^n + \alpha}{n x_i^{n-1}} = \frac{(n-1)x_i^n + \alpha}{n x_i^{n-1}} = \\ = \frac{1}{n}\left((n-1)x_i + \frac{\alpha}{x_i^{n-1}}\right), \quad i = 0, 1, 2, \ldots, \end{aligned}$$

che rappresenta l'espressione funzionale cercata.
Una possibile implementazione in MATLAB di questo metodo iterativo può essere la seguente:

Codice 2.8: Metodo di Newton per il calcolo di $\sqrt[n]{\alpha}$.

```
% newtonNrt(n, alpha, x0, imax, tolx)
% Metodo di Newton ottimizzato per l'approssimazione della radice
% n-esima.
%
% Input:
%   -n: l'indice della radice;
%   -alpha: l'argomento della radice;
%   -x0: l'approssimazione iniziale;
%   -imax: il numero massimo di iterazioni;
%   -tolx: la tolleranza desiderata.
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:19 CET

function [] = newtonNrt(n, alpha, x0, imax, tolx)
    format long e
    disp('x0 ='), disp(x0)
    x = ((n-1)*x0+alpha/(x0^(n-1)))/n;
    disp('x1 ='), disp(x)
    i = 1;
    while( i<imax ) && ( abs(x-x0)>tolx )
       i = i+1;
       x0 = x;
       x = ((n-1)*x0+alpha/(x0^(n-1)))/n;
       str = sprintf('x%d =', i);
       disp(str), disp(x)
    end
    if abs(x-x0)>tolx
        disp('NON CONVERGE entro la tolleranza ed il numero di passi
            richiesti')
    else
        disp('CONVERGE entro la tolleranza ed il numero di passi
            richiesti')
    end
```

Esercizio 2.3. *In analogia con quanto visto nell'Esercizio 2.1, definire una procedura iterativa basata sul metodo delle secanti per determinare* $\sqrt{\alpha}$. *Confrontare con l'Esercizio 1.4.*

Soluzione.
Come precedentemente visto nell'Esercizio 2.1 si utilizzerà la funzione $f(x) = x^2 - \alpha$, che si annulla in $x = \pm\sqrt{\alpha}$.
Con questa funzione, l'iterazione del metodo delle secanti risulta essere

$$\begin{aligned} x_{i+1} &= \frac{f(x_i)x_{i-1} - f(x_{i-1})x_i}{f(x_i) - f(x_{i-1})} = \\ &= \frac{(x_i^2 - \alpha)x_{i-1} - (x_{i-1}^2 - \alpha)x_i}{x_i^2 - \alpha - x_{i-1}^2 + \alpha} = \\ &= \frac{x_i^2 x_{i-1} - \alpha x_{i-1} - x_{i-1}^2 x_i + \alpha x_i}{x_i^2 - x_{i-1}^2} = \\ &= \frac{x_i x_{i-1}(x_i - x_{i-1}) + \alpha(x_i - x_{i-1})}{(x_i - x_{i-1})(x_i + x_{i-1})} = \\ &= \frac{(x_i - x_{i-1})(x_i x_{i-1} + \alpha)}{(x_i - x_{i-1})(x_i + x_{i-1})} = \\ &= \frac{x_i x_{i-1} + \alpha}{x_i + x_{i-1}}, \quad i = 0, 1, 2, \ldots. \end{aligned}$$

Possiamo allora definire la seguente implementazione in MATLAB del metodo delle secanti per il calcolo di $\sqrt{\alpha}$:

Codice 2.9: Metodo delle secanti per il calcolo di $\sqrt{\alpha}$.

```
% secantiSqrt(alpha, x0, imax, tolx)
% Metodo delle secanti ottimizzato per l'approssimazione della radice
% quadrata.
%
% Input:
%   -alpha: l'argomento della radice quadrata;
%   -x0: l'approssimazione iniziale;
%   -imax: il numero massimo di iterazioni;
%   -tolx: la tolleranza desiderata.
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:24 CET

function [] = secantiSqrt(alpha, x0, imax, tolx)
    format long e
    disp('x0 ='), disp(x0)
    x = (x0+alpha/x0)/2;
    disp('x1 ='), disp(x)
    i = 1;
    while ( i<imax ) && ( abs(x-x0)>tolx )
        i = i+1;
        x1 = (x*x0 + alpha)/(x + x0);
```

```
        x0 = x;
        x = x1;
        str = sprintf('x%d =', i);
        disp(str), disp(x)
    end
    if abs(x-x0)>tolx
        disp('NON CONVERGE entro la tolleranza ed il numero di passi
            richiesti')
    else
        disp('CONVERGE entro la tolleranza ed il numero di passi
            richiesti')
    end
```

Eseguendo il metodo con $\alpha = 2$ ed $x_0 = 2$, si hanno le seguenti approssimazioni (applicando un passo del metodo di Newton per calcolare x_1):

i	x_i
0	2
1	1.5
2	1.428571428571429...
3	1.414634146341463...
4	1.414215686274510...
5	1.414213562688869...
6	1.414213562373095...
7	1.414213562373095...

Comparando questi valori con quelli visti nella tabella presentata nell'Esercizio 1.4 si può vedere che dalla sesta iterazione in poi si ha un errore commesso di convergenza sull'approssimazione minore di 10^{-12}. Inoltre si può notare come la convergenza superlineare sia leggermente più lenta rispetto alla convergenza quadratica del metodo di Newton visto nell'Esercizio 2.1.

Riferimenti MATLAB
Codice 2.11 (pagina 59)

Esercizio 2.4. *Discutere la convergenza del metodo di Newton, applicato per determinare le radici dell'equazione*

$$x^3 - 5x = 0,$$

in funzione della scelta del punto iniziale x_0.

Soluzione.
La funzione in questione è $f(x) = x^3 - 5x$ e la sua derivata prima è $f'(x) = 3x^2 - 5$. Quindi, applicando il metodo di Newton, si ottiene il seguente metodo iterativo:

$$\Phi(x) = x - \frac{f(x)}{f'(x)} = x - \frac{x^3 - 5x}{3x^2 - 5} = \frac{3x^3 - 5x - x^3 + 5x}{3x^2 - 5} = \frac{2x^3}{3x^2 - 5},$$

il quale risulta definito per ogni x escluso per $x = \pm\sqrt{\frac{5}{3}}$, dove si annulla il denominatore. Quindi sicuramente non converge per $x = \pm\sqrt{\frac{5}{3}}$.
Vediamo adesso se ci sono punti che presentano un comportamento ciclico del tipo $\Phi(x) = -x$:

$$\begin{aligned}\Phi(x) = \frac{2x^3}{3x^2-5} &= -x\\ 2x^3 &= 5x - 3x^3\\ x(5x^2-5) &= 0\\ x(x-1)(x+1) &= 0\\ x = 0 \vee x &= \pm 1.\end{aligned}$$

Per $x = 0$ il metodo converge, in quanto 0 è radice dell'equazione; per $x = 1$ invece viene prodotta la successione $-1, 1-1, 1, \ldots$ (per $x = -1$ si ha la stessa successione ma con i segni invertiti). Quindi il metodo non converge scegliendo $x = \pm 1$ come punto iniziale.
In sintesi, facendo alcuni test sulla convergenza del metodo, si deduce che il metodo converge nei seguenti intervalli:

$$\begin{aligned}\left(-\infty, -\sqrt{\frac{5}{3}}\right) &\longrightarrow -\sqrt{5}\\ \left(-\sqrt{\frac{5}{3}}, -1\right) &\longrightarrow \sqrt{5}\\ (-1, 1) &\longrightarrow 0\\ \left(1, \sqrt{\frac{5}{3}}\right) &\longrightarrow -\sqrt{5}\\ \left(\sqrt{\frac{5}{3}}, +\infty\right) &\longrightarrow \sqrt{5}.\end{aligned}$$

Riferimenti MATLAB
Codice 2.12 (pagina 59)

Esercizio 2.5. *Comparare il metodo di Newton (2.4), il metodo di Newton modificato (2.5) ed il metodo di accelerazione di Aitken (2.6), per approssimare gli zeri delle funzioni*

$$f_1(x) = (x-1)^{10}, \qquad f_2(x) = (x-1)^{10}e^x,$$

per valori decrescenti della tolleranza tol_x. *Utilizzare, in tutti i casi, il punto iniziale* $x_0 = 10$.

Soluzione.
Per quanto riguarda la prima funzione $f_1(x) = (x-1)^10$ si ha che l'unico zero presente è $x^* = 1$ di molteplicità $m = 10$, infatti $f_1(1) = f_1'(1) = \cdots = f_1^{(9)}(1) = 0$ e $f_1^{10}(1) = 10! \neq 0$.
Nella seguente tabella si sono riportati i risultati ottenuti dall'esecuzione dei metodi di Newton, di Newton modificato e di Aitken per l'approssimazione di tale radice per valori decrescenti della tolleranza tol_x ($\tilde{x}$ indica l'approssimazione calcolata dal metodo) ed $x_0 = 10$ come approssimazione iniziale:

$f_1(x) = (x-1)^{10}$			
tol_x	Newton	Newton modificato	Aitken
10^{-1}	$\tilde{x} = 1.8863 \quad i = 22$	$\tilde{x} = 1 \quad i = 1$	$\tilde{x} = 1 \quad i = 2$
10^{-2}	$\tilde{x} = 1.0873 \quad i = 44$	$\tilde{x} = 1 \quad i = 1$	$\tilde{x} = 1 \quad i = 2$
10^{-3}	$\tilde{x} = 1.0086 \quad i = 66$	$\tilde{x} = 1 \quad i = 1$	$\tilde{x} = 1 \quad i = 2$
10^{-4}	$\tilde{x} = 1.0008 \quad i = 88$	$\tilde{x} = 1 \quad i = 1$	$\tilde{x} = 1 \quad i = 2$
10^{-5}	$\tilde{x} = 1.0001 \quad i = 110$	$\tilde{x} = 1 \quad i = 1$	$\tilde{x} = 1 \quad i = 2$
10^{-6}	$\tilde{x} = 1 \quad i = 132$	$\tilde{x} = 1 \quad i = 1$	$\tilde{x} = 1 \quad i = 2$
10^{-7}	$\tilde{x} = 1 \quad i = 153$	$\tilde{x} = 1 \quad i = 1$	$\tilde{x} = 1 \quad i = 2$
10^{-8}	$\tilde{x} = 1 \quad i = 175$	$\tilde{x} = 1 \quad i = 1$	$\tilde{x} = 1 \quad i = 2$
10^{-9}	$\tilde{x} = 1 \quad i = 197$	$\tilde{x} = 1 \quad i = 1$	$\tilde{x} = 1 \quad i = 2$
10^{-10}	$\tilde{x} = 1 \quad i = 219$	$\tilde{x} = 1 \quad i = 1$	$\tilde{x} = 1 \quad i = 2$
10^{-11}	$\tilde{x} = 1 \quad i = 241$	$\tilde{x} = 1 \quad i = 1$	$\tilde{x} = 1 \quad i = 2$
10^{-12}	$\tilde{x} = 1 \quad i = 263$	$\tilde{x} = 1 \quad i = 1$	$\tilde{x} = 1 \quad i = 2$
10^{-13}	$\tilde{x} = 1 \quad i = 285$	$\tilde{x} = 1 \quad i = 1$	$\tilde{x} = 1 \quad i = 2$
10^{-14}	$\tilde{x} = 1 \quad i = 306$	$\tilde{x} = 1 \quad i = 1$	$\tilde{x} = 1 \quad i = 2$
10^{-15}	$\tilde{x} = 1 \quad i = 329$	$\tilde{x} = 1 \quad i = 1$	$\tilde{x} = 1 \quad i = 2$

Osservando i risultati ottenuti si può vedere come il metodo di Newton perda la convergenza quadratica il presenza di radici multiple ed inoltre che, in questi casi, i metodi di Newton modificato e di accelerazione di Aitken rispondono molto bene alla presenza di radici multiple.

Per quanto riguarda la seconda funzione $f_2(x) = (x-1)^{10}e^x$ si ha, come per la prima, che l'unico zero è $x^* = 1$ di molteplicità $m = 10$.
Di seguito i risultati ottenuti:

$f_2(x) = (x-1)^{10}e^x$						
tol_x	Newton		Newton modificato		Aitken	
10^{-1}	$\tilde{x} = 1.9216$	$i = 30$	$\tilde{x} = 1$	$i = 5$	$\tilde{x} = 0.9994$	$i = 5$
10^{-2}	$\tilde{x} = 1.0896$	$i = 53$	$\tilde{x} = 1$	$i = 5$	$\tilde{x} = 1$	$i = 6$
10^{-3}	$\tilde{x} = 1.0089$	$i = 75$	$\tilde{x} = 1$	$i = 6$	$\tilde{x} = 1$	$i = 6$
10^{-4}	$\tilde{x} = 1.0009$	$i = 97$	$\tilde{x} = 1$	$i = 6$	$\tilde{x} = 1$	$i = 7$
10^{-5}	$\tilde{x} = 1.0001$	$i = 119$	$\tilde{x} = 1$	$i = 6$	$\tilde{x} = 1$	$i = 7$
10^{-6}	$\tilde{x} = 1$	$i = 141$	$\tilde{x} = 1$	$i = 6$	$\tilde{x} = 1$	$i = 7$
10^{-7}	$\tilde{x} = 1$	$i = 163$	$\tilde{x} = 1$	$i = 7$	$\tilde{x} = 1$	$i = 7$
10^{-8}	$\tilde{x} = 1$	$i = 185$	$\tilde{x} = 1$	$i = 7$	$\tilde{x} = 1$	$i = 7$
10^{-9}	$\tilde{x} = 1$	$i = 207$	$\tilde{x} = 1$	$i = 7$	$\tilde{x} = 1$	$i = 7$
10^{-10}	$\tilde{x} = 1$	$i = 228$	$\tilde{x} = 1$	$i = 7$	$\tilde{x} = 1$	$i = 7$
10^{-11}	$\tilde{x} = 1$	$i = 250$	$\tilde{x} = 1$	$i = 7$	$\tilde{x} = 1$	$i = 7$
10^{-12}	$\tilde{x} = 1$	$i = 272$	$\tilde{x} = 1$	$i = 7$	$\tilde{x} = 1$	$i = 7$
10^{-13}	$\tilde{x} = 1$	$i = 294$	$\tilde{x} = 1$	$i = 7$	$\tilde{x} = 1$	$i = 7$
10^{-14}	$\tilde{x} = 1$	$i = 316$	$\tilde{x} = 1$	$i = 7$	$\tilde{x} = 1$	$i = 7$
10^{-15}	$\tilde{x} = 1$	$i = 337$	$\tilde{x} = 1$	$i = 7$	$\tilde{x} = 1$	$i = 7$

I risultati sono molto simili ai precedenti. Si nota soltanto che il fattore e^x ha leggermente rallentato i metodi di Newton modificato e di Aitken.

Riferimenti MATLAB
Codice 2.13 (pagina 60)

Esercizio 2.6. *È possibile, nel caso delle funzioni del precedente esercizio, utilizzare il metodo di bisezione per determinarne lo zero?*

Soluzione.
No non è possibile in quanto non si verifica una delle ipotesi del Teorema degli Zeri, ovvero che esistano due punti a e b tali che $f(a)f(b) < 0$ per poter formare un intervallo di confidenza. Infatti entrambe le funzioni risultano essere positive $\forall x \in \mathbb{R}$: il fattore $(x-1)^{10}$ è sempre positivo in quanto è elevato ad una potenza pari, mentre il fattore e^x è sempre positivo per definizione.

Esercizio 2.7. *Costruire una tabella in cui si comparano, a partire dallo stesso punto iniziale $x_0 = 0$, e per valori decrescenti della tolleranza tol_x, il numero di iterazioni richieste per la convergenza dei metodi di Newton, corde e secanti, utilizzati per determinare lo zero della funzione*

$$f(x) = x - \cos x$$

.

Soluzione.
L'equazione $f(x) = 0$ presenta una sola radice semplice in $x^* \approx 0.739085$.
Di seguito la tabella con i dati ottenuti dall'esecuzione dei tre metodi con tolleranza decrescente ed approssimazione iniziale $x_0 = 0$:

$f(x) = x - \cos(x)$			
tol_x	Newton	Corde	Secanti
10^{-1}	$\tilde{x} = 0.7391 \quad i = 3$	$\tilde{x} = 0.7014 \quad i = 6$	$\tilde{x} = 0.7363 \quad i = 3$
10^{-2}	$\tilde{x} = 0.7391 \quad i = 4$	$\tilde{x} = 0.7356 \quad i = 12$	$\tilde{x} = 0.7391 \quad i = 4$
10^{-3}	$\tilde{x} = 0.7391 \quad i = 4$	$\tilde{x} = 0.7388 \quad i = 18$	$\tilde{x} = 0.7391 \quad i = 5$
10^{-4}	$\tilde{x} = 0.7391 \quad i = 4$	$\tilde{x} = 0.7391 \quad i = 24$	$\tilde{x} = 0.7391 \quad i = 5$
10^{-5}	$\tilde{x} = 0.7391 \quad i = 5$	$\tilde{x} = 0.7391 \quad i = 30$	$\tilde{x} = 0.7391 \quad i = 6$
10^{-6}	$\tilde{x} = 0.7391 \quad i = 5$	$\tilde{x} = 0.7391 \quad i = 35$	$\tilde{x} = 0.7391 \quad i = 6$
10^{-7}	$\tilde{x} = 0.7391 \quad i = 5$	$\tilde{x} = 0.7391 \quad i = 41$	$\tilde{x} = 0.7391 \quad i = 6$
10^{-8}	$\tilde{x} = 0.7391 \quad i = 5$	$\tilde{x} = 0.7391 \quad i = 47$	$\tilde{x} = 0.7391 \quad i = 7$
10^{-9}	$\tilde{x} = 0.7391 \quad i = 5$	$\tilde{x} = 0.7391 \quad i = 53$	$\tilde{x} = 0.7391 \quad i = 7$
10^{-10}	$\tilde{x} = 0.7391 \quad i = 6$	$\tilde{x} = 0.7391 \quad i = 59$	$\tilde{x} = 0.7391 \quad i = 7$
10^{-11}	$\tilde{x} = 0.7391 \quad i = 6$	$\tilde{x} = 0.7391 \quad i = 65$	$\tilde{x} = 0.7391 \quad i = 7$
10^{-12}	$\tilde{x} = 0.7391 \quad i = 6$	$\tilde{x} = 0.7391 \quad i = 70$	$\tilde{x} = 0.7391 \quad i = 7$
10^{-13}	$\tilde{x} = 0.7391 \quad i = 6$	$\tilde{x} = 0.7391 \quad i = 76$	$\tilde{x} = 0.7391 \quad i = 8$
10^{-14}	$\tilde{x} = 0.7391 \quad i = 6$	$\tilde{x} = 0.7391 \quad i = 82$	$\tilde{x} = 0.7391 \quad i = 8$
10^{-15}	$\tilde{x} = 0.7391 \quad i = 6$	$\tilde{x} = 0.7391 \quad i = 88$	$\tilde{x} = 0.7391 \quad i = 8$

Si vede da questi risultati che i metodi di Newton e delle secanti convergono molto velocemente alla soluzione, mentre il metodo delle corde, seppur convergendo, richiede molti più passi d'iterazione. Tuttavia, osservando il tempo d'esecuzione impiegato dai tre metodi per eseguire un singolo step, si deduce che i metodi quasi-Newton (corde e secanti) hanno un tempo di esecuzione medio per step inferiore a quello del metodo di Newton: infatti, in media, un passo d'iterazione del metodo delle secanti dura circa $1/2$ rispetto a quello di Newton e quello delle corde $1/4$. Quindi, in questo caso, il metodo più efficiente sembra essere quello delle secanti, che combina un'alta convergenza con un basso tempo di esecuzione.

Riferimenti MATLAB
Codice 2.14 (pagina 60)

Esercizio 2.8. *Completare i confronti del precedente esercizio inserendo quelli con il metodo di bisezione, con intervallo di confidenza iniziale* $[0, 1]$.

Soluzione.
Vediamo i risultati dell'esecuzione del metodo di bisezione utilizzando come intervallo di confidenza iniziale $[0, 1]$:

$f(x) = x - \cos(x)$		
tol_x	Approssimazione	Iterazioni
10^{-1}	$\tilde{x} = 0.75$	$i = 1$
10^{-2}	$\tilde{x} = 0.7344$	$i = 5$
10^{-3}	$\tilde{x} = 0.7383$	$i = 8$
10^{-4}	$\tilde{x} = 0.7390$	$i = 11$
10^{-5}	$\tilde{x} = 0.7391$	$i = 15$
10^{-6}	$\tilde{x} = 0.7391$	$i = 18$
10^{-7}	$\tilde{x} = 0.7391$	$i = 19$
10^{-8}	$\tilde{x} = 0.7391$	$i = 23$
10^{-9}	$\tilde{x} = 0.7391$	$i = 27$
10^{-10}	$\tilde{x} = 0.7391$	$i = 29$
10^{-11}	$\tilde{x} = 0.7391$	$i = 34$
10^{-12}	$\tilde{x} = 0.7391$	$i = 38$
10^{-13}	$\tilde{x} = 0.7391$	$i = 40$
10^{-14}	$\tilde{x} = 0.7391$	$i = 43$
10^{-15}	$\tilde{x} = 0.7391$	$i = 48$

Si deduce da questi risultati che il metodo di bisezione, in questo caso, converge più lentamente dei metodi di Newton e delle secanti ma, allo stesso tempo, più velocemente del metodo delle corde. Tuttavia, rispetto ai metodi di Newton e delle secanti, il metodo di bisezione presenta un minor tempo medio di esecuzione per step, ovvero un minor costo computazionale.

Riferimenti MATLAB
Codice 2.15 (pagina 61)

Esercizio 2.9. *Quali controlli introdurreste, negli algoritmi del metodo di Newton, del metodo di accelerazione di Aitken e del metodo delle secanti, al fine di rendere più "robuste" le corrispondenti iterazioni?*

Soluzione.
Oltre al controllo sul numero massimo di iterazioni eseguite e sull'errore commesso (rispetto ad una tolleranza fissata) si può inserire un controllo sulle divisioni per zero: su $f'(x_i)$ per i metodi di Newton e di Aitken, su $(x_{i+1} - 2x_i + x_{i-1})$ per il metodo di Aitken e su $(f(x_i) - f(x_{i-1}))$ per il metodo delle secanti.

Codice degli esercizi

Codice 2.10: Esercizio 2.1.

```
% Esercizio 2.1
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:10 CET

newtonSqrt(2, 2, 100, eps);
```

Codice 2.11: Esercizio 2.3.

```
% Esercizio 2.3
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:10 CET

secantiSqrt(2, 2, 100, eps);
```

Codice 2.12: Esercizio 2.4.

```
% Esercizio 2.4
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:10 CET

f = inline('x^3-5*x');
f1 = inline('3*x^2-5');

disp('x minore di -sqrt(5/3):')
newton(f, f1, -sqrt(5/3)-eps, 100, eps, 0, 0, 0);
disp('Premere un tasto per continuare'), pause

disp('x tra -sqrt(5/3) e -1:')
newton(f, f1, (-sqrt(5/3)-1)/2, 100, eps, 0, 0, 0);
disp('Premere un tasto per continuare'), pause

disp('x tra -1 e 0:')
newton(f, f1, -1/2, 100, eps, 0, 0, 0);
disp('Premere un tasto per continuare'), pause

disp('x tra 1 e 0:')
newton(f, f1, 1/2, 100, eps, 0, 0, 0);
disp('Premere un tasto per continuare'), pause

disp('x tra 1 e sqrt(5/3):')
newton(f, f1, (1+sqrt(5/3))/2, 100, eps, 0, 0, 0);
disp('Premere un tasto per continuare'), pause
```

```
28
disp('x maggiore di sqrt(5/3):')
30 newton(f, f1, sqrt(5/3)+eps, 100, eps, 0, 0, 0);
```

Codice 2.13: Esercizio 2.5.

```
% Esercizio 2.5
2 %
% Autore: Tommaso Papini,
4 % Ultima modifica: 4 Novembre 2012, 11:10 CET

6 f = inline('(x-1)^10');
f1 = inline('10*(x-1)^9');
8 tolx = 10^-1;
disp('f1(x)=(x-1)^10')
10 while tolx>eps
    disp(' '), str = sprintf('Tolleranza %d', tolx); disp(str)
12  disp('Newton: '), newton(f, f1, 10, 1000, tolx, 0, 0, 0);
    disp('Newton modificato: '), newtonMod(f, f1, 10, 10, 1000, tolx,
        0, 0, 0);
14  disp('Aitken: '), aitken(f, f1, 10, 1000, tolx, 0, 0, 0);
    tolx = tolx/10;
16  disp('Premere un tasto per continuare'), pause
end
18

f = inline('((x-1)^10)*exp(x)');
20 f1 = inline('((x-1)^9)*(x+9)*exp(x)');
tolx = 10^-1;
22 disp(' '), disp('f2(x)=((x-1)^10)*exp(x)')
while tolx>eps
24  disp(' '), str = sprintf('Tolleranza %d', tolx); disp(str)
    disp('Newton: '), newton(f, f1, 10, 1000, tolx, 0, 0, 0);
26  disp('Newton modificato: '), newtonMod(f, f1, 10, 10, 1000, tolx,
        0, 0, 0);
    disp('Aitken: '), aitken(f, f1, 10, 1000, tolx, 0, 0, 0);
28  tolx = tolx/10;
    disp('Premere un tasto per continuare'), pause
30 end
```

Codice 2.14: Esercizio 2.7.

```
% Esercizio 2.7
2 %
% Autore: Tommaso Papini,
4 % Ultima modifica: 4 Novembre 2012, 11:10 CET
```

```
f = inline('x-cos(x)');
f1 = inline('1+sin(x)');
tolx = 10^-1;
while tolx>eps
    disp(' '), str = sprintf('Tolleranza %d', tolx); disp(str)
    disp('Newton: '), newton(f, f1, 0, 1000, tolx, 0, 0, 0);
    disp('Corde: '), corde(f, f1, 0, 1000, tolx, 0, 0, 0);
    disp('Secanti: '), secanti(f, f1, 0, 1000, tolx, 0, 0, 0);
    tolx = tolx/10;
    disp('Premere un tasto per continuare'), pause
end
```

Codice 2.15: Esercizio 2.8.

```
% Esercizio 2.8
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:10 CET

f = inline('x-cos(x)');
f1 = inline('1+sin(x)');
tolx = 10^-1;
while tolx>eps
    disp(' '), str = sprintf('Tolleranza %d', tolx); disp(str)
    bisezione(f, 0, 1, tolx, 0, 0, 0);
    tolx = tolx/10;
    disp('Premere un tasto per continuare'), pause
end
```

Capitolo 3

Sistemi lineari e nonlineari

Indice

Questo capitolo tratta della risoluzione di **sistemi di equazioni lineari**, o semplicemente **sistemi lineari**, ovvero sistemi del tipo

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases},$$

esprimibile anche nella forma matriciale

$$A\underline{x} = \underline{b}, \tag{3.1}$$

$$\begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ \vdots \\ \vdots \\ b_m \end{pmatrix}$$

dove $A = (a_{ij}) \in \mathbb{R}^{m \times n}$ è la *matrice dei coefficienti*, $\underline{x} = (x_j) \in \mathbb{R}^n$ è il *vettore delle incognite* e $\underline{b} = (b_i) \in \mathbb{R}^m$ è il *vettore dei termini noti*. Chiaramente, A e $\underline{b}$ rappresentano i dati del problema, mentre $\underline{x}$ rappresenta la soluzione da calcolare. In questo capitolo supporremo che $m \geq n$ e che $rank(A) = n$, ovvero che la matrice A abbia rango massimo (ricordando che il *rango* di una matrice è il massimo numero di righe, o colonne, linearmente indipendenti). In particolare studiamo un primo caso, più semplice in cui $m = n$, ovvero la matrice A è quadrata e il sistema è composto da n equazioni in n incognite. Essendo allora la matrice A *quadrata* e di *rango massimo*, si dice che A è **nonsingolare**. Un'importante proprietà delle matrici nonsingolari è che esiste una ed una sola loro inversa, quindi esiste ed è unica la matrice inversa A^{-1}, ovvero esiste ed è unica la soluzione del sistema (3.1):

$$\underline{x} = A^{-1}\underline{b}.$$

Questa, seppur semplice, soluzione risulta tuttavia, nella maggior parte dei casi, inefficiente rispetto ad altri metodi più intelligenti che vedremo in seguito.

3.1 Sistemi lineari: casi semplici

Se la matrice A possiede particolari caratteristiche, allora si potrà risolvere il sistema (3.1) in modo relativamente semplice. In particolare si hanno tre casi in cui la risoluzione risulta molto semplice: quando A è *diagonale*, *triangolare* o *ortogonale*. Successivamente si utilizzeranno opportuni *metodi di fattorizzazione* per ricondurre tutti gli altri casi ad una combinazione dei tre casi semplici appena elencati.

Matrici diagonali

Una matrice si dice **diagonale** se ha tutti gli elementi, esclusi quelli sulla *diagonale principale*, nulli, ovvero una matrice del tipo

$$A = \begin{pmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & a_{nn} \end{pmatrix}.$$

Quindi il sistema corrispondente sarà della forma

$$\begin{cases} a_{11}x_1 = b_1 \\ a_{22}x_2 = b_2 \\ \vdots \\ a_{nn}x_n = b_n \end{cases},$$

e gli elementi del vettore soluzione sono facilmente calcolati come

$$x_i = \frac{b_i}{a_{ii}}, \quad i = 1, \dots n.$$

Osserviamo che, essendo A nonsingolare (e quindi di rango massimo $= n$), $a_{ii} \neq 0$ per $i = 1 \dots n$. Quindi le operazioni appena descritte sono *ben definite*.
Il costo computazionale per calcolare tutte le componenti della soluzione risulta essere n **flop**, in quanto il metodo risolutivo consiste in n divisioni. Anche l'occupazione di memoria risulta essere **lineare**, in quanto gli elementi significativi di A, ovvero gli elementi sulla diagonale principale, possono essere memorizzati in un vettore di lunghezza n.

Matrici triangolari

In questo tipo di matrice nonsingolare gli elementi significativi si trovano lungo la diagonale principale e in una porzione triangolare della matrice. Se gli elementi si trovano nella porzione triangolare strettamente inferiore, si parla di **matrice triangolare inferiore**, altrimenti di **matrice triangolare superiore**:

- A triangolare inferiore: $a_{ij} = 0$ se $i < j$,

$$A = \begin{pmatrix} a_{11} & 0 & \dots & 0 \\ \vdots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & 0 \\ a_{n1} & \dots & \dots & a_{nn} \end{pmatrix};$$

- A triangolare superiore: $a_{ij} = 0$ se $i > j$,

$$A = \begin{pmatrix} a_{11} & \dots & \dots & a_{1n} \\ 0 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \dots & 0 & a_{nn} \end{pmatrix}.$$

Per quanto riguarda il caso in cui A sia *triangolare inferiore*, il sistema (3.1) diventa

$$\begin{cases} a_{11}x_1 & = b_1 \\ a_{21}x_1 + a_{22}x_2 & = b_2 \\ \vdots & \vdots \ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \dots + a_{nn}x_n & = b_n \end{cases},$$

e gli elementi del vettore soluzione vengono calcolati tramite *sostituzioni successive in avanti*, ovvero viene innanzitutto calcolato $x_1 = {}^{b_1}/_{a_{11}}$, quindi viene calcolato x_2 utilizzando il valore di x_1, e così via:

$$\begin{aligned} x_1 &= \frac{b_1}{a_{11}} \\ x_2 &= \frac{b_3 - a_{21}x_1}{a_{22}} \\ &\vdots \\ x_n &= \frac{b_n - \sum_{j=1}^{n-1} a_{nj}x_j}{a_{nn}}. \end{aligned}$$

Quindi in generale si calcola un generico x_i come $x_i = \dfrac{b_i - \sum_{j=i}^{i-1} a_{ij}x_j}{a_{ii}}$, ovvero utilizzando i valori degli $i-1$ elementi della soluzione precedenti x_i, già calcolati. Implementando in MATLAB la risoluzione di sistemi lineari triangolari inferiori, si può scegliere di effettuare gli accessi agli elementi della matrice dei coefficienti per riga (Codice 3.1) o per colonna (Codice 3.2).

Codice 3.1: Risoluzione di sistemi lineari triangolari inferiori con accesso per riga.

```
% b = triangolareInfRiga(A, b)
% Metodo per la risoluzione di sistemi lineari triangolari inferiori,
% accedendo agli elementi per riga.
%
% Input:
%   -A: la matrice dei coefficienti;
%   -b: vettore dei termini noti.
% Output:
%   -b: vettore delle soluzioni.
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:26 CET

function [b] = triangolareInfRiga(A, b)
    for i=1:length(A)
        for j=1:i-1
            b(i)=b(i)-A(i,j)*b(j);
        end
        if A(i,i)==0
            error('La matrice è singolare!')
        else
            b(i)=b(i)/A(i,i);
        end
    end
```

Codice 3.2: Risoluzione di sistemi lineari triangolari inferiori con accesso per colonna.

```
% b = triangolareInfCol(A, b)
% Metodo per la risoluzione di sistemi lineari triangolari inferiori,
% accedendo agli elementi per colonna.
% Input:
%   -A: la matrice dei coefficienti;
%   -b: vettore dei termini noti.
% Output:
%   -b: vettore delle soluzioni.
%
% Autore: Tommaso Papini,
% Ultima modifica: 6 Ottobre 2012, 10:58 CEST

function [b] = triangolareInfCol(A, b)
    for j=1:length(A)
        if A(j,j)==0
            error('La matrice è singolare!')
        else
            b(j)=b(j)/A(j,j);
        end
        for i=j+1:length(A)
            b(i)=b(i)-A(i,j)*b(j);
        end
    end
```

Nel caso, invece, in cui A sia triangolare superiore, il sistema assume la forma

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ \vdots \quad \vdots \vdots & \\ a_{nn}x_n &= b_n \end{cases},$$

quindi si potrà applicare l'algoritmo visto per le matrici triangolari inferiori ma utilizzandolo al contrario (*sostituzioni successive all'indietro*), ovvero partendo col calcolare $x_n = {}^{b_n}/_{a_{nn}}$, poi x_{n-1}, e così via:

$$\begin{aligned} x_n &= \frac{b_n}{a_{nn}} \\ x_{n-1} &= \frac{b_{n-1} - a_{n-1}x_n}{a_{n-1,n-1}} \\ &\vdots \\ x_1 &= \frac{b_1 - \sum_{j=n}^{2} a_{1j}x_j}{a_{11}}. \end{aligned}$$

La componente i-esima viene quindi calcolata come $x_i = \dfrac{b_i - \sum_{j=n}^{i+1} a_{ij}x_j}{a_{ii}}$.
Come per il caso precedente, possiamo definire un metodo risolutivo con accesso agli elementi della matrice A per riga (Codice 3.3) ed uno con accesso per colonna (Codice 3.4).

Codice 3.3: Risoluzione di sistemi lineari triangolari superiori con accesso per riga.

```
% b = triangolareSupRiga(A, b)
% Metodo per la risoluzione di sistemi lineari triangolari superiori,
% accedendo agli elementi per riga.
%
% Input:
%   -A: la matrice dei coefficienti;
%   -b: vettore dei termini noti.
% Output:
%   -b: vettore delle soluzioni.
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:27 CET

function [b] = triangolareSupRiga(A, b)
    for i=length(A):-1:1
        for j=i+1:length(A)
            b(i)=b(i)-A(i,j)*b(j);
        end
        if A(i,i)==0
            error('La matrice è singolare!')
        else
            b(i)=b(i)/A(i,i);
        end
    end
```

Codice 3.4: Risoluzione di sistemi lineari triangolari superiori con accesso per colonna.

```
% b = triangolareSupCol(A, b)
% Metodo per la risoluzione di sistemi lineari triangolari superiori,
% accedendo agli elementi per colonna.
%
% Input:
%   -A: la matrice dei coefficienti;
%   -b: vettore dei termini noti.
% Output:
%   -b: vettore delle soluzioni.
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:27 CET

function [b] = triangolareSupCol(A, b)
    for j=length(A):-1:1
        if A(j,j)==0
            error('La matrice è singolare!')
        else
            b(j)=b(j)/A(j,j);
        end
        for i=1:j-1
            b(i)=b(i)-A(i,j)*b(j);
        end
    end
```

In entrambi i casi, sono preferibili i metodi che risolvono il sistema triangolare accedendo agli elementi della matrice dei coefficienti per colonna (Codici 3.2 e 3.4). Infatti MATLAB memorizza le matrici per colonna, ovvero memorizza gli elementi di una stessa colonna in posizioni contigue di memoria. In generale, in MATLAB, accedere agli elementi di una matrice per colonna risulta essere più efficiente rispetto all'accesso per riga.

Essendo A nonsingolare, sicuramente tutti gli elementi diagonali a_{ii} risultano diversi da zero, ovvero $a_{ii} \neq 0, i = 1, \dots n$, quindi le operazioni, in entrambi i casi, sono sempre *ben definite*.
Per quanto riguarda l'occupazione di memoria, in entrambi i casi si deve memorizzare soltanto la porzione triangolare della matrice A, ovvero

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \approx \frac{n^2}{2}$$

posizioni di memoria. Invece per il costo computazionale si ha che al primo passo è necessario 1 `flop`, al secondo 3 `flop`, al terzo 5 `flop`, ..., al passo n-esimo $2n-1$ `flop`, per un totale di

$$\sum_{i=1}^{n}(2i-1) = n^2 \quad \textbf{flop}.$$

Matrici ortogonali

Una matrice A si dice **ortogonale** se risulta che la sua inversa e la sua trasposta coincidono, ovvero tale che $A^{-1} = A^T$. Quindi, essendo il calcolo della trasposta molto veloce, il sistema (3.1) è immediatamente risolvibile come

$$\underline{x} = A^{-1}\underline{b} = A^T\underline{b}.$$

Allora il costo computazionale corrisponde al costo di un *prodotto matrice-vettore*, ovvero $2mn$ *flop* nel caso generale, $2n^2$ **flop** essendo A nonsingolare (e quindi quadrata). Invece, per quanto riguarda l'occupazione di memoria, si ha che saranno necessarie circa n^2 posizioni di memoria, in quanto tutti gli elementi della matrice A possono essere elementi significativi.

Metodi di fattorizzazione

Data la semplicità e l'efficienza della risoluzione dei tre casi appena esaminati, per poter risolvere tutti gli altri tipi di sistema lineare si ricorre ad opportuni **metodi di fattorizzazione** che riconducano la matrice A ad un insieme di matrici nonsingolari che siano *diagonali*, *triangolari* od *ortogonali*. In particolare si cerca di ottenere una fattorizzazione per la matrice A del tipo

$$A = F_1 F_2 \dots F_k,$$

per un opportuno k, con i fattori $F_i \in \mathbb{R}^{n\times n}$, $i = 1, \dots, k$, matrici nonsingolari ed appartenenti ad una delle categorie precedentemente esaminate (diagonale, triangolare, ortogonale).
Una volta ottenuta tale fattorizzazione, si dovranno quindi risolvere k sistemi lineari "semplici", generando una successione di *soluzioni intermedie*, $\underline{x_i} \in \mathbb{R}^n$, $i = 1, \dots, k$, con $\underline{x} = \underline{x_k}$. Chiamando il fattore $F_2 \dots F_k \underline{x} \equiv \underline{x_1}$, ovvero la prima soluzione intermedia, si ha che il primo sistema lineare da risolvere sarà

$$\begin{aligned} &A\underline{x} = \underline{b}, \\ &F_1 \underbrace{F_2 \dots F_k \underline{x}}_{\underline{x_1}} = \underline{b}, \\ &F_1 \underline{x_1} = \underline{b}. \end{aligned}$$

Analogamente la seconda soluzione intermedia sarà $\underline{x_2} \equiv F_3 \dots F_k \underline{x}$ e il secondo sistema lineare da risolvere risulta essere

$$\begin{aligned} &F_2 \underbrace{F_3 \dots F_k \underline{x}}_{\underline{x_2}} = \underline{x_1}, \\ &F_2 \underline{x_2} = \underline{x_1}. \end{aligned}$$

Reiterando sino a k, si ottiene la seguente successione di sistemi lineari i quali, risolti in ordine, portano alla soluzione $\underline{x}$ finale:

$$F_1 \underline{x_1} = \underline{b}, \quad F_2 \underline{x_2} = \underline{x_1}, \quad \dots \quad F_k \underline{x_k} = \underline{x_{k-1}}, \quad \underline{x} \equiv \underline{x_k}.$$

Computazionalmente, tutti questi sistemi lineari risultano essere, per costruzione, di uno dei tipi più semplici visti precedentemente e quindi facilmente risolvibili. Inoltre, per quanto riguarda l'occupazione di memoria, si può utilizzare un solo vettore per tutte le soluzioni intermedie e per la soluzione finale in quanto per ogni sistema lineare da risolvere sarà necessario memorizzare una sola soluzione intermedia (tutte le precedenti non saranno più necessarie).

3.2 Fattorizzazione LU di una matrice

Definizione 3.1. *Se la matrice A del sistema (3.1) può essere riscritta come*

$$A = LU, \tag{3.2}$$

con $L \in \mathbb{R}^{n\times n}$ triangolare inferiore a diagonale unitaria (L infatti sta per lower) ed $U \in \mathbb{R}^{n\times n}$ triangolare superiore (U sta per upper), allora A si dice ***fattorizzabile LU****.*

Teorema 3.1 (Unicità della fattorizzazione LU)**.** *Se A è nonsingolare e la fattorizzazione LU (3.2) esiste, allora tale fattorizzazione è anche* ***unica****.*

Supponiamo adesso di avere un vettore

$$\underline{v} = (v_1, \ldots, v_n)^T \in \mathbb{R}^n,$$

del quale ne vogliamo azzerare tutte le componenti dalla $(k+1)$-esima in poi, lasciando invariate le prime k, mediante moltiplicazione a sinistra per una matrice $L \in \mathbb{R}^{n \times n}$ *triangolare inferiore a diagonale unitaria.*

Definizione 3.2. *Se $v_k \neq 0$, allora è possibile definire il* ***vettore elementare di Gauss****:*

$$\underline{g} = \frac{1}{v_k}(\underbrace{0, \ldots, 0}_{k}, v_{k+1}, \ldots, v_n)^T.$$

Definizione 3.3. *Utilizzando il vettore elementare di Gauss, si può definire la corrispondente* ***matrice elementare di Gauss****:*

$$L = I - \underline{g}\,\underline{e_k}^T,$$

*ricordando che $\underline{e_k}$ è il k-**esimo versore della base canonica**, ovvero un vettore (di lunghezza opportuna) con tutte le sue componenti a 0 tranne la k-esima che vale 1:*

$$\underline{e_k} = (\underbrace{0, \ldots, 0}_{k-1}, 1, \underbrace{0, \ldots, 0}_{n-k})^T \in \mathbb{R}^n.$$

Quindi si ha che la matrice elementare di Gauss appena definita è della forma

$$L = \begin{pmatrix} 1 & & & & & \\ & 1 & & & & \\ & & \ddots & & & \\ & & & \ddots & & \\ & & & & 1 & \\ & & & & & 1 \end{pmatrix} - \begin{matrix} \\ \\ \\ \\ k \\ \\ \\ \end{matrix} \left\{ \begin{pmatrix} 0 \\ \vdots \\ 0 \\ \frac{v_{k+1}}{v_k} \\ \vdots \\ \frac{v_n}{v_k} \end{pmatrix} (\underbrace{0, \ldots, 0}_{k-1}, 1, \underbrace{0, \ldots, 0}_{n-k}) \right.$$

$$= \begin{pmatrix} 1 & & & & & \\ & 1 & & & & \\ & & \ddots & & & \\ & & & \ddots & & \\ & & & & 1 & \\ & & & & & 1 \end{pmatrix} - \begin{matrix} \\ \\ \\ \\ \\ \\ \text{riga } k+1 \end{matrix} \overset{\text{k}}{\begin{pmatrix} 0 & \ldots & \ldots & \ldots & \ldots & 0 \\ \vdots & \ddots & & & & \vdots \\ \vdots & 0 & 0 & & & \vdots \\ \vdots & \vdots & \frac{v_{k+1}}{v_k} & \ddots & & \vdots \\ \vdots & \vdots & \vdots & 0 & \ddots & \vdots \\ 0 & 0 & \frac{v_n}{v_k} & 0 & \ldots & 0 \end{pmatrix}} =$$

$$= \begin{matrix} \\ \\ \\ \\ \\ \\ \text{riga } k+1 \end{matrix} \overset{\text{k}}{\begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & -\frac{v_{k+1}}{v_k} & \ddots & & \\ & & \vdots & & \ddots & \\ & & -\frac{v_n}{v_k} & & & 1 \end{pmatrix}} .$$

Allora risulta che

$$L\underline{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_k \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

infatti si avrà che i primi k elementi di $L\underline{v}$ saranno il risultato della moltiplicazione del solo elemento corrispondente per 1; invece per gli elementi dal $(k+1)$-esimo in poi si ha la somma tra la frazione negativa presente in $L\underline{v}$ moltiplicata per v_k (in quanto tali elementi si trovano tutti nella colonna k-esima) e l'elemento corrispondente moltiplicato per 1 (come prima, essendo la diagonale unitaria).

Quest'ultima somma, per costruzione, da sempre come risultato 0. Ad esempio per il $(k+1)$-esimo elemento si ha:

$$(L\underline{v})_{k+1} = -\frac{v_{k+1}}{v_k} v_k + 1 \cdot v_{k+1} = -v_{k+1} + v_{k+1} = 0.$$

Quindi la matrice elementare di Gauss L (triangolare inferiore a diagonale unitaria) ha appunto la caratteristica di annullare le componenti di $\underline{v}$ dalla $(k+1)$-esima in poi, lasciando invariate le prime k, se moltiplicata a sinistra per $\underline{v}$. Si noti che sia il vettore che la matrice elementari di Gauss sono definiti se e solo se $v_k \neq 0$. L'inversa della matrice elementare di Gauss L, L^{-1}, si ottiene facilmente come

$$L^{-1} = I + \underline{g}\,\underline{e_k}^T. \tag{3.3}$$

Infatti si ha che

$$L^{-1}L = (I + \underline{g}\,\underline{e_k}^T)(I - \underline{g}\,\underline{e_k}^T) = I - \underline{g}\,\underline{e_k}^T + \underline{g}\,\underline{e_k}^T - \underline{g}\,\underbrace{\underline{e_k}^T\,\underline{g}}_{=0}\,\underline{e_k}^T = I,$$

essendo $\underline{e_k}^T\,\underline{g}$ il k-esimo elemento del vettore elementare di Gauss, che è nullo per definizione di $\underline{g}$.

Tornando adesso al problema della fattorizzazione della matrice A del sistema (3.1), l'idea di base è quella di trasformare la matrice A in una matrice *triangolare superiore*, moltiplicando A a sinistra per opportune matrici elementari di Gauss. Tale metodo iterativo è detto **metodo di eliminazione di Gauss** e fattorizza la matrice A LU in $n-1$ passi.
Indichiamo con

$$A \equiv A^{(1)} = \begin{pmatrix} a_{11}^{(1)} & \dots & a_{1n}^{(1)} \\ \vdots & & \vdots \\ a_{n1}^{(1)} & \dots & a_{nn}^{(1)} \end{pmatrix}$$

la matrice al primo passo, dove l'indice superiore indica il passo più recente in cui un certo elemento è stato modificato.
Il primo passo per rendere A triangolare superiore è quello di rendere nulli tutti gli elementi della prima colonna dal secondo in poi, ovvero far assumere ad A, almeno per quanto riguarda la prima colonna, una struttura uguale a quella di una matrice triangolare superiore. Se risulta che

$$a_{11}^{(1)} \neq 0,$$

allora possiamo definire il vettore elementare di Gauss $\underline{g_1}$ e la corrispondente matrice L_1,

$$\underline{g_1} \equiv \frac{1}{a_{11}^{(1)}}(0, a_{21}^{(1)}, \dots, a_{n1}^{(1)})^T, \qquad L_1 \equiv I - \underline{g_1}\,\underline{e_1}^T = \begin{pmatrix} 1 & & & \\ -\frac{a_{21}^{(1)}}{a_{11}^{(1)}} & 1 & & \\ \vdots & & \ddots & \\ -\frac{a_{n1}^{(1)}}{a_{11}^{(1)}} & & & 1 \end{pmatrix},$$

tali che

$$L_1 A = \begin{pmatrix} a_{11}^{(1)} & \dots & \dots & a_{1n}^{1} \\ 0 & a_{22}^{2} & \dots & a_{2n}^{(2)} \\ \vdots & \vdots & & \vdots \\ 0 & a_{n2}^{(2)} & \dots & a_{nn}^{2} \end{pmatrix} \equiv A^{(2)}.$$

Osserviamo che, tramite questa procedura, nel passaggio da $A^{(1)}$ ad $A^{(2)}$ la prima riga della matrice è rimasta invariata.
Reiterando fino al passo i-esimo, avremo ottenuto, se $a_{jj}^{(j)} \neq 0$ per ogni $j < i$:

$$L_{i-1} \dots L_2 L_1 A = \begin{pmatrix} a_{11}^{(1)} & \dots & \dots & \dots & \dots & a_{1n}^{(1)} \\ 0 & \ddots & & & & \vdots \\ \vdots & \ddots & a_{i-1,i-1}^{(i-1)} & \dots & \dots & a_{i-1,n}^{(i-1)} \\ \vdots & & 0 & a_{ii}^{(i)} & \dots & a_{in}^{(i)} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & a_{ni}^{(i)} & \dots & a_{nn}^{(i)} \end{pmatrix} \equiv A^{(i)}.$$

Se a sua volta risulta che

$$a_{ii}^{(i)} \neq 0, \tag{3.4}$$

allora possiamo definire l'i-esimo vettore elementare di Gauss e la relativa matrice:

$$\underline{g_i} \equiv \frac{1}{a_{ii}^{(i)}} (\underbrace{0, \dots, 0}_{i}, a_{i+1,i}^{(i)}, \dots, a_{ni}^{(i)})^T,$$

$$L_i \equiv I - \underline{g_i}\, \underline{e_i}^T = \begin{pmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & -\frac{a_{i+1,i}^{(i)}}{a_{ii}^{(i)}} & \ddots & & \\ & & \vdots & & \ddots & \\ & & -\frac{a_{ni}^{(i)}}{a_{ii}^{(i)}} & & & 1 \end{pmatrix} \begin{matrix} \\ \\ i \\ \\ \\ \\ \end{matrix} \text{ riga } i+1 \ , \tag{3.5}$$

tali che

$$L_i A^{(i)} = L_i \dots L_1 A = \begin{pmatrix} a_{11}^{(1)} & \dots & \dots & \dots & \dots & a_{1n}^{(1)} \\ 0 & \ddots & & & & \vdots \\ \vdots & \ddots & a_{ii}^{(i)} & \dots & \dots & a_{in}^{(i)} \\ \vdots & & 0 & a_{i+1,i+1}^{(i+1)} & \dots & a_{i+1,n}^{(i+1)} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & a_{n,i+1}^{(i+1)} & \dots & a_{nn}^{(i+1)} \end{pmatrix} \equiv A^{(i+1)}.$$

Dal momento che le prime i righe dell'i-esima matrice elementare di Gauss L_i coincidono con le prime i righe della matrice identità I, le prime i righe della matrice non vengono modificate nel passaggio da $A^{(i)}$) ad $A^{(i+1)}$.
Se risulta che $a_{jj}^{(j)} \neq 0$ per ogni $j < n$, allora al passo $n-1$ avremo ottenuto

$$L_{n-1} \dots L_1 A = \begin{pmatrix} a_{11}^{(1)} & \dots & \dots & a_{1n}^{(1)} \\ 0 & \ddots & & \vdots \\ \vdots & \ddots & a_{n-1,n-1}^{(n-1)} & a_{n-1,n}^{(n-1)} \\ 0 & \dots & 0 & a_{nn}^{(n)} \end{pmatrix} \equiv A^{(n)} \equiv U. \tag{3.6}$$

Quindi la matrice A di partenza è stata correttamente trasformata nella matrice U *triangolare superiore*. Denotando con il simbolo L^{-1} la matrice

$$L_{n-1} \dots L_1,$$

risulta che

$$\begin{aligned} &L_{n-1} \dots L_1 A = U \\ &L^{-1} A = U \\ &A = LU \end{aligned}$$

Risulta infatti che L^{-1} è una matrice *triangolare inferiore a diagonale unitaria*, in quanto prodotto di matrici triangolari inferiori a diagonale unitaria (vedi Esercizio 3.3), e tale risulta anche la sua inversa L (vedi Esercizio 3.4). Denotando con $g_{ki} \equiv \underline{e_k}^T \, \underline{g_i}$, ovvero il k-esimo elemento dell'i-esimo vettore elementare di Gauss, ricordiamo che, per costruzione del generico $\underline{g_i}$:

$$g_{ki} = 0, \qquad \text{per } k \leq i.$$

Allora la matrice L ricercata sarà data da (vedi (3.3))

$$\begin{aligned} L = (L^{-1})^{-1} &= (L_{n-1} \dots L_1)^{-1} = \\ &= L_1^{-1} \dots L_{n-1}^{-1} = \\ &= (I + \underline{g_1}\, \underline{e_1}^T) \dots (I + \underline{g_{n-1}}\, \underline{e_{n-1}}^T) = \\ &= I + \underline{g_1}\, \underline{e_1}^T + \dots + \underline{g_{n-1}}\, \underline{e_{n-1}}^T, \end{aligned}$$

dato che gli $\underline{e_i}^T$ con indice minore vengono sempre moltiplicati a sinistra per i $\underline{g_j}$ di indice maggiore, infatti i fattori $(\underline{g_i}\, \underline{e_i}^T\, \underline{g_j}\, \underline{e_j}^T)$ risulteranno tutti pari a zero, per quanto detto prima. Quindi la matrice L risulta essere

$$L = \begin{pmatrix} 1 & & & \\ g_{21} & 1 & & \\ \vdots & \ddots & \ddots & \\ g_{n1} & \dots & g_{n,n-1} & 1 \end{pmatrix}, \tag{3.7}$$

ovvero la matrice identità con gli elementi significativi dei vettori elementari di Gauss nella parte strettamente inferiore (nelle colonne corrispondenti).
Le matrici L ed U così ottenute formano la fattorizzazione LU ricercata per la matrice A.

Deriviamo adesso condizioni sufficienti affinché esista la fattorizzazione LU di A.
Abbiamo visto che per poter eseguire il metodo di eliminazione di Gauss è necessario poter costruire, ad ogni passo, la matrice elementare d Gauss e quindi il vettore elementare di Gauss, il che equivale a richiedere che sia verificata la (3.4) ad ogni passo.

Lemma 3.1. *Se A è nonsingolare, la fattorizzazione (3.2) è definita se e solo se $a_{ii}^{(i)} \neq 0$, $i = 1, 2, \ldots, n$, ovvero se e solo se U è nonsingolare.*

Definizione 3.4. *Si dice **sottomatrice principale di ordine** k di una generica matrice $A = (a_{ij}) \in \mathbb{R}^{n\times n}$:*

$$A_k = \begin{pmatrix} a_{11} & \ldots & a_{1k} \\ \vdots & & \vdots \\ a_{n1} & \ldots & a_{nn} \end{pmatrix},$$

ovvero A_k è la porzione di A ottenuta dall'intersezione delle sue prime k righe con le sue prime k colonne.

Definizione 3.5. *Il determinante della sottomatrice principale di ordine k A_k, $det(A_k)$, è detto **minore principale di ordine** k della matrice A.*

Come casi estremi, il minore principale di ordine 1 è a_{11}, mentre il minore principale di ordine n coincide con $det(A)$ ($= n$ se A è non singolare).

Lemma 3.2. *Una matrice triangolare è nonsingolare se e solo se tutti i suoi minori principali sono non nulli.*

Lemma 3.3. *Il minore di ordine k di A in (3.2) coincide con il minore di ordine k di U in (3.6).*

Quindi, grazie ai Lemmi 3.1, 3.2 e 3.3, possiamo enunciare il seguente Teorema:

Teorema 3.2 (Esistenza della fattorizzazione LU)**.** *Se A è nonsingolare, la fattorizzazione (3.2) esiste se e solo se tutti i minori principali di A sono non nulli.*

Si osservi che richiedere che tutti i minori principali di A siano non nulli è molto più restrittivo che richiedere che A sia nonsingolare, che equivale a richiedere che soltanto il minore principale di ordine massimo sia non nullo. Di seguito una

matrice nonsingolare d'esempio che ha tutti i minori principali nulli tranne quello di ordine massimo:

$$A = \begin{pmatrix} 0 & \dots & \dots & 0 & 1 \\ 1 & 0 & \dots & \dots & 0 \\ & 1 & \ddots & & \vdots \\ & & \ddots & \ddots & \vdots \\ & & & 1 & 0 \end{pmatrix}.$$

3.3 Costo computazionale

Per quanto riguarda l'occupazione di memoria del metodo di eliminazione di Gauss si ha che al passo i-esimo, vengono azzerate le componenti dalla $i+1$ alla n della colonna i della matrice $A^{(i)}$. Queste componenti sono esattamente $n-i$, come il numero di elementi significativi (le ultime $n-1$ componenti) dell'i-esimo vettore elementare di Gauss. Quindi queste ultime $n-i$ posizioni della colonna i di A, anziché azzerate, possono essere riscritte con gli elementi significativi di $\underline{g_i}$, che sono proprio gli elementi che, esattamente in colonna i, andranno a formare la matrice L. Quindi, con questo approccio, alla fine del metodo avremo riscritto la matrice A con

- le componenti significative della matrice U nella parte *strettamente triangolare superiore* (vedi (3.6));
- le componenti significative della matrice L (escludendo quindi la diagonale unitaria, vedi (3.7)) nella porzione *strettamente triangolare inferiore.*

Si conclude quindi che, dal punto di vista di occupazione di memoria, il metodo di eliminazione di Gauss non richiede spazio addizionale.

Analizzando invece il costo computazionale del metodo di eliminazione di Gauss, risulta che al passo i-esimo dovremo calcolare l'i-esimo vettore elementare di Gauss $\underline{g_i}$ e successivamente il prodotto $L_i A^{(i)}$, che è dato da

$$L_i A^{(i)} = (I - \underline{g_i}\,\underline{e_i}^T)A^{(i)} = A^{(i)} - \underline{g_i}(\underline{e_i}^T A^{(i)}) \equiv A^{(i+1)}.$$

Considerando che

- le prime i componenti del vettore $\underline{g_i}$ sono nulle,
- il vettore $\underline{e_i}^T A^{(i)}$ rappresenta l'i-esima riga della matrice $A^{(i)}$, le cui prime $i-1$ componenti sono nulle,
- è noto a priori (e quindi inutile calcolarle) che le ultime $n-i$ componenti della colonna i della matrice $A^{(i+1)}$ sono nulle (dalla $i+1$ alla n),

risulta che soltanto la sottomatrice quadrata delimitata da $(i+1, i+1)$ e (n, n) dovrà essere calcolata e modificata. Il costo computazionale si dimostra quindi essere (vedi Esercizio 3.6), per una matrice quadrata $n \times n$,

$$\approx \frac{2}{3}n^3 \quad \textbf{flop}. \tag{3.8}$$

Con questi accorgimenti, possiamo infine definire il metodo di fattorizzazione *LU* (Codice 3.5) ed il metodo per la risoluzione di un sistema lineare tramite fattorizzazione *LU* della matrice dei coefficienti (Codice 3.6).

Codice 3.5: Fattorizzazione *LU* di una matrice

```
% A = fattorizzaLU(A)
% Fattorizzazione LU di una matrice nonsingolare con tutti i minori
% principali non nulli.
%
% Input:
%   -A: la matrice nonsingolare da fattorizzare LU.
% Output:
%   -A: la matrice riscritta con le informazioni dei fattori L ed U.
%
% Autore: Tommaso Papini,
% Ultima modifica: 7 Ottobre 2012, 13:17 CEST

function [A] = fattorizzaLU(A)
    [m,n]=size(A);
    if m~=n
        error('La matrice non è quadrata!');
    end
    for i=1:n-1
        if A(i,i)==0
            error('La matrice non è fattorizzabile LU!');
        end
        A(i+1:n,i) = A(i+1:n,i)/A(i,i);
        A(i+1:n,i+1:n) = A(i+1:n,i+1:n)-A(i+1:n,i)*A(i,i+1:n);
    end
end
```

Codice 3.6: Risoluzione di un sistema lineare tramite fattorizzazione *LU* della matrice dei coefficienti

```
% b = risolviSistemaLU(A,b)
% Risolve il sistema lineare Ax=b fattorizzando LU la matrice A,
% nonsingolare e con tutti i minori principali non nulli.
%
% Input:
%   -A: matrice nonsingolare dei coefficienti;
%   -b: vettore dei termini noti.
% Output:
%   -b: vettore delle soluzioni.
%
```

```
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:22 CET

function [b]= risolviSistemaLU(A,b)
    A = fattorizzaLU(A);
    b = triangolareInfCol(tril(A,-1)+eye(length(A)), b);
    b = triangolareSupCol(triu(A), b);
end
```

3.4 Matrici a diagonale dominante

Studiamo adesso una particolare tipologia di matrici per le quali la nonsingolarità deriva da una sua proprietà algebrica e che hanno tutte le loro sottomatrici principali che godono della medesima proprietà.

Definizione 3.6. *Data a matrice $A = (a_{ij}) \in \mathbb{R}^{n\times n}$, si dice che A è:*

- ***diagonale dominante per righe*** *se*

$$|a_{ii}| > \sum_{j\neq i} |a_{ij}|, \qquad i = 1, \dots, n,$$

  ovvero se ogni elemento diagonale è strettamente maggiore della somma degli altri elementi sulla stessa riga (in valore assoluto);

- ***diagonale dominante per colonne*** *se*

$$|a_{ii}| > \sum_{j\neq i} |a_{ji}|, \qquad i = 1, \dots, n,$$

  ovvero se ogni elemento diagonale è strettamente maggiore della somma degli altri elementi sulla stessa colonna (in valore assoluto).

Lemma 3.4. *Se una matrice A è diagonale dominante per righe (rispettivamente, per colonne), allora tali sono tutte le sue sottomatrici principali.*

Lemma 3.5. *Una matrice A è diagonale dominante per righe (rispettivamente, per colonne) se e solo se A^T è diagonale dominante per colonne (rispettivamente, per righe).*

Lemma 3.6. *Se una matrice $A \in \mathbb{R}^{n\times n}$ è diagonale dominante per righe (rispettivamente, per colonne), allora è nonsingolare.*

Combinando i Lemmi 3.6 e 3.4 risulta che se una matrice è dominante per righe/colonne, allora tutte le sue sottomatrici principali sono nonsingolari (ovvero con determinante non nullo). Quindi si ricava facilmente il seguente Teorema:

Teorema 3.3. *Se A è diagonale dominante, per righe e/o colonne, allora è fattorizzabile LU.*

3.5 Matrici sdp: fattorizzazione LDL^T

Un'altra tipologia di matrici sempre fattorizzabili LU sono le matrici *simmetriche* e *definite positive* (più brevemente, *sdp*).

Definizione 3.7. *Una matrice $A \in \mathbb{R}^{n\times n}$ è **sdp** se è simmetrica (ovvero $A = A^T$) e, per ogni $\underline{x} \in \mathbb{R}^n$, $\underline{x} \neq 0$, risulta*

$$\underline{x}^T A \underline{x} > 0.$$

Per verificare se una matrice A simmetrica è definita positiva si può utilizzare il seguente metodo:

Teorema 3.4 (Teorema di Jacobi)**.** *Sia A una matrice simmetrica di rango n in cui tutti i minori principali, $\delta_1, \delta_2, \ldots, \delta_n$, sono non nulli. Allora A è congruente ad una matrice diagonale B del tipo*

$$B = \begin{pmatrix} \delta_1 & & & \\ & {}^{\delta_2}/_{\delta_1} & & \\ & & \ddots & \\ & & & {}^{\delta_n}/_{\delta_{n-1}} \end{pmatrix}.$$

Grazie al Teorema 3.4, basta studiare la positività degli elementi della diagonale della matrice B congruente:

- se tutti positivi: A è **definita positiva**;
- se tutti negativi: A è **definita negativa**;
- altrimenti: A è **non definita**.

Un altro metodo per studiare se una matrice è definita positiva consiste nell'applicare il Criterio di Sylvester.

Teorema 3.5 (Criterio di Sylvester)**.** *Sia $A \in \mathbb{R}^{n\times n}$ una matrice simmetrica reale. Per $i = 1, \ldots, n$, sia d_i il minore principale di ordine i di A. Allora A è definita positiva se e solo se*

$$d_i > 0, \qquad i = 1, \ldots, n.$$

Dal Criterio di Sylvester si può dedurre che:

Corollario 3.1. *Sia $A \in \mathbb{R}^{n\times n}$ una matrice simmetrica reale. Per $i = 1, \ldots, n$, sia d_i il minore principale di ordine i di A. Allora A è definita negativa se e solo se*

$$(-1)^i d_i > 0, \qquad i = 1, \ldots, n.$$

Lemma 3.7. *Tutte le sottomatrici principali di una matrice sdp sono a loro volta sdp.*

Lemma 3.8. *Una matrice sdp è nonsingolare.*

Teorema 3.6. *Gli elementi diagonali di una matrice sdp sono positivi.*

Teorema 3.7. *A è sdp se e solo se*

$$A = LDL^T, \tag{3.9}$$

con

- *L triangolare inferiore a diagonale unitaria,*
- *D diagonale con elementi diagonali positivi.*

Per la simmetria delle matrici ai due membri della (3.9), è sufficiente eguagliare gli elementi (i, j) di ciascun membro, per $i \geq j$, ovvero per la sola parte triangolare inferiore della matrice. Se $A = (a_{ij})$ e

$$L = \begin{pmatrix} 1 & & & \\ l_{21} & \ddots & & \\ \vdots & \ddots & \ddots & \\ l_{n1} & \dots & l_{n,n-1} & 1 \end{pmatrix} = \begin{pmatrix} l_{11} & & \\ \vdots & \ddots & \\ l_{n1} & \dots & l_{nn} \end{pmatrix}, \quad l_{jj} = 1,\ j = 1, \dots, n,$$

$$D = \begin{pmatrix} d_1 & & \\ & \ddots & \\ & & d_n \end{pmatrix},$$

allora, eguagliando gli elementi di uguale indice, per $i \geq j$, si ottiene:

$$a_{ij} = \underline{e_i}^T A \underline{e_j} \overset{1}{=} \underline{e_i}^T (LDL^T) \underline{e_j} = (\underline{e_i}^T L) D (\underline{e_j}^T L)^T \overset{2}{=}$$

$$= (l_{i1}, \ldots, l_{ii}, 0, \ldots, 0) \begin{pmatrix} d_1 & & \\ & \ddots & \\ & & d_n \end{pmatrix} \begin{pmatrix} l_{j1} \\ \vdots \\ l_{jj} \\ 0 \\ \vdots \\ 0 \end{pmatrix} =$$

$$= (l_{i1}d_1, \ldots, l_{ii}d_i, 0, \ldots, 0) \begin{pmatrix} l_{j1} \\ \vdots \\ l_{jj} \\ 0 \\ \vdots \\ 0 \end{pmatrix} =$$

$$= \sum_{k=1}^{j} l_{ik} l_{jk} d_k \overset{3}{=} \sum_{k=1}^{j-1} l_{ik} l_{jk} d_k + l_{ij} l_{jj} d_j \overset{4}{=}$$

$$= \sum_{k=1}^{j-1} l_{ik} l_{jk} d_k + l_{ij} d_j.^5$$

Distinguendo quindi i due casi $i = j$ ed $i > j$, si possono ricavare (vedi Esercizio 3.14) le seguenti espressioni per d_j ed l_{ij}, valide per $j = 1, \ldots, n$:

- $i = j$: diagonale

$$d_j = a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2 d_k. \tag{3.10}$$

- $i > j$: porzione strettamente triangolare inferiore

$$l_{ij} = \frac{a_{ij} - \sum_{k=1}^{j-1} l_{ik} l_{jk} d_k}{d_j}, \qquad i = j+1, \ldots, n. \tag{3.11}$$

[1] Con $\underline{e_i}^T$ viene selezionata la riga i-esima di A, mentre con $\underline{e_j}$ viene selezionata la colonna j-esima. Insieme selezionano l'elemento (i, j).

[2] $(\underline{e_i}^T L)$ seleziona la riga i-esima di L; $(\underline{e_j}^T L)^T$ seleziona la riga j-esima trasposta di L (ovvero la colonna j-esima di L^T).

[3] La sommatoria arriva fino a j in quanto, essendo $i \geq j$, gli elementi con indice maggiore di j verranno azzerati (poiché le ultime $n - j$ componenti del vettore colonna sulla destra sono nulle).

[4] Viene portato fuori dalla sommatoria l'ultimo prodotto (di indice j.

[5] Essendo $l_{jj} = 1$ in quanto elemento diagonale di L.

Si osservi che la matrice A, essendo *simmetrica*, può essere memorizzata in forma compressa, memorizzandone soltanto la porzione triangolare inferiore (o superiore). Inoltre si nota dalle espressioni poc'anzi ricavate, che il valore di a_{ij} non è più utilizzato una volta calcolato l_{ij} (analogamente, d_j se $i = j$). Quindi la porzione strettamente triangolare inferiore della matrice A può essere riscritta con gli elementi significativi della matrice L (la diagonale è formata sempre da tutti 1, quindi è inutile la sua memorizzazione) e la sua diagonale con gli elementi della matrice D. Se ne deduce che, dal punto di vista di occupazione di memoria, la fattorizzazione LDL^T non richiede spazio aggiuntivo (risparmiando ulteriore spazio nel caso in cui A venga memorizzata in formato compresso).

Per quanto riguarda, invece, il costo computazionale di tale fattorizzazione, si può dimostrare (vedi Esercizio 3.15) che vale circa

$$\approx \frac{1}{3}n^3 \textbf{ flop}, \tag{3.12}$$

ovvero circa la metà rispetto alla fattorizzazione LU con il metodo di eliminazione di Gauss.

Di seguito, le implementazioni in MATLAB della fattorizzazione LDL^T e della risoluzione di sistemi lineari tramite fattorizzazione LDL^T della matrice dei coefficienti:

Codice 3.7: Fattorizzazione LDL^T di una matrice.

```
% A = fattorizzaLDLt(A)
% Fattorizzazione LDLt di una matrice simmetrica e definita posiziva
% (sdp).
%
% Input:
%   -A: la matrice sdp da fattorizzare LDLt.
% Output:
%   -A: la matrice riscritta con le informazioni di L e D.
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:14 CET

function [A] = fattorizzaLDLt(A)
    [m,n]=size(A);
    if m~=n
        error('La matrice non è quadrata!');
    end
    if A(1,1)<=0
        error('La matrice non è sdp!');
    end
    A(2:n,1) = A(2:n,1)/A(1,1);
    for j=2:n
        v = ( A(j,1:(j-1))').*diag(A(1:(j-1),1:(j-1)));
        A(j,j) = A(j,j)-A(j,1:(j-1))*v;
        if A(j,j)<=0
```

```
            error('La matrice non è sdp!');
        end
        A((j+1):n,j) = (A((j+1):n,j)-A((j+1):n,1:(j-1))*v)/A(j,j);
    end
end
```

Codice 3.8: Risoluzione di un sistema lineare tramite fattorizzazione LDL^T della matrice dei coefficienti.

```
% b = risolviSistemaLDLt(A, b)
% Risoluzione di un sistema lineare con matrice dei coefficienti A sdp
% fattorizzando LDLt la matrice dei coefficienti.
%
% Input:
%   -A: la matrice sdp dei coefficienti;
%   -b: vettore dei termini noti.
% Output:
%   -b: vettore soluzione del sistema.
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:22 CET

function [b] = risolviSistemaLDLt(A, b)
    A = fattorizzaLDLt(A);
    b = triangolareInfCol(tril(A,-1)+eye(length(A)), b);
    b = diagonale(diag(A), b);
    b = triangolareSupCol((tril(A,-1)+eye(length(A)))', b);
end
```

3.6 *Pivoting*

Studiamo adesso il caso in cui non tutte le ipotesi del Teorema 3.2 sono verificate. In particolare studiamo il caso in cui la matrice A è nonsingolare, ma esiste almeno un minore principale di A nullo. L'obiettivo di questo capitolo è quello di definire un nuovo metodo affinché la fattorizzazione sia possibile anche sotto la sola ipotesi di nonsingolarità.
Supponiamo quindi di trovarsi al primo passo del metodo di eliminazione di Gauss e che non sia verificata la condizione (3.4), ovvero che $a_{ii}^{(i)} = 0$. Tuttavia, essendo A nonsingolare, esisterà sicuramente nella sua prima colonna un elemento non nullo:

$$|a_{k_1 1}^{(1)}| \equiv \max_{k \geq 1} |a_{k1}^{(1)}| > 0,$$

con k_1 che indica l'indice della riga dell'elemento massimo nella prima colonna (in valore assoluto) e $a_{k_1 1}$ l'elemento stesso.

Definizione 3.8. *La matrice identità di ordine n con le righe (e quindi, essendo simmetrica, anche le colonne) i e k_i, con $k_i \geq i$, scambiate tra loro*

$$P_i \equiv \left(\begin{array}{c|ccc|c} I_{i-1} & & & & \\ \hline & 0 & \underline{0}^T & 1 & \\ & \underline{0} & I_{k_i-i-1} & \underline{0} & \\ & 1 & \underline{0}^T & 0 & \\ \hline & & & & I_{n-k_i} \end{array}\right) \begin{array}{l} \\ \textit{riga } i \\ \\ \textit{riga } k_i \\ \\ \end{array}, \tag{3.13}$$

è detta ***matrice elementare di permutazione****.*

Si osserva che la matrice P_i è *simmetrica* e *ortogonale*, ovvero

$$P_i = P_i^T = P_i^{-1}.$$

Inoltre un'importante caratteristica della matrice elementare di permutazione è quella di scambiare tra loro le righe i e k_i se moltiplicata a sinistra per un vettore $\underline{v} = (v_1, \ldots, v_n)^T \in \mathbb{R}^n$

$$P_i \begin{pmatrix} v_1 \\ \vdots \\ v_i \\ \vdots \\ v_{k_1} \\ \vdots \\ v_n \end{pmatrix} = \begin{pmatrix} v_1 \\ \vdots \\ v_{k_i} \\ \vdots \\ v_i \\ \vdots \\ v_n \end{pmatrix},$$

o per una matrice A

$$P_i \begin{pmatrix} a_{11} & \ldots & \ldots & \ldots & a_{1n} \\ \vdots & & & & \vdots \\ a_{i1} & \ldots & \ldots & \ldots & a_{in} \\ \vdots & & & & \vdots \\ a_{k_i1} & \ldots & \ldots & \ldots & a_{k_in} \\ \vdots & & & & \vdots \\ a_{n1} & \ldots & \ldots & \ldots & a_{nn} \end{pmatrix} = \begin{pmatrix} a_{11} & \ldots & \ldots & \ldots & a_{1n} \\ \vdots & & & & \vdots \\ a_{k_i1} & \ldots & \ldots & \ldots & a_{k_in} \\ \vdots & & & & \vdots \\ a_{i1} & \ldots & \ldots & \ldots & a_{in} \\ \vdots & & & & \vdots \\ a_{n1} & \ldots & \ldots & \ldots & a_{nn} \end{pmatrix}.$$

Quindi la matrice elementare di permutazione relativa alla prima riga sarà

$$P_1 \equiv \left(\begin{array}{ccc|c} 0 & \underline{0}^T & 1 & \\ \underline{0} & I_{k_1-2} & \underline{0} & \\ 1 & \underline{0}^T & 0 & \\ \hline & & & I_{n-k_1} \end{array}\right) \text{ riga } k_1 \ .$$

Moltiplicandola a sinistra per la matrice al primo step $A^{(1)}$, otterremo quindi la stessa matrice $A^{(1)}$ ma con le righe 1 e k_1 permutate tra loro:

$$P_1 A^{(1)} = \begin{pmatrix} a^{(1)}_{k_1 1} & \dots & \dots & \dots & a^{(1)}_{k_1 n} \\ \vdots & & & & \vdots \\ a^{(1)}_{11} & \dots & \dots & \dots & a^{(1)}_{1n} \\ \vdots & & & & \vdots \\ a^{(1)}_{n1} & \dots & \dots & \dots & a^{(1)}_{nn} \end{pmatrix} \begin{matrix} \text{riga } 1 \\ \\ \\ \text{riga } k_1 \\ \\ \end{matrix} .$$

Quindi risulta chiaro che adesso è possibile applicare il primo passo del metodo di eliminazione di Gauss, definendo il primo vettore elementare di Gauss

$$\underline{g_1} = \frac{1}{a^{(1)}_{k_1 1}} (0, a^{(1)}_{21}, \dots, \overset{k_1}{a^{(1)}_{11}}, \dots, a^{(1)}_{n1})^T,$$

e la prima matrice elementare di Gauss

$$L_1 = I - \underline{g_1}\, \underline{e_1}^T = \begin{pmatrix} 1 & & & & & \\ -\frac{a^{(1)}_{21}}{a^{(1)}_{k_1 1}} & \ddots & & & & \\ \vdots & & \ddots & & & \\ -\frac{a^{(1)}_{11}}{a^{(1)}_{k_1 1}} & & & \ddots & & \\ \vdots & & & & \ddots & \\ -\frac{a^{(1)}_{n1}}{a^{(1)}_{k_1 1}} & & & & & 1 \end{pmatrix} \begin{matrix} \\ \\ \\ \text{riga } k_1 \\ \\ \\ \end{matrix} .$$

Segue che, combinando la permutazione di P_1 e l'eliminazione di Gauss di L_1 otteniamo la matrice al secondo step, ovvero la matrice con la prima colonna strutturalmente uguale a quella di una matrice *triangolare superiore*:

$$L_1 P_1 A^{(1)} = \begin{pmatrix} a^{(1)}_{k_1 1} & \dots & \dots & a^{(1)}_{k_1 n} \\ 0 & a^{(2)}_{22} & \dots & a^{(2)}_{2n} \\ \vdots & \vdots & & \vdots \\ 0 & a^{(2)}_{n2} & \dots & a^{(2)}_{nn} \end{pmatrix} \equiv A^{(2)}.$$

Analogamente a quanto visto in Sezione 3.2, la procedura viene reiterata, otte-

nendo, al generico passo i-esimo, la matrice

$$L_{i-1}P_{i-1}\dots L_1P_1A = \begin{pmatrix} a_{k_1 1}^{(1)} & \dots & \dots & \dots & \dots & a_{k_1 n}^{(1)} \\ 0 & \ddots & & & & \vdots \\ \vdots & \ddots & a_{k_{i-1},i-1}^{(i-1)} & \dots & \dots & a_{k_{i-1},n}^{(i-1)} \\ \vdots & & 0 & a_{ii}^{(i)} & \dots & a_{in}^{i} \\ \vdots & & \vdots & & & \\ 0 & \dots & 0 & a_{ni}^{(i)} & \dots & a_{nn}^{(i)} \end{pmatrix} \equiv A^{(i)}.$$

Definiamo allora l'i-esimo elemento **pivot** (o **di perno**)

$$|a_{k_i i}^{(i)}| \equiv \max_{k \geq i} |a_{ki}^{(i)}|, \tag{3.14}$$

per il quale si avrà, essendo A nonsingolare per ipotesi, $a_{k_i i}^{(i)} \neq 0$ (vedi Esercizio 3.19) Si definisce quindi, a partire dall'i-esimo pivot (3.14), l'i-esima matrice elementare di permutazione P_i (vedi (3.13)). Quindi, moltiplicando P_i a sinistra per $A^{(i)}$ si ottiene la permutazione delle righe i e k_i della matrice $A^{(i)}$ (ovviamente, soltanto nelle colonne dalla i alla n si potrà osservare un effettivo cambiamento). Possiamo a questo punto definire l'i-esimo vettore elementare di Gauss

$$\underline{g_i} = \frac{1}{a_{k_i i}^{(i)}}(\underbrace{0,\dots,0}_{i},\overset{k_i}{a_{i+1,i}^{(i)}},\dots,a_{ii}^{(i)},\dots,a_{ni}^{(i)})^T, \tag{3.15}$$

e l'i-esima matrice elementare di Gauss

$$L_i = I - \underline{g_i}\,\underline{e_i}^T = \begin{pmatrix} \overset{i}{1} & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & -\frac{a_{i+1,i}^{(i)}}{a_{k_i i}^{(i)}} & \ddots & & \\ & & \vdots & & \ddots & \\ & & -\frac{a_{ii}^{(i)}}{a_{k_i i}^{(i)}} & & & \ddots \\ & & \vdots & & & & \ddots \\ & & -\frac{a_{ni}^{(i)}}{a_{k_i i}^{(i)}} & & & & & 1 \end{pmatrix} \begin{matrix} \\ \\ \\ \text{riga } i+1 \\ \\ \\ \\ \end{matrix},$$

tali che

$$L_i P_i A^{(i)} = L_i P_i \dots L_1 P_1 A = \begin{pmatrix} a_{k_1 1}^{(1)} & \dots & \dots & \dots & \dots & a_{k_1 n}^{(1)} \\ 0 & \ddots & & & & \vdots \\ \vdots & \ddots & a_{k_i i}^{(i)} & \dots & \dots & a_{k_i n}^{(i)} \\ \vdots & & 0 & a_{i+1,i+1}^{(i+1)} & \dots & a_{i+1,n}^{(i+1)} \\ \vdots & & \vdots & \vdots & & \vdots \\ 0 & \dots & 0 & a_{n,i+1}^{(i+1)} & \dots & a_{nn}^{(i+1)} \end{pmatrix} \equiv A^{(i+1)}.$$

Procedendo in questo modo, se A è nonsingolare, sarà sempre possibile effettuare il passo di eliminazione di Gauss sulla matrice permutata, fino al passo $n-1$, ottenendo

$$L_{n-1} P_{n-1} \dots L_1 P_1 A = \begin{pmatrix} a_{k_1 1}^{(1)} & \dots & \dots & a_{1n}^{(1)} \\ & \ddots & & \vdots \\ & & a_{k_{n-1},n-1}^{(n-1)} & a_{k_{n-1},n}^{(n-1)} \\ & & & a_{nn}^{(n)} \end{pmatrix} = A^{(n)} \equiv U. \quad (3.16)$$

Considerando che le matrici elementari di permutazione P_i sono *simmetriche* ($P_i = P_i^T$) ed *ortogonali* ($P_i^T = P_i^{-1}$), e che quindi $P_i = P_i^{-1}$, allora possiamo riscrivere la (3.16) come

$$\hat{L}_{n-1} \dots \hat{L}_1 P A = U, \quad (3.17)$$

con

$$\begin{aligned} &\hat{L}_{n-1} \equiv L_{n-1} \\ &\hat{L}_i \equiv P_{n-1} \dots P_{i+1} L_i P_{i+1} \dots P_{n-1}, \quad i = 1, \dots, n-2, \\ &P \equiv P_{n-1} \dots P_1. \end{aligned}$$

Le due espressioni risultano equivalenti, infatti, se prendiamo ad esempio $n = 4$, risulta:

$$\begin{aligned} \hat{L}_3\ \hat{L}_2\ \hat{L}_1\ PA &= L_3\ P_3 L_2 \underbrace{P_3\ P_3}_{I} P_2 L_1 P_2 \underbrace{P_3\ P_3}_{I} P_2 P_1 A = \\ &= L_3\ P_3 L_2\ P_2 L_1 \underbrace{P_2\ P_2}_{I} P_1 A = \\ &= L_3\ P_3 L_2\ P_2 L_1\ P_1 A = \\ &= U. \end{aligned}$$

Osserviamo che P è una **matrice di permutazione** (non più *elementare* in quanto non interessa soltanto due righe), ovvero una matrice *ortogonale* che, moltiplicata a sinistra per un vettore (rispettivamente, per una matrice) ne permuta le componenti (rispettivamente, le righe). Inoltre sappiamo, per costruzione della

generica matrice elementare di permutazione (vedi (3.13)), che la riga i di P_j, con $i < j$, corrisponde alla riga i della matrice identità, ovvero $\underline{e_i}^T$, in quanto la matrice P_j è la matrice identità con due righe (una di indice $j > i$, l'altra con indice maggiore di j) permutate tra loro. Quindi risulta che $\underline{e_i}^T P_j = \underline{e_i}^T$, per $i < j$.
Si ha allora che

$$\begin{aligned}
\hat{L}_i &= P_{n-1} \dots P_{i+1} L_i P_{i+1} \dots P_{n-1} = \\
&= P_{n-1} \dots P_{i+1} (I - \underline{g_i}\, \underline{e_i}^T) P_{i+1} \dots P_{n-1} = \\
&= P_{n-1} \dots P_{i+1} I P_{i+1} \dots P_{n-1} - (P_{n-1} \dots P_{i+1} \underline{g_i}\, \underline{e_i}^T P_{i+1} \dots P_{n-1}) = \\
&= I - \underbrace{(P_{n-1} \dots P_{i+1} \underline{g_i})}_{\underline{\hat{g}_i}} (\underline{e_i}^T P_{i+1} \dots P_{n-1}) \equiv \\
&\equiv I - \underline{\hat{g}_i}\, \underline{e_i}^T, \qquad (3.18)
\end{aligned}$$

dove $\underline{\hat{g}_i}$ ha la stessa struttura del vettore $\underline{g_i}$ ma con le ultime $n - i$ componenti permutate tra loro (senza quindi mai scambiare una componente nulla con una non nulla). Quindi la struttura della matrice $\hat{L}_i$ è analoga a quella dell'i-esima matrice elementare di Gauss (3.5) (triangolare inferiore a diagonale unitaria). Quindi, come già visto nella Sezione 3.2, la matrice $\hat{L}_{n-1} \dots \hat{L}_1 \equiv L^{-1}$ è una matrice triangolare inferiore a diagonale unitaria e tale risulta essere la sua inversa L.
Pertanto si ottiene, dalla (3.17) e dalle considerazioni appena fatte, la seguente **fattorizzazione LU con pivoting parziale**:

$$PA = LU. \qquad (3.19)$$

La tecnica di *pivoting* utilizzata per questa fattorizzazione è detta *parziale* in quanto le permutazioni effettuate sono sempre tra sole righe di una matrice (quando si implementano permutazioni che scambiano sia righe che colonne allora si parla di **pivoting totale**).

Teorema 3.8. *Se A è una matrice nonsingolare, allora esiste una matrice di permutazione P tale che PA è fattorizzabile LU.*

Per poter risolvere, quindi, il sistema (3.1) tramite fattorizzazione LU con pivoting parziale si ha:

$$\begin{aligned}
&A\underline{x} = \underline{b}, \\
&PA\underline{x} = P\underline{b}, \\
&L \underbrace{U\underline{x}}_{\underline{y}} = P\underline{b}, \\
&L\underline{y} = P\underline{b}, \quad U\underline{x} = \underline{y}.
\end{aligned}$$

Quindi, una volta ottenuta la fattorizzazione (3.19), si moltiplica il vettore dei termini noti $\underline{b}$ per la matrice di permutazione P e successivamente si risolvono i

due sistemi triangolari $L\underline{y} = P\underline{b}$ e $U\underline{x} = \underline{y}$ (nell'ordine specificato).

Riguardo all'occupazione di memoria della fattorizzazione (3.19) si vede che i vettori di Gauss (3.15) devono essere soggetti a tutte le permutazioni successive alla loro definizione (ovvero, per il vettore $\underline{g_i}$, dalla $i+1$ alla $n-1$, vedi (3.18)) e possono quindi essere convenientemente memorizzati (limitandosi alle sole componenti significative) nella porzione strettamente triangolare inferiore (altrimenti formata di soli elementi nulli) della matrice A. Quindi la matrice A può essere riscritta con le componenti significative delle matrici L ed U, analogamente a quanto visto in Sezione 3.2.
Inoltre le informazioni della matrice di permutazione P possono essere memorizzate all'interno di un vettore $\underline{p}$ come segue:

$$\underline{p} = (k_1, k_2, \ldots, k_n)^T.$$

Per quanto riguarda invece il costo computazionale, si ha che per calcolare i *pivot* (3.14) ai passi $i = 1, 2, \ldots, n-1$, sono necessari

$$\sum_{i=1}^{n-1}(n-i) \approx \frac{n^2}{2} \quad \text{confronti.}$$

Questo costo è chiaramente trascurabile rispetto al numero di `flop` eseguite per ottenere la fattorizzazione, quindi il costo complessivo rimane identico a quello della fattorizzazione LU senza pivoting, ovvero

$$\approx \frac{2}{3}n^3 \quad \textbf{flop}.$$

Il metodo di fattorizzazione LU con *pivoting* parziale può essere implementato in MATLAB come segue:

Codice 3.9: Fattorizzazione LU con *pivoting* parziale di una matrice.

```
% [A, p] = fattorizzaLUpivoting(A)
% Fattorizzazione LU con pivoting parziale di una matrice nonsingolare.
%
% Input:
%   -A: la matrice da fattorizzare LU con pivoting parziale.
% Output:
%   -A: la matrice riscritta con l'informazione dei fattori L ed U;
%   -p: vettore contente l'informazione della matrice di permutazione
%   P.
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:15 CET

function [A, p] = fattorizzaLUpivoting(A)
    [m,n]=size(A);
```

```
    if m~=n
        error('La matrice non è quadrata!');
    end
    p=[1:n];
    for i=1:(n-1)
        [aki, ki] = max(abs(A(i:n, i)));
        if aki==0
            error('La matrice è singolare!');
        end
        ki = ki +i -1;
        if ki>i
            A([i, ki], :) = A([ki, i], :);
            p([i, ki]) = p([ki, i]);
        end
        A((i+1):n, i) = A((i+1):n, i)/A(i, i);
        A((i+1):n, (i+1):n) = A((i+1):n, (i+1):n) -A((i+1):n, i)*A(i, (
            i+1):n);
    end
end
```

Codice 3.10: Risoluzione di un sistema lineare tramite fattorizzazione *LU* con *pivoting* parziale della matrice dei coefficienti.

```
% b = risolviSistemaLUpivoting(A,b)
% Risolve il sistema lineare Ax=b fattorizzando LU con pivoting
% parziale la matrice nonsingolare A.
%
% Input:
%   -A: matrice nonsingolare dei coefficienti;
%   -b: vettore dei termini noti.
% Output:
%   -b: vettore delle soluzioni.
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:22 CET

function [b]= risolviSistemaLUpivoting(A,b)
    [A,p] = fattorizzaLUpivoting(A);
    P=zeros(length(A));
    for i=1:length(A)
        P(i, p(i)) = 1;
    end
    b = triangolareInfCol(tril(A,-1)+eye(length(A)), P*b);
    b = triangolareSupCol(triu(A), b);
end
```

3.7 Condizionamento del problema

Studieremo adesso come perturbazioni sui dati in ingresso al sistema lineare (3.1) si ripercuotono sulla soluzione. In particolare studieremo il sistema lineare

perturbato

$$(A + \Delta A)(\underline{x} + \underline{\Delta x}) = \underline{b} + \underline{\Delta b}, \tag{3.20}$$

dove ΔA e $\underline{\Delta b}$ rappresentano le perturbazioni sui dati in ingresso (rispettivamente, sulla matrice dei coefficienti e sul vettore dei termini noti) e $\underline{\Delta x}$ invece la perturbazione sulla soluzione finale.
Supponiamo, per semplicità di esposizione, che le perturbazioni appena descritte dipendano da un *parametro scalare di perturbazione* $\varepsilon \approx 0$,

$$A(\varepsilon) = A + \varepsilon F, \quad F \in \mathbb{R}^{n\times n}, \quad \Rightarrow \quad \Delta A = \varepsilon F, \tag{3.21}$$

$$\underline{b(\varepsilon)} = \underline{b} + \varepsilon \underline{f}, \quad \underline{f} \in \mathbb{R}^n, \quad \Rightarrow \quad \underline{\Delta b} = \varepsilon \underline{f}. \tag{3.22}$$

Quindi il sistema perturbato (3.20) diviene

$$A(\varepsilon)\underline{x(\varepsilon)} = \underline{b(\varepsilon)}, \tag{3.23}$$

dove $\underline{x(\varepsilon)}$ rappresenta la soluzione del sistema perturbato in funzione del parametro ε. Ovviamente se l'errore risulta essere nullo ($\varepsilon = 0$), allora i dati in ingresso risultano essere esatti

$$A(0) = A, \qquad \underline{b(0)} = \underline{b}, \tag{3.24}$$

e, di conseguenza, anche la soluzione del sistema

$$\underline{x(0)} = \underline{x}. \tag{3.25}$$

Sviluppando in $\varepsilon = 0$ si ottiene, per ε sufficientemente piccolo:

$$\begin{aligned}
\underline{x(\varepsilon)} &= P_{1,x}(\varepsilon; 0) + R_1(\varepsilon; 0) = \\
&= \underline{x(0)} + (\varepsilon - 0)\underline{\dot{x}(0)} + O((\varepsilon - 0)^2) = \\
&= \underline{x} + \varepsilon\underline{\dot{x}(0)} + O(\varepsilon^2) \approx \\
&\approx \underline{x} + \varepsilon\underline{\dot{x}(0)},
\end{aligned}$$

ovvero

$$\begin{aligned}
\underline{x(\varepsilon)} &\equiv \underline{x} + \underline{\Delta x}, \\
\underline{\Delta x} &= \underline{x(\varepsilon)} - \underline{x} \approx \\
&\approx \underline{x} + \varepsilon\underline{\dot{x}(0)} - \underline{x} \approx \\
&\approx \varepsilon\underline{\dot{x}(0)}.
\end{aligned} \tag{3.26}$$

Inoltre, sviluppando anche il sistema (3.23) in $\varepsilon = 0$, otteniamo (considerando le (3.24) e (3.25)):

$$\begin{aligned}
&A(\varepsilon)\underline{x(\varepsilon)} = \underline{b(\varepsilon)}, \\
&P_{1,Ax}(\varepsilon; 0) + R_1(\varepsilon; 0) = P_{1,b}(\varepsilon; 0) + R_1(\varepsilon; 0), \\
&\underbrace{A(0)\underline{x(0)}}_{\underline{b(0)}=\underline{b}} + (\varepsilon - 0)[\dot{A}(0)\underbrace{\underline{x(0)}}_{\underline{x}} + \underbrace{A(0)}_{A}\underline{\dot{x}(0)}] = \underbrace{\underline{b(0)}}_{\underline{b}} + (\varepsilon - 0)\underline{\dot{b}(0)}, \\
&\underline{b} + \varepsilon[\dot{A}(0)\underline{x(0)} + A\underline{\dot{x}(0)}] = \underline{b} + \varepsilon\underline{\dot{b}(0)}, \\
&\dot{A}(0)\underline{x(0)} + A\underline{\dot{x}(0)} = \underline{\dot{b}(0)}.
\end{aligned} \tag{3.27}$$

Grazie a considerazioni del tutto analoghe alla (3.26), si ha che

$$\Delta A \approx \varepsilon \dot{A}(0), \qquad \underline{\Delta b} \approx \varepsilon \underline{\dot{b}(0)},$$

ovvero (grazie alle (3.21) e (3.22):

$$F \approx \dot{A}(0), \qquad \underline{f} \approx \underline{\dot{b}(0)}.$$

Allora si ottiene, per la (3.27) e le considerazioni appena fatte, che

$$\begin{aligned} A\underline{\dot{x}(0)} &= \underline{\dot{b}(0)} - \dot{A}(0)\underline{x}, \\ \underline{\dot{x}(0)} &= A^{-1}(\underline{\dot{b}(0)} - \dot{A}(0)\underline{x}), \\ \underline{\dot{x}(0)} &\approx A^{-1}(\underline{f} - F\underline{x}). \end{aligned} \tag{3.28}$$

Sostituendo la (3.28) nella (3.27), considerando nuovamente le (3.21) e (3.22) e ricordando le proprietà della funzione norma

- $||\underline{x} + \underline{y}|| \leq ||\underline{x}|| + ||\underline{y}||$,
- $||A\underline{x}|| \leq ||A|| \cdots ||\underline{x}||$,

si ottiene che:

$$\begin{aligned} \frac{||\underline{\Delta x}||}{||\underline{x}||} \approx \frac{||A^{-1}(\underline{f} - F\underline{x})||}{||\underline{x}||} \equiv \frac{||A^{-1}(\underline{\Delta b} - \Delta A\underline{x})||}{||\underline{x}||} \leq \\ \leq \frac{||A^{-1}||(||\underline{\Delta b}|| - ||\Delta A|| \cdot ||\underline{x}||)}{||\underline{x}||} = \\ = ||A^{-1}|| \left(\frac{||\underline{\Delta b}||}{||\underline{x}||} + ||\Delta A|| \right) = \\ = ||A|| \cdot ||A^{-1}|| \left(\frac{||\underline{\Delta b}||}{||A|| \cdot ||\underline{x}||} + \frac{||\Delta A||}{||A||} \right) \leq \\ \leq ||A|| \cdot ||A^{-1}|| \left(\frac{||\underline{\Delta b}||}{||\underline{b}||} + \frac{||\Delta A||}{||A||} \right). \end{aligned}$$

Se, a questo punto, si considera che

- la quantità $\frac{||\underline{\Delta x}||}{||\underline{x}||}$ può essere interpretata, in un certo senso, come una sorta di *errore relativo* sulla soluzione e, in modo analogo,
- $\frac{||\Delta A||}{||A||}$ e $\frac{||\underline{\Delta b}||}{||\underline{b}||}$ come *errori relativi* sui dati di ingresso,

si ottiene che la quantità

$$k(A) \equiv ||A|| \cdot ||A^{-1}||,$$

definisce il *numero di condizionamento del problema.*

Definizione 3.9. *La quantità*

$$k(A) \equiv ||A|| \cdot ||A^{-1}||,$$

è denominata ***numero di condizionamento della matrice*** *A.*

Si osservi che, per qualsiasi norma indotta su matrice, si ha che

$$k(A) = ||A|| \cdot ||A^{-1}|| \geq ||AA^{-1}|| = ||I|| = 1.$$

Se

- $k(A)$ è una quantità "piccola", la matrice A si dice **ben condizionata**;
- $k(A) \gg 1$, allora la matrice si dice **malcondizionata**.

3.8 Sistemi lineari sovradeterminati

Un **sistema lineare sovradeterminato** è un sistema di equazioni lineari con più equazioni (m) che incognite (n), con la matrice A dei coefficienti di rango massimo ($= n$). Formalmente si vuole quindi risolvere il seguente sistema:

$$A\underline{x} = \underline{b}, \qquad A \in \mathbb{R}^{m \times n}, \qquad m > n \equiv rank(A). \tag{3.29}$$

Si osserva che, in generale, il sistema (3.29) non ammette soluzione. Infatti affinché ammetta soluzione deve risultare che $\underline{b} \in ran(A)$, con $ran(A)$ che indica il *range* di A. Tuttavia $\underline{b} \in \mathbb{R}^m$, mentre $dim(ran(A)) \equiv rank(A) = n < m$. L'unico caso in cui potrebbe ammettere soluzione è che la matrice dei coefficienti A presenti almeno $m-n$ righe eliminabili dal sistema in quanto linearmente dipendenti da altre.
Ricercheremo allora come soluzione il vettore $\underline{x} \in \mathbb{R}^n$ tale che il **vettore residuo**, $\underline{r} \in \mathbb{R}^m$,

$$\underline{r} = \begin{pmatrix} r_1 \\ \vdots \\ r_m \end{pmatrix} \equiv A\underline{x} - \underline{b},$$

sia minimizzato. Più precisamente ricercheremo il vettore $\underline{x}$ che minimizzi la quantità

$$\sum_{i=1}^{m} |r_i|^2 = ||\underline{r}||_2^2 = ||A\underline{x} - \underline{b}||_2^2, \tag{3.30}$$

dove la quantità

$$||\underline{r}||_2 \equiv \sqrt{\sum_{i=1}^{m} |r_i|^2} = \sqrt{\underline{r}^* \underline{r}},$$

indica la **norma 2** sul vettore r (o **norma Euclidea**) ed $\underline{r}^*$ l'**aggiunto** di $\underline{r}$ definito come $(\bar{\underline{r}})^T$, definendo anche $\bar{\underline{r}} \equiv (\bar{r}_i)$ come il **coniugato** di r (dove il coniugato di un *numero complesso* $z = x + iy$ è definito come $\bar{z} = x - iy$). Quindi, se $\underline{r}$ è a *valori reali*, risulta che $\bar{\underline{r}} = \underline{r}$ e $\underline{r}^* = \underline{r}^T$, ottenendo di conseguenza la seguente norma Euclidea:

$$||\underline{r}||_2 \equiv \sqrt{\underline{r}^T \underline{r}}.$$

Si parla quindi di soluzione del sistema (3.29) nel senso dei **minimi quadrati**.
Si osservi come risolvere il sistema (3.29) nel senso dei *minimi quadrati* significhi

minimizzare la norma di r in modo che il sistema lineare $A\underline{x} = \underline{b} + \underline{r}$ ammetta soluzione, ovvero tale $\underline{b} + \underline{r} \in ran(A)$.
Per risolvere questo tipo di problema si utilizza la **fattorizzazione** QR, descritta di seguito.

Teorema 3.9 (Fattorizzazione QR)**.** *Sia A la matrice dei coefficienti del sistema (3.29), esistono*

- $Q \in \mathbb{R}^{m\times m}$*, matrice ortogonale ($Q^{-1} = Q^T$),*
- $\hat{R} \in \mathbb{R}^{n\times n}$*, matrice triangolare superiore e nonsingolare (quadrata e di rango massimo),*

tali che

$$A = QR \equiv Q\begin{pmatrix}\hat{R}\\ O\end{pmatrix}, \tag{3.31}$$

dove O indica la matrice nulla di dimensioni opportune (in questo caso $O \in \mathbb{R}^{(m-n)\times n}$).

Tornando alla minimizzazione della (3.30) si ha allora che

$$\begin{aligned}||\underline{r}||_2^2 = ||A\underline{x} - \underline{b}||_2^2 = ||QR\underline{x} - \underline{b}||_2^2 \overset{1}{=} ||Q(\underline{x} - Q^{-1}\underline{b})||_2^2 =\\ = ||Q(\underline{x} - \underbrace{Q^T\underline{b}}_{\underline{g}})||_2^2 \equiv ||Q(R\underline{x} - \underline{g})||_2^2 \overset{2}{=} ||R\underline{x} - \underline{g}||_2^2 =\\ = \left|\left|\begin{pmatrix}\hat{R}\\ O\end{pmatrix}x - \begin{pmatrix}\underline{g_1}\\ \underline{g_2}\end{pmatrix}\right|\right|_2^2 \overset{3}{=} \left|\left|\begin{pmatrix}\hat{R}\underline{x} - \underline{g_1}\\ -\underline{g_2}\end{pmatrix}\right|\right|_2^2 =\\ = ||\hat{R}\underline{x} - \underline{g_1}||_2^2 + ||\underline{g_2}||_2^2.\end{aligned}$$ [4]

Detto questo, si vede che la minima norma possibile per $\underline{r}$ è

$$||\underline{r}||_2^2 = ||\underline{g_2}||_2^2,$$

[1] Essendo Q, per definizione, ortogonale, risulta $Q^{-1} = Q^T$.

[2] In quanto moltiplicare un vettore a sinistra per una matrice ortogonale non cambia la sua norma Euclidea: infatti, applicando la definizione di norma Euclidea su vettori reali si ha

$$||Qx||_2 = \sqrt{(Qx)^T(Qx)} = \sqrt{x^TQ^TQx} = \sqrt{x^Tx} = ||x||_2,$$

ricordando che, essendo Q ortogonale, $Q^TQ = Q^{-1}Q = I$.

[3] Considerando la partizione $\underline{g} = \begin{pmatrix}\underline{g_1}\\ \underline{g_2}\end{pmatrix}$, con $\underline{g_1} \in \mathbb{R}^n$, $\underline{g_2} \in \mathbb{R}^{m-n}$.

[4] In quanto essendo per definizione la norma Euclidea al quadrato la somma delle componenti di un vettore in valore assoluto elevate al quadrato, la norma Euclidea su un vettore può essere vista come somma delle norme Euclidee su le sue parti. Inoltre utilizzando nella definizione di norma il valore assoluto, si ha che $||-\underline{g_2}||_2 = ||\underline{g_2}||_2$.

in quanto soltanto il primo termine dipende dal vettore $\underline{x}$ e la minima norma Euclidea possibile è 0. Quindi si tratta di scegliere $\underline{x}$ come soluzione del sistema lineare

$$\hat{R}\underline{x} = \underline{g_1}. \tag{3.32}$$

Si può osservare che:

- il sistema lineare (3.32) ammette *un'unica soluzione*, essendo $\hat{R}$ nonsingolare per definizione. Questa soluzione è ovviamente la soluzione ai minimi quadrati del sistema lineare sovradeterminato (3.29);
- la matrice $\hat{R}$ è triangolare superiore, quindi il sistema lineare (3.32) è facilmente risolvibile;
- il fattore Q non è esplicitamente richiesto, una volta effettuato il prodotto $Q^T\underline{b}$ per ottenere il vettore $\underline{g}$.

3.8.1 Esistenza della fattorizzazione QR

Si consideri il seguente problema: dato un vettore $\underline{z} in \mathbb{R}^m$

$$\underline{z} = (z_1, \ldots, z_m)^T, \qquad \underline{z} neq 0,$$

si determini una matrice *ortogonale* H tale che

$$H\underline{z} = \alpha \underline{e_1}, \tag{3.33}$$

con $\alpha \in \mathbb{R}$. Si osserva che

$$\underline{z}^T H^T = (H\underline{z})^T = (\alpha \underline{e_1})^T = \alpha \underline{e_1}^T.$$

Avendo quindi

$$||\underline{z}||_2^2 = \underline{z}^T\underline{z} = \underline{z}^T \underbrace{H^T H}_{I} \underline{z} = \alpha^2 \underbrace{\underline{e_1}^T \underline{e_1}}_{1} = \alpha^2,$$

segue che

$$\alpha = \pm ||\underline{z}||_2. \tag{3.34}$$

Consideriamo allora la matrice H della forma

$$H = I - \frac{2}{\underline{v}^T\underline{v}} \underline{v}\,\underline{v}^T \qquad \underline{v} \neq 0, \tag{3.35}$$

dove il vettore $\underline{v} \in \mathbb{R}^m$ sarà scelto in modo da soddisfare la (3.33). Si osserva che la matrice H così definita è *simmetrica* per costruzione (I è simmetrica, $\frac{2}{\underline{v}^T\underline{v}}$ è uno scalare e $\underline{v}\,\underline{v}^T$ è una matrice simmetrica in quando risultato del prodotto di

un vettore per il suo trasposto). Inoltre H risulta essere anche *ortogonale*, infatti:

$$\begin{aligned} H^T H = H^2 \overset{1}{=} I^2 - \frac{4}{\underline{v}^T\underline{v}}\underline{v}\,\underline{v}^T + \frac{4}{(\underline{v}^T\underline{v})^2}(\underline{v}\,\underline{v}^T)^2 = \\ = I - \frac{4}{\underline{v}^T\underline{v}}\underline{v}\,\underline{v}^T + \frac{4}{(\underline{v}^T\underline{v})^2}\underline{v}(\underbrace{\underline{v}^T\underline{v}}_{\text{scalare}})\underline{v}^T = \\ = I - \frac{4}{\underline{v}^T\underline{v}}\underline{v}\,\underline{v}^T + \frac{4(\underline{v}^T\underline{v})}{(\underline{v}^T\underline{v})^2}\underline{v}\underline{v}^T = \\ = I - \frac{4}{\underline{v}^T\underline{v}}\underline{v}\,\underline{v}^T + \frac{4}{\underline{v}^T\underline{v}}\underline{v}\,\underline{v}^T = \\ = I = H^{-1}H. \end{aligned}$$

Dimostriamo adesso che scegliendo il vettore $\underline{v}$ come

$$\underline{v} = \underline{z} - \alpha\underline{e_1} \tag{3.36}$$

viene soddisfatta la (3.33):

$$\begin{aligned} H\underline{z} = (I - \frac{2}{\underline{v}^T\underline{v}}\underline{v}\,\underline{v}^T)\underline{z} = \\ = \underline{z} - \frac{2}{\underline{v}^T\underline{v}}\underline{v}(\underbrace{\underline{v}^T\underline{z}}_{\text{scalare}}) = \\ = \underline{z} - \frac{2}{\underline{v}^T\underline{v}}[(\underline{z} - \alpha\underline{e_1})^T\underline{z}]\underline{v} = \\ = \underline{z} - \frac{2}{\underline{v}^T\underline{v}}[(\underline{z}^T - \alpha\underline{e_1}^T)\underline{z}]\underline{v} = \\ = \underline{z} - \frac{2}{\underline{v}^T\underline{v}}(\underline{z}^T\underline{z} - \alpha\underline{e_1}^T\underline{z})\underline{v} = \\ = \underline{z} - \frac{2}{\underline{v}^T\underline{v}}(\underline{z}^T\underline{z} - \alpha z_1)(\underline{z} - \alpha\underline{e_1}) = \\ = \underline{z} - \frac{2}{\underline{v}^T\underline{v}}(\underline{z}^T\underline{z} - \alpha z_1)\underline{z} + \frac{2}{\underline{v}^T\underline{v}}(\underline{z}^T\underline{z} - \alpha z_1)\alpha\underline{e_1} = \\ = \left[1 - \frac{2}{\underline{v}^T\underline{v}}(\underline{z}^T\underline{z} - \alpha z_1)\right]\underline{z} + \frac{2}{\underline{v}^T\underline{v}}(\underline{z}^T\underline{z} - \alpha z_1)\alpha\underline{e_1} \equiv \\ \equiv (*). \end{aligned}$$

[1] Essendo H *simmetrica*, $H^T = H$.

Per la (3.34) si ha quindi che

$$\begin{aligned}
\frac{2}{\underline{v}^T\underline{v}}(\underline{z}^T\underline{z} - \alpha z_1) &= \frac{2(||\underline{z}||_2^2 - \alpha z_1)}{(\underline{z} - \alpha\underline{e_1})^T(\underline{z} - \alpha\underline{e_1})} = \\
&= \frac{2||\underline{z}||_2^2 - 2\alpha z_1}{(\underline{z}^T - \alpha\underline{e_1}^T)(\underline{z} - \alpha\underline{e_1})} = \\
&= \frac{2||\underline{z}||_2^2 - 2\alpha z_1}{\underline{z}^T\underline{z} - \alpha\underbrace{\underline{z}^T\underline{e_1}}_{z_1} - \alpha\underbrace{\underline{e_1}^T\underline{z}}_{z_1} + \alpha^2\underbrace{\underline{e_1}^T\underline{e_1}}_{1}} = \\
&= \frac{2||\underline{z}||_2^2 - 2\alpha z_1}{\underbrace{\underline{z}^T\underline{z}}_{||\underline{z}||_2^2} - 2\alpha z_1 + \underbrace{\alpha^2}_{||\underline{z}||_2^2}} = \\
&= \frac{2||\underline{z}||_2^2 - 2\alpha z_1}{2||\underline{z}||_2^2 - 2\alpha z_1} = \\
&= 1,
\end{aligned}$$

ottenendo, infine, la tesi:

$$\begin{aligned}
(*) &= (1-1)\underline{z} + 1 \cdot \alpha\underline{e_1} = \\
&= \alpha\underline{e_1}.
\end{aligned}$$

Definizione 3.10. *La matrice H definita dalle (3.34), (3.35) e (3.36), soddisfacente la (3.33), è detta* ***matrice elementare di Householder****, mentre il vettore v definito dalla (3.36) prende il nome di* ***vettore di Householder****.*

Osserviamo che la prima componente del vettore $\underline{v}$ è calcolata come $v_1 = z_1 - \alpha$, mentre le successive coincidono con le componenti analoghe in $\underline{z}$. Quindi, mentre in aritmetica esatta la scelta del segno di α è indifferente, in *aritmetica finita* è preferibile scegliere il segno di α in modo che i termini z_1 e $-\alpha$ siano di segno concorde, ovvero in modo che z_1 ed α abbiano segno opposto, affinché la somma algebrica $z_1 - \alpha$ risulti sempre *ben condizionata* (vedi Sezione 1.4). Una conseguenza di questa scelta è che, essendo per ipotesi $\underline{z} \neq 0$, la prima componente di $\underline{v}$ risulta sempre $v_1 \neq 0$.

Le matrici di Householder godono di una proprietà di *invarianza per scalamento* del corrispondente vettore di Householder (ovvero la matrice H non cambia se il vettore $\underline{v}$ viene scalato):

Teorema 3.10. *Sia H la matrice elementare di Householder della forma (3.35) definita dal vettore v.*
Allora, per ogni $\beta \neq 0$, la matrice di Householder definita dal vettore $\beta\underline{v}$ (ovvero il vettore $\underline{v}$ scalato di un fattore β) coincide con H.

Dimostrazione del Teorema 3.9

Utilizziamo una notazione, come per la fattorizzazione LU e la fattorizzazione con *Pivoting*, in cui l'indice tra parentesi in altro indica il passo più recente in cui l'elemento (o la matrice) corrispondente è stato modificato.
Poniamo innanzitutto

$$A = \begin{pmatrix} a_{11}^{(0)} & \dots & a_{1n}^{(0)} \\ \vdots & & \vdots \\ \vdots & & \vdots \\ a_{m1}^{(0)} & \dots & a_{mn}^{(0)} \end{pmatrix}$$

Considerando il primo vettore colonna di $A^{(0)}$, possiamo definire la matrice elementare di Householder H_1 che, moltiplicata a sinistra per tale vettore colonna, ne annulla tutte le componenti tranne la prima, che verrà sostituita da più o meno (a seconda dei segni) la norma Euclidea del vettore stesso. Ovvero definiamo la matrice $H_1 \in \mathbb{R}^{m \times m}$ tale che

$$H_1 \begin{pmatrix} a_{11}^{(0)} \\ \vdots \\ \vdots \\ a_{m1}^{(0)} \end{pmatrix} = \begin{pmatrix} a_{11}^{(1)} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \in \mathbb{R}^m,$$

dove $a_{11}^{(1)}$ è calcolato come

$$a_{11}^{(1)} = \pm \left\| \begin{pmatrix} a_{11}^{(0)} \\ \vdots \\ \vdots \\ a_{m1}^{(0)} \end{pmatrix} \right\|_2 .$$

Osserviamo che, avendo A rango massimo per ipotesi, sicuramente

$$a_{11}^{(1)} \neq 0,$$

in quanto altrimenti la prima colonna della matrice sarebbe nulla (essendo calcolato come la norma Euclidea sulla prima colonna).
Quindi moltiplicando H_1 a sinistra per la matrice $A^{(0)}$ si ottiene

$$H_1 A^{(0)} = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \dots & a_{1n}^{(1)} \\ 0 & a_{22}^{(1)} & \dots & a_{2n}^{(1)} \\ \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots \\ 0 & a_{m2}^{(1)} & \dots & a_{mn}^{(1)} \end{pmatrix} \equiv A^{(1)}.$$

Considerando adesso la porzione della seconda colonna che va dall'elemento diagonale $a_{22}^{(1)}$ in poi, definiamo la matrice elementare di Householder $H^{(2)} \in \mathbb{R}^{m-1 \times m-1}$

tale che

$$H^{(2)} \begin{pmatrix} a_{22}^{(1)} \\ \vdots \\ \vdots \\ a_{m2}^{(1)} \end{pmatrix} = \begin{pmatrix} a_{22}^{(2)} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \in \mathbb{R}^{m-1}.$$

Definendo quindi la matrice

$$H_2 \equiv \left(\begin{array}{c|c} 1 & \\ \hline & H^{(2)} \end{array} \right) \in \mathbb{R}^{m \times m},$$

che risulta ancora ortogonale e, lasciando invariata la prima riga (in quanto la sua prima riga coincide con quella della matrice identità), si comporta come una matrice elementare di Householder con la sottomatrice delimitata da $a_{22}^{(1)}$ e $a_{mn}^{(1)}$, ovvero otteniamo che

$$H_2 A^{(1)} = H_2 H_1 A = \begin{pmatrix} a_{11}^{(1)} & a_{12}^{(1)} & a_{13}^{(1)} & \dots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & a_{23}^{(2)} & \dots & a_{2n}^{(2)} \\ 0 & 0 & a_{33}^{(2)} & \dots & a_{3n}^{(2)} \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & a_{m3}^{(2)} & \dots & a_{mn}^{(2)} \end{pmatrix} \equiv A^{(2)}$$

Anche in questo caso osserviamo che sicuramente

$$a_{22}^{(2)} \neq 0,$$

se A ha rango massimo.
Reiterando il processo, dopo n passi avremo ottenuto che

$$A^{(n)} \equiv H_n \dots H_1 A = \begin{pmatrix} a_{11}^{(1)} & \dots & a_{1n}^{(1)} \\ 0 & \ddots & \vdots \\ \vdots & \ddots & a_{nn}^{(n)} \\ \vdots & & 0 \\ \vdots & & \vdots \\ 0 & \dots & 0 \end{pmatrix} \equiv R$$

Ponendo infine

$$H_n \dots H_1 \equiv Q^T,$$

si ottiene infine la fattorizzazione (3.31), infatti

$$Q^T A = R,$$
$$A = QR.$$

3.8.2 Il metodo di Householder

La dimostrazione appena svolta del Teorema (3.9) descrive il **metodo di fattorizzazione** QR di **Householder**.
Utilizzando la proprietà di scalamento dei vettori di Householder (vedi Teorema 3.10), possiamo scalarli in modo che la loro prima componente sia pari ad 1, e quindi nota. Quindi, per quanto riguarda l'occupazione di memoria, la matrice A può essere riscritta con l'informazione della sua fattorizzazione QR e in particolare con la porzione *triangolare superiore* di $\hat{R}$ e la parte significativa (ovvero escluso il primo elemento, il cui posto viene occupato dalla diagonale di $\hat{R}$) dei vettori di Householder generati ad ogni passo (non è necessario memorizzare le matrici di Householder, in quanto esse sono velocemente calcolabili a partire dai soli vettori). Il corrispondente costo computazionale si dimostra invece essere pari a

$$\approx \frac{2}{3}n^2(3m-n) \quad \textbf{flop}. \tag{3.37}$$

Osserviamo, infine, che il metodo di fattorizzazione QR di Householder può essere impiegato anche per la fattorizzazione di matrici nonsingolari, ovvero in cui $m = n$. Tuttavia si vede che in questo caso il costo risulta essere pari a

$$\begin{aligned}\approx \frac{2}{3}n^2(3n-n) = \frac{2}{3}n^2 \cdot 2n = \\ = \frac{4}{3}n^3 \quad \textbf{flop},\end{aligned}$$

ovvero circa il *doppio* rispetto al costo della fattorizzazione LU (con o senza *pivoting*). Quindi in presenza di matrici nonsingolari la fattorizzazione LU è sempre da preferirsi.

I Codici 3.11 e 3.12 implementano, rispettivamente, il metodo di Householder per la fattorizzazione QR di una matrice ed il metodo per la risoluzione di un sistema lineare sovradeterminato tramite fattorizzazione QR della matrice dei coefficienti.

Codice 3.11: Fattorizzazione QR di Householder di una matrice.

```
% A = fattorizzaQR(A)
% Fattorizzazione QR di Householder per matrici mxn con m>=n.
%
% Input:
%   -A: la matrice da fattorizzare QR.
% Output:
%   -A: la matrice riscritta con le informazioni dei fattori Q ed R.
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:15 CET

function [A] = fattorizzaQR(A)
    [m,n]=size(A);
```

```
    for i=1:n
        alfa = norm(A(i:m, i), 2);
        if alfa==0
            error('La matrice non ha rango massimo!');
        end
        if(A(i,i))>=0
            alfa = -alfa;
        end
        v1 = A(i,i)-alfa;
        A(i,i) = alfa;
        A(i+1:m, i) = A(i+1:m, i)/v1;
        beta = -v1/alfa;
        A(i:m, i+1:n) = A(i:m, i+1:n) -(beta*[1; A(i+1:m, i)])*([1 A(i
            +1:m, i)']*A(i:m, i+1:n));
    end
end
```

Codice 3.12: Risoluzione di un sistema lineare sovradeterminato tramite fattorizzazione QR della matrice dei coefficienti.

```
% b = risolviSistemaQR(A,b)
% Risoluzione di un sistema lineare sovredeterminato del tipo Ax=b
% tramite fattorizzazione QR di Householder della matrice dei
% coefficienti.
%
% Input:
%   -A: matrice dei coefficienti;
%   -b: vettore dei termini noti.
% Output:
%   -b: vettore delle soluzioni del sistema lineare sovradeterminato.
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:23 CET

function [b] = risolviSistemaQR(A,b)
    [m,n] = size(A);
    A = fattorizzaQR(A);
    Qt=eye(m);
    for i=1:n
        Qt = [eye(i-1) zeros(i-1, m-i+1); zeros(i-1, m-i+1)' (eye(m-i
            +1)-(2/norm([1; A(i+1:m, i)], 2)^2)*([1; A(i+1:m, i)]*[1 A(
            i+1:m, i)']))]*Qt;
    end
    b = triangolareSupCol(triu(A(1:n, :)), Qt(1:n, :)*b);
end
```

3.9 Cenni sulla risoluzione di sistemi nonlineari

Per la risoluzione di sistemi nonlineari, ovvero del tipo

$$F(\underline{x}) = \underline{0}, \qquad F : \Omega \subseteq \mathbb{R}^n \to \mathbb{R}^n, \tag{3.38}$$

con F costituita dalle *funzioni componenti*

$$F(\underline{x}) = \begin{pmatrix} f_1(\underline{x}) \\ \vdots \\ f_n(\underline{x}) \end{pmatrix}, \quad f_i : \Omega \subseteq \mathbb{R}^n \to \mathbb{R},$$

ed $\underline{x}$ il vettore delle incognite che risolvano il sistema (3.38)

$$x = \begin{pmatrix} x_1 \\ \dots \\ x_n \end{pmatrix} \in \mathbb{R}^n,$$

si utilizza il **metodo di Newton**, ovvero un *metodo iterativo* definito da

$$\underline{x}^{k+1} = \underline{x}^k - J_F(\underline{x}^k)^{-1} F(\underline{x}^k), \quad k = 0, 1, \dots, \tag{3.39}$$

partendo da un'approssimazione $\underline{x}^0$ assegnata. $J_F(\underline{x})$ indica la **matrice Jacobiana**, ovvero la matrice delle derivate parziali:

$$J_F(\underline{x}) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(\underline{x}) & \dots & \frac{\partial f_1}{\partial x_n}(\underline{x}) \\ \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1}(\underline{x}) & \dots & \frac{\partial f_n}{\partial x_n}(\underline{x}) \end{pmatrix}.$$

Si osserva che l'iterazione (3.39) è definita se $J_F(\underline{x}^k)$ è *nonsingolare*, condizione che riterremo sempre sempre sempre soddisfatta in un opportuno intorno della soluzione $\underline{x}^*$. Inoltre si dimostra che, se F ha derivate continue e l'approssimazione iniziale $\underline{x}^0$ appartiene ad un opportuno intorno della soluzione $\underline{x}^*$, il metodo di Newton **converge quadraticamente** (godee quindi di proprietà di *convergenza locale*).
In pratica, ogni passo dell'iterazione (3.39) corrisponde a risolvere il seguente sistema lineare:

$$\begin{cases} J_F(\underline{x}^k)\underline{d}^k = -F(\underline{x}^k) \\ \underline{x}^{k+1} = \underline{x}^k + \underline{d}^k \end{cases}, \tag{3.40}$$

dove il vettore temporaneo delle incognite $\underline{d}^k$ viene utilizzato per poter spezzare l'iterazione (3.39) in due equazioni. Quindi la risoluzione del sistema nonlineare (3.38) si riconduce alla risoluzione di una successione di sistemi lineari. Ovviamente, per ogni sistema lineare della successione sarà necessario fattorizzare LU la matrice Jacobiana.
Per ridurre il costo computazionale ad ogni passo, si può implementare una modifica equivalente al metodo delle corde visto in Sezione 2.8, ovvero utilizzando un'approssimazione dello Jacobiano corrispondente alla matrice Jacobiana calcolata sull'approssimazione iniziale $\underline{x}^0$:

$$\begin{cases} J_F(\underline{x}^0)\underline{d}^k = -F(\underline{x}^k) \\ \underline{x}^{k+1} = \underline{x}^k + \underline{d}^k \end{cases}, \tag{3.41}$$

dovendo quindi fattorizzare la matrice $J_F(\underline{y}^0)$ una sola volta. Si dimostra che in questo caso la convergenza del metodo è soltanto **lineare**.
Di seguito, proponiamo un'implementazione in MATLAB del metodo delle corde per la risoluzione di sistemi nonlineari:

Codice 3.13: Metodo delle corde per la risoluzione di sistemi nonlineari.

```
% y = CordeNonLin( f, J, y, tol )
% Risoluzione di sistemi non lineari con il metodo delle corde.
%
% Input :
%   -f: funzione;
%   -J: Jacobiano;
%   -y: punto iniziale;
%   -tol: tolleranza.
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:04 CET

function [ y ] = CordeNonLin ( f, J, y, tol )

app =[y (1); y (2)];
while ( norm (app)>tol)
app = J\(- feval (f,y));
y (3:4) = y (1:2);
y (1:2) = y (1:2)+ app;
end
```

Esercizi

Esercizio 3.1. *Scrivere gli Algoritmi per la risoluzione di sistemi triangolari inferiori e superiori, con accesso agli elementi per riga e per colonna, in modo da controllare che la matrice dei coefficienti sia nonsingolare.*

Soluzione.
Sappiamo che una matrice è *nonsingolare* se e solo se il suo determinante è diverso da zero. Sappiamo inoltre (è facilmente verificabile) che il determinante di una matrice triangolare, inferiore o superiore, coincide con il prodotto dei suoi elementi diagonali, in quanto gli elementi strettamente triangolari (inferiori o superiori) non contribuiscono al calcolo del determinante (in quanto durante il calcolo si annullano con gli altri elementi nelle rispettive posizioni simmetriche).
Per controllare allora che una matrice triangolare sia *nonsingolare* si può o controllare a priori che tutti gli elementi diagonali siano diversi da zero, oppure controllare elemento per elemento durante il calcolo della soluzione del sistema, senza aggiungere complessità computazionale. Nelle soluzioni proposte (vedi Codici 3.1, 3.2, 3.3 e 3.4, pp. 66-68) si è optato per il controllo elemento per elemento durante il normale calcolo della soluzione.

Esercizio 3.2. *Dimostrare che la somma ed il prodotto di matrici triangolari inferiori (superiori), è una matrice triangolare inferiore (superiore)*

Soluzione.
Supponiamo che le matrici operande siano $A, B \in \mathbb{R}^{n\times n}$ triangolari inferiori (superiori) con $a_{ij} = b_{ij} = 0$ per $i < j$ $(i > j)$.

Somma
Sia $C = A + B$, ovvero $c_{ij} = a_{ij} + b_{ij}$ per $1 \leq i, j \leq n$. Risulta allora, per $i < j$ $(i > j)$

$$c_{ij} = a_{ij} + b_{ij} = 0 + 0 = 0,$$

ovvero C è una matrice triangolare inferiore (superiore).

Prodotto
Sia $C = AB$, ovvero $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$ per $1 \leq i, j \leq n$. Si osserva che, per matrici triangolari inferiori, $a_ik = 0$ se $k > i$, quindi gli ultimi $n - i$ termini della sommatoria saranno sicuramente nulli. Inoltre si può osservare che anche $b_{kj} = 0$ se $k < j$, quindi i primi $j - 1$ termini della sommatoria risulteranno anch'essi nulli. Deduciamo allora che per calcolare il valore di c_{ij} basta considerare soltanto i termini della sommatoria che vanno da j ad i, ovvero $c_{ij} = \sum_{k=j}^{i} a_{ik}b_{kj}$. Risulta allora chiaro che, se $i < j$, la sommatoria non sarà composta di alcun termine e quindi $c_{ij} = 0$, ovvero C è una matrice triangolare inferiore.
Se la matrice è triangolare superiore il discorso è analogo a quello appena fatto: $a_{ik} = 0$ se $k < i$, $b_{kj} = 0$ se $k > j$, quindi $c_{ij} = \sum_{k=i}^{j} a_{ik}b_{kj}$, ovvero $c_{ij} = 0$ se $i > j$, cioè C risulta triangolare superiore.

Esercizio 3.3. *Dimostrare che il prodotto di due matrici triangolari inferiori (superiori) a diagonale unitaria è a sua volta una matrice triangolare inferiore (superiore) a diagonale unitaria*

Soluzione.
Dall'Esercizio 3.2 risulta che per $A, B \in \mathbb{R}^{n\times n}$ triangolari inferiori (rispettivamente, superiori) si ha $C = AB$ con $c_{ij} = \sum_{k=j}^{i} a_{ik}b_{kj}$ (rispettivamente, $c_{ij} = \sum_{k=i}^{j} a_{ik}b_{kj}$). Se inoltre A e B hanno diagonale unitaria, allora gli elementi diagonali $(j = i)$ della matrice C saranno calcolati come

$$c_{ii} = \sum_{k=i}^{i} a_{ik}b_{ki} = a_{ii}b_{ii} = 1 \cdots 1 = 1.$$

Quindi C risulta essere triangolare inferiore (superiore), per quando dimostrato nell'Esercizio 3.2, e a diagonale unitaria.

Esercizio 3.4. *Dimostrare che la matrice inversa di una matrice triangolare inferiore (superiore) è a sua volta triangolare inferiore (superiore). Dimostrare inoltre che, se la matrice ha diagonale unitaria, tale è anche la diagonale della sua inversa.*

Soluzione.
Sia $A =\in \mathbb{R}^{n\times n}$ una matrice triangolare inferiore non singolare ($a_{i,i} \neq 0$, $i = 1,\dots,n$), si procede per induzione su n:

- $n = 1$: $A = a_{1,1}$, la matrice è uno scalare e la proprietà è banalmente verificata.
- $n-1 \Rightarrow n$: $A = \left(\begin{array}{c|c} A_{n-1} & \underline{0} \\ \hline \underline{a}^T & a_{n,n} \end{array}\right)$, utilizzando l'ipotesi induttiva per A_{n-1} si ha $A^{-1} = \left(\begin{array}{c|c} A_{n-1}^{-1} & \underline{0} \\ \hline \underline{v}^T & w \end{array}\right)$; si verifica che $AA^{-1} = I$:

$$
\begin{aligned}
AA^{-1} &= \left(\begin{array}{c|c} A_{n-1} & \underline{0} \\ \hline \underline{a}^T & a_{n,n} \end{array}\right)\left(\begin{array}{c|c} A_{n-1}^{-1} & \underline{0} \\ \hline \underline{v}^T & w \end{array}\right) = \\
&= \left(\begin{array}{c|c} A_{n-1}A_{n-1}^{-1} + \underline{0}\,\underline{v}^T & A_{n-1}\underline{0} + \underline{0}w \\ \hline \underline{a}^T A_{n-1}^{-1} + a_{n,n}\underline{v}^T & \underline{a}^T\underline{0} + a_{n,n}w \end{array}\right) = \\
&= \left(\begin{array}{c|c} I & \underline{0} \\ \hline \underline{a}^T A_{n-1}^{-1} + a_{n,n}\underline{v}^T & a_{n,n}w \end{array}\right) = I \Rightarrow \\
&\Rightarrow \left\{\begin{array}{l} \underline{a}^T A_{n-1}^{-1} + a_{n,n}\underline{v} = \underline{0} \\ a_{n,n}w = 1 \end{array}\right. \Rightarrow \left\{\begin{array}{l} \underline{v}^T = -\frac{\underline{a}^T A_{n-1}^{-1}}{a_{n,n}} \\ w = \frac{1}{a_{n,n}} \end{array}\right. .
\end{aligned}
$$

Se la matrice ha diagonale unitaria ($a_{i,i} = 1$, $i = 1,\dots,n$), è tale anche la sua inversa poiché $w = \frac{1}{a_{n,n}} = 1$. Analoghe considerazioni valgono per le matrici triangolari superiori.

Esercizio 3.5. *Dimostrare i Lemmi 3.2 e 3.3.*

Soluzione.
Lemma 3.2
Una matrice triangolare è non singolare se e solo se tutti i suoi minori principali sono non nulli.
Sia A una matrice triangolare, risulta $det(A) = \prod_{i=1}^{n} a_{i,i}$.

- $\Rightarrow$ A è non singolare, quindi $det(A) \neq 0$ ovvero $\prod_{i=1}^{n} a_{i,i} \neq 0$ cioè $a_{i,i} \neq 0$ per $i = 1,\dots,n$. Il minore di ordine k è $det(A_k) = \prod_{i=1}^{k} a_{i,i} \neq 0$ poiché $a_{i,i} \neq 0$ per $i = 1,\dots,k$ per $k = 1,\dots,k$.
- $\Leftarrow$ Tutti i minori principali di A sono non nulli, quindi anche $det(A) = det(A_n) \neq 0$ quindi la matrice A è non singolare.

Lemma 3.3
Nella fattorizzazione LU, il minore di ordine k in A coincide con il minore di ordine k in U ovvero $det(A_k) = det(U_k)$.
La matrice A_k si ottiene estraendo da A le prime k righe e le prime k colonne, quindi $A_k = \begin{pmatrix} I_k & O \end{pmatrix} A \begin{pmatrix} I_k \\ O^T \end{pmatrix}$ con $I_k \in \mathbb{R}^{k\times k}$ e $O \in \mathbb{R}^{k\times n-k}$. Ricordando $A = LU$, risulta $A_k = \begin{pmatrix} I_k & O \end{pmatrix} LU \begin{pmatrix} I_k \\ O^T \end{pmatrix} = \left[\begin{pmatrix} I_k & O \end{pmatrix} L\right] \left[U \begin{pmatrix} I_k \\ O^T \end{pmatrix}\right] = \begin{pmatrix} L_k & O \end{pmatrix} \begin{pmatrix} U_k \\ O^T \end{pmatrix} = L_k U_k$; per le proprietà del determinate, si ha $det(A_K) = det(L_k U_K) = det(L_k)det(U_K) = det(U_K)$ perché $det(L_k) = 1$ in quanto L è una matrice triangolare inferiore a diagonale unitaria.

Esercizio 3.6. *Dimostrare che il numero di* `flop` *richiesti dall'Algoritmo di fattorizzazione LU è dato da (3.8) (considerare che* $\sum_{i=1}^{n-1} i^2 = \frac{n(n-1)(2n-1)}{6}$*).*

Soluzione.
L'algoritmo esegue $n-i$ passi; al passo i-esimo si eseguono $n-i$ divisioni per calcolare il vettore di Gauss e 2 operazioni (una sottrazione e una moltiplicazione) sulla sottomatrice di $(n-i)^2$ elementi. Il costo è quindi $\sum_{i=1}^{n-1} \left(n - i + 2(n-i)^2\right) = \sum_{i=1}^{n-1} \left(k + k^2\right)$; considerando solo i termini quadratici, risultano circa $\sum_{i=1}^{n-1} \left(k^2\right) = 2\frac{n(n-1)(2n-1)}{6} \approx \frac{2}{3}n^3$ flops

Esercizio 3.7. *Scrivere una* `function` MATLAB *che implementi efficientemente l'Algoritmo di fattorizzazione LU di una matrice.*

Soluzione.
Per l'implementazione in MATLAB della fattorizzazione LU si veda il Codice 3.5 a pagina 78. Nella soluzione proposta la matrice A viene riscritta con le informazioni dei fattori L ed U e restituita come output.

Esercizio 3.8. *Scrivere una* `function` MATLAB *che, avendo in ingresso la matrice A riscritta dall'algoritmo di fattorizzazione LU ed un vettore* $\underline{x}$ *contenente i termini noti del sistema lineare (3.1), ne calcoli efficientemente la soluzione.*

Soluzione.
Per l'implementazione in MATLAB della risoluzione di un sistema lineare tramite fattorizzazione LU della matrice dei coefficienti si veda il Codice 3.6 a pagina 78. Nella soluzione proposta si è deciso di passare come input la matrice dei coefficienti ancora da fattorizzare. Quindi la `function risolviSistemaLU` fattorizza la

matrice A e quindi risolve in ordine i due sistemi lineari triangolari, riscrivendo il vettore dei termini noti b con le soluzioni, prima intermedie, poi finali. Inoltre, a scapito di eleganza e leggibilità del codice, si è deciso di non estrarre da A i fattori L ed U in modo esplicito, così da non occupare inutilmente posizioni di memoria che si erano risparmiate riscrivendo A con L ed U.

Esercizio 3.9. *Dimostrare i Lemmi 3.4 e 3.5.*

Soluzione.
Lemma 3.4
Supponiamo $A \in \mathbb{R}^{n\times n}$ a diagonale dominante per righe e consideriamo A^k la sottomatrice principale di ordine k di A per $k = 1, \ldots, n$. Risulta $\left|a^k_{i,i}\right| > \sum_{j=1, j\neq i}^{k} |a_{i,j}|$ in quanto si considerano solo le prime k colonne della matrice A e quindi A^k è a diagonale dominante per righe. Analogamente per le matrici a diagonale dominante per colonne: in questo caso si sommano solo i termini delle prime k righe.

Lemma 3.5
Nel trasporre la matrice, gli elementi diagonali non subiscono modifiche mentre l'elemento $a_{i,j}$ diviene l'elemento $a_{j,i}$ per $i = 1, \ldots, n$, $j = 1, \ldots, n$, $i \neq j$. Dunque se in A $|a_{i,i}| > \sum_{i\neq j} |a_{i,j}|$, in A^T $|a_{i,i}| > \sum_{i\neq j} |a_{j,i}|$ per $i = 1, \ldots, n$ ovvero se la matrice è a digaonale dominante per righe la sua trasposta è a diagonale dominante per colonne e, viceversa, se è la diagonale dominante per colonne la sua trasposta lo è per righe.

Esercizio 3.10. *Dimostrare la parte del Teorema 3.7 in cui la (3.9) implica che A è sdp.*

Soluzione.
Una matrice A è sdp se e solo se $A = LDL^T$.

- $A = LDL^T \Rightarrow A$ è sdp:
  - A è simmetrica: $A^T = (LDL^T)^T = (L^T)^T D^T L^T = LDL^T = A$ poichè $D^T = D$ in quanto diagonale.
  - A è definita positiva: $\forall \underline{x} \in \mathbb{R}, \neq \underline{0}$: $\underline{x}^T A\underline{x} = \underline{x}^T LDL^T \underline{x} = (\underline{x}^T L)D(L^T \underline{x}) = (L^T \underline{x})^T D(L^T \underline{x}) = \underline{y}^T D\underline{y}$ con $\underline{y} \neq \underline{0}$ poichè L non singolare e $\underline{x} \neq \underline{0}$. Risulta $\underline{x}^T A\underline{x} = \underline{y}^T D\underline{y} = \sum_{i=1}^{n} d_i y_i^2 > 0$ in quanto D ha elementi diagonali positivi.

Esercizio 3.11. *Dimostrare che, se A è nonsingolare, le matrici A^TA e AA^T sono sdp*

Soluzione.

- A^TA è sdp:
  - simmetrica: $A^TA = (A^TA)^T = A^T(A^T)^T = A^TA$;
  - definita positiva: $\forall \underline{x} \neq \underline{0}$ risulta $\underline{x}^TA^TA\underline{x} = (\underline{x}^TA^T)(A\underline{x}) = (A\underline{x})^T(A\underline{x}) = \underline{y}^T\underline{y} = ||\underline{y}||_2^2 > 0$ in quanto $\underline{y} \neq \underline{0}$ poichè A non singolare ed $\underline{x} \neq \underline{0}$.
- AA^T è sdp:
  - simmetrica: $AA^T = (AA^T)^T = (A^T)^TA^T = AA^T$;
  - definita positiva: $\forall \underline{x} \neq \underline{0}$ risulta $\underline{x}^TAA^T\underline{x} = (\underline{x}^TA)(A^T\underline{x}) = (A^T\underline{x})^T(A^T\underline{x}) = \underline{y}^T\underline{y} = ||\underline{y}||_2^2 > 0$ in quanto $\underline{y} \neq \underline{0}$ poichè A non singolare ed $\underline{x} \neq \underline{0}$.

Esercizio 3.12. *Dimostrare che se $A \in \mathbb{R}^{m\times n}$, con $m \geq n = rank(A)$, allora la matrice A^TA è sdp*

Soluzione.

- simmetrica: $A^TA = (A^TA)^T = A^T(A^T)^T = A^TA$;
- definita positiva: $\forall \underline{x} \neq \underline{0}$, $\underline{x}^TA^TA\underline{x} = (\underline{x}^TA^T)(A\underline{x}) = (A\underline{x})^T(A\underline{x}) = \underline{y}^T\underline{y} = ||\underline{y}||_2^2 \geq 0$; al più è 0 se $\underline{y} = \underline{0}$ ma ciò non può verificarsi perchè il rango di A è massimo ovvero $dim(null(A)) = 0$ e quindi $\underline{y} \notin null(A)$ a meno di $\underline{x} = \underline{0}$.

Esercizio 3.13. *Data una matrice $A \in \mathbb{R}^{n\times n}$, dimostrare che essa può essere scritta come*

$$A = \frac{1}{2}(A + A^T) + \frac{1}{2}(A - A^T) \equiv A_s + A_a,$$

*dove $A_s = A_s^T$ è detta **parte simmetrica** di A, mentre $A_a = -A_a^T$ è detta **parte antisimmetrica** di A. Dimostrare inoltre che, dato un generico vettore $\underline{x} \in \mathbb{R}^n$, risulta*

$$\underline{x}^TA\underline{x} = \underline{x}^TA_s\underline{x}.$$

Soluzione.
Per la proprietà distrubutiva e la somma di matrici, banalmente, si ha $A = \frac{1}{2}(A + A^T) + \frac{1}{2}(A - A^T) = \frac{1}{2}A + \frac{1}{2}A^T + \frac{1}{2}A - \frac{1}{2}A^T$. Si prova che

- $A_s = A_s^T$: $\frac{1}{2}(A + A^T) = \left[\frac{1}{2}(A + A^T)\right]^T = \frac{1}{2}(A + A^T)^T = \frac{1}{2}(A^T + (A^T)^T) = \frac{1}{2}(A^T + A) = \frac{1}{2}(A + A^T)$
- $A_a = -A_a^T$: $\frac{1}{2}(A - A^T) = -\left[\frac{1}{2}(A - A^T)\right]^T = -\frac{1}{2}(A - A^T)^T = -\frac{1}{2}(A^T - (A^T)^T) = -\frac{1}{2}(A^T - A) = \frac{1}{2}(A - A^T)$

Il fatto che $\underline{x}^T A \underline{x} = \underline{x}^T A_s \underline{x}$ implica $\underline{x}^T A_a \underline{x} = 0$ in quanto $\underline{x}^T A \underline{x} = \underline{x}^T A_s \underline{x} + \underline{x}^T A_a \underline{x}$. Si calcola $\underline{x}^T A_a \underline{x} = \underline{x}^T \left(\frac{1}{2}(A - A^T)\right) \underline{x} = \frac{1}{2}\left(\underline{x}^T A \underline{x} - \underline{x}^T A^T \underline{x}\right) = \frac{1}{2}\left(\underline{x}^T A \underline{x} - (A\underline{x})^T \underline{x}\right) = \frac{1}{2}\left(\underline{x}^T \underline{y} - \underline{x}^T \underline{y}\right)$; siccome il prodotto di due vettori è commutativo, $\underline{x}^T \underline{y} = \underline{x}^T \underline{y}$ e quindi $\underline{x}^T A_a \underline{x} = 0$.

Esercizio 3.14. *Dimostrare la consistenza delle formule (3.10) e (3.11). (Suggerimento: ragionare per induzione; infatti, per $j = 1$ esse sono definite).*

Soluzione.
Si procede per induzione su j:

- $j = 1$:
  - $d_1 = a_{1,1} - \sum_{k=1}^{0} l_{1,k}^2 d_k = a_{1,1}$, ben definita;
  - $l_{i,1} = \frac{a_{i,1} - \sum_{k=1}^{0} l_{i,k} d_k l_{1,k}}{d_1} = \frac{a_{i,1}}{d_1}$ per $i = j + 1, \ldots, n$, ben definita in quanto l'elemento d_1 è già stato calcolato; si completa la prima colonna di L.
- $j - 1 \Rightarrow j$:
  - $d_j = a_{j,j} - \sum_{k=1}^{j-1} l_{j,k}^2 d_k$, ben definita in quanto vengono utilizzati i primi $j - 1$ elementi della riga j-esima di L che sono stati calcolati sotto l'ipotesi induttiva;
  - $l_{i,j} = \dfrac{a_{i,j} - \sum_{k=1}^{j-1} l_{i,k} d_k l_{j,k}}{d_j}$ per $i = j + 1, \ldots, n$, si utilizzano i termini $d_1, \ldots, d_{j-1}$ e $l_{j,1}, \ldots, l_{j,j+1}$ calcolati sotto l'ipotesi induttiva.

Esercizio 3.15. *Dimostrare che il numero di* `flop` *richiesti dall'Algoritmo di fattorizzazione LDL^T è dato da (3.12) (vedi anche l'Esercizio 3.6).*

Soluzione.
Il contributo maggiore al costo computazionale è dato dal calcolo di L: l'algoritmo esegue esegue $j - 1$ somme di 2 prodotti, una sottrazione ed una divisione per un costo di $2(j - 1) + 2 = 2j$ `flop`. Poiché L è triangolare si dovrà eseguire tale calcolo $n - j$ volte per ciascuna colonna; il costo totale dell'algoritmo è quindi di $\sum_{j=1}^{n} 2j(n - j) = 2n \sum_{j=1}^{n} j - 2 \sum_{j=1}^{n} j^2 = 2n \frac{n(n+1)}{2} - 2 \frac{n(n+1)(2n+1)}{6} \approx n^3 - \frac{2}{3} n^3 = \frac{1}{3} n^3$ `flop`.

Esercizio 3.16. *Scrivere una* `function` MATLAB *che implementi efficientemente l'algoritmo di fattorizzazione* LDL^T *per matrici sdp.*

Soluzione.
Per l'implementazione dell'Algoritmo di fattorizzazione LDL^T di una matrice *sdp* si consulti il Codice 3.7 a pagina 83.

Esercizio 3.17. *Scrivere una* `function` MATLAB *che, avendo in ingresso la matrice* A *prodotta dalla precedente* `function`*, contenente la fattorizzazione* LDL^T *della matrice sdp originaria, ed un vettore di termini noti,* $\underline{x}$*, calcoli efficientemente la soluzione del corrispondente sistema lineare.*

Soluzione.
Per l'implementazione in MATLAB della risoluzione di un sistema lineare tramite fattorizzazione LDL^T della matrice *sdp* dei coefficienti si veda il 3.8 a pagina 84. Nella soluzione proposta si è deciso di passare come input la matrice dei coefficienti ancora da fattorizzare. Quindi la `function risolviSistemaLDLt` fattorizza la matrice A e quindi risolve in ordine i tre sistemi lineari triangolari, riscrivendo il vettore dei termini noti b con le soluzioni, prima intermedie, poi finali. Inoltre, a scapito di eleganza e leggibilità del codice, si è deciso di non estrarre da A i fattori L e D in modo esplicito, così da non occupare inutilmente posizioni di memoria che si erano risparmiate riscrivendo A con L e D.

Esercizio 3.18. *Utilizzare la* `function` *dell'Esercizio 3.16 per verificare che la matrice*

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

non è sdp.

Soluzione.
Eseguendo il codice relativo, dove sostanzialmente viene richiamata la `function fattorizzaLDLt` con argomento la matrice A specificata, vediamo che, durante l'esecuzione dell'algoritmo di fattorizzazione, viene generato un messaggio d'errore, il quale informa che la matrice non è *sdp*, seppur simmetrica.

Riferimenti MATLAB
Codice 3.14 (pagina 123)

Esercizio 3.19. *Dimostrare che, al passo i-esimo di eliminazione di Gauss con pivoting parziale, si ha (vedi (3.14)) $a_{k_i i}^{(i)} \neq 0$, se A è nonsingolare.*

Soluzione.
Al passo j-esimo, l'elemento $a_{k,i}^{(j)}$ è dato da $\max_{k\geq 1} |a_{k,j}^{(j)}|$; se $a_{k,i}^{(j)} = 0$ allora tutti gli elementi della colonna j-esima sarebbero zero contraddicendo il fatto che la matrice A sia nonsingolare.

Esercizio 3.20. *Con riferimento alla seguente matrice*

$$A = \begin{pmatrix} 0 & \dots & \dots & 0 & 1 \\ 2 & 0 & \dots & \dots & 0 \\ & 3 & \ddots & & \vdots \\ & & \ddots & \ddots & \vdots \\ & & & n & 0 \end{pmatrix},$$

quale è la matrice di permutazione P che rende PA fattorizzabile LU? Chi sono, in tal caso, i fattori L e U?

Soluzione.
La matrice ha tutti i minori principali nulli, eccetto l'ultimo. La matrice di permutazione che rende A fattorizzabile LU è

$$P = \begin{pmatrix} 0 & \dots & \dots & 0 & 1 \\ 1 & 0 & \dots & \dots & 0 \\ & 1 & 0 & \dots & 0 \\ & & \ddots & 0 & 0 \\ & & & \ddots & 1 \end{pmatrix}$$

infatti,

$$PA = \begin{pmatrix} 1 & & & & \\ & 2 & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & n \end{pmatrix} = LU$$

con i fattori $L = I_n$ e $U \equiv PA$.

Esercizio 3.21. *Scrivere una* `function` MATLAB *che implementi efficientemente l'algoritmo di fattorizzazione LU con pivoting parziale.*

Soluzione.
Consultare il Codice 3.9 a pagina 90, per la `function fattorizzaLUpivoting` relativa alla fattorizzazione LU di una matrice nonsingloare con pivoting parziale. Nella soluzione proposta, l'output generato dall'Algoritmo è composto dalla matrice di partenza riscritta con le informazioni dei fattori L ed U ed il vettore contente le informazioni della matrice di permutazione.

Esercizio 3.22. *Scrivere una `function` MATLAB che, avendo in ingresso la matrice A prodotta dalla precedente `function`, contenente la fattorizzazione LU della matrice permutata, il vettore $\underline{p}$ contenente l'informazione relativa alla corrispondente matrice di permutazione, ed un vettore di termini noti, $\underline{x}$, calcoli efficientemente la soluzione del corrispondente sistema lineare.*

Soluzione.
Per l'implementazione in MATLAB della risoluzione di un sistema lineare tramite fattorizzazione LU con pivoting parziale della matrice nonsingolare dei coefficienti si veda il Codice 3.10 a pagina 91. Nella soluzione proposta si è deciso di passare come input la matrice dei coefficienti ancora da fattorizzare. Quindi la `function risolviSistemaLUpivoting` fattorizza la matrice A, costruisce la matrice di permutazione a partire dal vettore p e quindi risolve in ordine i due sistemi lineari triangolari, riscrivendo il vettore dei termini noti b con le soluzioni, prima intermedie, poi finali. Inoltre, a scapito di eleganza e leggibilità del codice, si è deciso di non estrarre da A i fattori L ed U in modo esplicito, così da non occupare inutilmente posizioni di memoria che si erano risparmiate riscrivendo A con L ed U.

Esercizio 3.23. *Costruire alcuni esempi di applicazione delle `function` degli Esercizi 3.21 e 3.22.*

Soluzione.
Gli esempi costruiti per testare la `function risolviSistemaLUpivoting` sono i seguenti:

- Esempio 1:

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 4 & 0 \end{pmatrix} \underline{x} = \begin{pmatrix} 4 \\ 58 \\ 2 \\ 7 \end{pmatrix}$$

$$\underline{x} = \begin{pmatrix} 29.0000 \\ 0.6667 \\ 1.7500 \\ 4.0000 \end{pmatrix}$$

  errore commesso: 0.

- Esempio 2:

$$\begin{pmatrix} -23 & 5 & -21 & 8 \\ 0 & 0 & 5 & 7 \\ 1 & 54 & 7 & 9 \\ 0 & -8 & 12 & 4 \end{pmatrix} \underline{x} = \begin{pmatrix} 10 \\ -4 \\ 76 \\ 23 \end{pmatrix}$$

$$\underline{x} = \begin{pmatrix} -5.0867 \\ 1.5532 \\ 4.1247 \\ -3.5176 \end{pmatrix}$$

errore commesso: $\approx 10^{-14}$.

Riferimenti MATLAB
Codice 3.15 (pagina 123)

Esercizio 3.24. *Le seguenti istruzioni* MATLAB *sono equivalenti a risolvere il dato sistema* 2×2 *con l'utilizzo del pivoting e non, rispettivamente. Spiegarne il differente risultato ottenuto. Concludere che l'utilizzo del pivoting migliora, in generale, la prognosi degli errori in aritmetica finita.*

```
A = [eps 1;1 0], b = [1; 1/4]
A\b
L = [1 0; 1/eps 1], U = [eps 1; 0 -1/eps]
L*U-A
U\(L\b)
```

Soluzione.
Nella riga 2 si risolve il sistema $A\underline{x} = \underline{b}$ con pivoting, in particolare:

$$P = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \; PA = \begin{pmatrix} 1 & 0 \\ \varepsilon & 1 \end{pmatrix} = LU \text{ con } L \equiv PA \text{ e } U \equiv I_2$$

$$\text{da cui } \begin{pmatrix} 1 & 0 \\ \varepsilon & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \frac{1}{4} \\ 1 \end{pmatrix} \Rightarrow \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \frac{1}{4} \\ 1 - \frac{1}{4}\varepsilon \end{pmatrix} \approx \begin{pmatrix} \frac{1}{4} \\ 1 \end{pmatrix}.$$

Nella riga 5 si risolve il sistema mediante fattorizzazione LU:

$$A = LU = \begin{pmatrix} 1 & 0 \\ \frac{1}{\varepsilon} & 1 \end{pmatrix} \begin{pmatrix} \varepsilon & 1 \\ 0 & -\frac{1}{\varepsilon} \end{pmatrix} \text{ da cui}$$

$$\begin{pmatrix} 1 & 0 \\ \frac{1}{\varepsilon} & 1 \end{pmatrix} \begin{pmatrix} z \\ w \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{1}{4} \end{pmatrix} \Rightarrow \begin{pmatrix} z \\ w \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{1}{4} - \frac{1}{\varepsilon} \end{pmatrix} \approx \begin{pmatrix} 1 \\ -\frac{1}{\varepsilon} \end{pmatrix} \text{ e}$$

$$\begin{pmatrix} \varepsilon & 1 \\ 0 & -\frac{1}{\varepsilon} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ -\frac{1}{\varepsilon} \end{pmatrix} \Rightarrow \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \frac{1-1}{\varepsilon} \\ 1 \end{pmatrix} \approx \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

Le approssimazioni sono dovute al calcolo in aritmetica finita; in ogni caso, l'utilizzo del pivoting produce un risultato più coerente con quello analitico che risulta essere

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \frac{1}{4} \\ 1 - \frac{1}{4}\varepsilon \end{pmatrix}.$$

Riferimenti MATLAB
Codice 3.16 (pagina 124)

Esercizio 3.25. *Si consideri la seguente matrice bidiagonale inferiore*

$$A = \begin{pmatrix} 1 & & & \\ 100 & 1 & & \\ & \ddots & \ddots & \\ & & 100 & 1 \end{pmatrix}_{10\times 10}.$$

Calcolare $k_\infty(A)$. Confrontate il risultato con quello fornito dalla `function cond` *di* MATLAB. *Dimostrare, e verificare, che $k_\infty(A) = k_1(A)$.*

Soluzione.
La matrice A è una matrice bidiagonale di Toeplitz, la sua inversa è

$$A^{-1} = \begin{pmatrix} 1 & & & & \\ -10^2 & \ddots & & & \\ 10^4 & \ddots & \ddots & & \\ \vdots & \ddots & \ddots & \ddots & \\ (-1)^{n-1}10^{2n-2} & \dots & 10^4 & -10^2 & 1 \end{pmatrix}_{10\times 10}.$$

Risulta

$$||A||_\infty = \max_{i=1,\dots,n} \sum_{j=1}^{n} |a_{i,j}| = 101,$$

$$||A^{-1}||_\infty = \max_{i=1,\dots,n} \sum_{j=1}^{n} |a_{i,j}| = \sum_{j=1}^{n} |a_{n,j}| = \sum_{s=0}^{n-1} 10^{2s} = \frac{10^{2n}-1}{10^2-1} = \frac{10^{2n}-1}{99}$$

$$\text{quindi } \kappa_\infty(A) = ||A||_\infty ||A^{-1}||_\infty = 101\frac{10^{2n}-1}{99} > 10^{2n},$$

nel caso $n = 10$, si ha $\kappa_\infty(A) > 10^{20}$ quindi il problema è malcondizionato. Su tale matrice, la `function cond` restituisce `Inf`.
La norma ∞ su una matrice è la somma massima delle righe, la norma 1 è la

somma massima delle colonne; nella matrice A tutte le colonne, come tutte le righe, hanno somma 101 quindi $||A||_\infty = ||A||_1 = 101$. Nella matrice A^{-1}, la norma ∞ considera l'n-esima riga mentre la norma 1 la prima colonna, in ogni caso, $||A||_\infty = ||A||_1 = \frac{10^{20}-1}{99}$. Quindi $\kappa_\infty(A) = \kappa_1(A) > 10^{20}$.

Riferimenti MATLAB
Codice 3.17 (pagina 124)

Esercizio 3.26. *Si consideri i seguenti vettori di* $\mathbb{R}^{10}$,

$$\underline{b} = \begin{pmatrix} 1 \\ 101 \\ \vdots \\ 101 \end{pmatrix}, \quad \underline{c} = 0.1 \cdot \begin{pmatrix} 1 \\ 101 \\ \vdots \\ 101 \end{pmatrix},$$

ed i seguenti sistemi lineari

$$A\underline{x} = \underline{b}, \qquad A\underline{y} = \underline{c},$$

in cui A è la matrice definita nel precedente Esercizio 3.25. Verificare che le soluzioni di questi sistemi lineari sono, rispettivamente, date da:

$$\underline{x} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}, \qquad \underline{y} = \begin{pmatrix} 0.1 \\ \vdots \\ 0.1 \end{pmatrix}.$$

Confrontare questi vettori con quelli calcolato dalle seguenti due serie di istruzioni MATLAB,

```
b=[1 101*ones(1,9)]';
x(1)=b(1); for i=2:10, x(i)=b(i)-100*x(i-1), end
x=x(:)

c=0.1+[1 101*ones(1,9)]';
y(1)=c(1); for i=2:10, y(i)=c(i)-100*y(i-1); end
y=y(:)
```

che implementano, rispettivamente, le risoluzioni dei due sistemi lineari. Spiegare i risultati ottenuti, alla luce di quanto visto in Sezione 3.7.

Soluzione.
Come mostrato nell'Esercizio 3.25, la matrice A risulta malcondizionata con $\kappa(A) \approx 10^{20}$; considerando il vettore $\underline{c}$ come una perturbazione di $\underline{b}$ si ha

$$\Delta\underline{b} = \underline{c} - \underline{b} = \begin{pmatrix} -0.9 \\ -90.9 \\ \vdots \\ -90.9 \end{pmatrix},$$

segue $\frac{||\Delta \underline{b}||}{||\underline{b}||} \approx \frac{\sqrt{0.9+9\cdot 90.9^2}}{\sqrt{1+9\cdot 101^2}} \approx 1$. Quindi

$$\frac{||\Delta \underline{x}||}{||\underline{x}||} \leq \kappa(A)\left(\frac{||\Delta \underline{b}||}{||\underline{b}||} + \frac{||\Delta A||}{||A||}\right) = \kappa(A)\frac{||\Delta \underline{b}||}{||\underline{b}||} \approx 10^{20}$$

ovvero a fronte di una perturbazione del vettore $\underline{b}$ di 0.1, si ha un errore sul risultato dell'ordine di 10^{20}.

Riferimenti MATLAB
Codice 3.18 (pagina 124)

Esercizio 3.27. *Dimostrare che il numero di* `flop` *richiesti dall'Algoritmo di fattorizzazione QR di Householder è dato da (3.37) (vedi anche l'Esercizio 3.6).*

Soluzione.
Ad ogni iterazione si eseguono:

- riga `12`, norma su $m-i+1$ elementi
  - $m-i+1$ quadrati,
  - $m-i$ somme,
  - 1 radice quadrata;
- riga `15`, calcolo di `v1`
  - 1 sottrazione;
- riga `17`, calcolo del vettore di Householder
  - $m-(i+1)+1 = m-i$ divisioni;
- riga `18`, calcolo di `beta`
  - 1 divisione;
- riga `19`, modifica della restante porzione della matrice
  - $(m-i+1)(n-(i+1)-1) = (m-i+1)(n-1)$ sottrazioni,
  - $1+m-(i+1)+1 = m-i+1$ moltiplicazioni,
  - $(1+(m-(i+1)+1)+(m-i+1))(n-(i+1)+1) = 2(n-i)(m-i+1)$ moltiplicazioni,
  - $(m-i+1)(n-i)$ moltiplicazioni.

In totale, risultano

$$4(m-i+1)(n-i+1) = 4[mn+m+n+1+i^2-(m+n+2)i]$$

`flop` ad iterazione; l'algoritmo esegue n iterazioni, il costo totale è di:

$$\begin{aligned}
&\sum_{i=0}^{n} 4[mn+m+n+1+i^2-(m+n+2)i] = \\
&= 4\left[\sum_{i=0}^{n}(mn+m+n+1)+\sum_{i=0}^{n} i^2-(m+n+2)\sum_{i=0}^{n} i\right] = \\
&= 4\left[n(mn+m+n+1)+\frac{n(n+1)(2n+1)}{6}-(m+n+2)\frac{n(n+1)}{2}\right] \approx \\
&\approx 4\left[mn^2+\frac{n^3}{3}-(m+n)\frac{n^2}{2}\right] = \\
&= 2mn^2-\frac{2}{3}n^3 = \frac{2}{3}n^2(3m-n)\ \texttt{flop}.
\end{aligned}$$

Esercizio 3.28. *Definendo il vettore (vedi (3.36)) $\underline{\hat{v}} = \frac{\underline{v}}{v_1}$, verificare che* `beta`*, come definito nel seguente algoritmo per la fattorizzazione QR di Householder, corrisponde alla quantità $\frac{2}{\underline{\hat{v}}^T\underline{\hat{v}}}$:*

```
for i=1:n
    alfa = norm( A(i:m,i) );
    if alfa==0, error('la matrice A non ha rango massimo'), end
    if A(i,i)>=0, alfa = -alfa; end
    v1 = A(i,i) -alfa;
    A(i,i) = alfa; A(i+1:m,i) = A(i+1:m,i)/v1;
    beta = -v1/alfa;
    A(i:m,i+1:n) = A(i:m,i+1:n) -(beta*[1; A(i+1:m,i)])*([1 A(i+1:m,i)
        ']*A(i:m,i+1:n));
end
```

.

Soluzione.
Dall'algoritmo risulta

$$\texttt{beta} = -\frac{\texttt{v1}}{\texttt{alpha}} = -\frac{a_{1,1}-\alpha}{\alpha}$$

che è equivalente a

$$\frac{2}{\underline{\hat{v}}^T\underline{\hat{v}}} = \frac{2}{\frac{\underline{v}}{v_1}\frac{\underline{v}^T}{v_1}} = \frac{2v_1^2}{\underline{v}^T\underline{v}} = \frac{2v_1^2}{(\underline{z}^T - \alpha\underline{e}_1^T)(\underline{z} - \alpha\underline{e}_1)} =$$
$$= \frac{2v_1^2}{\underline{z}^T\underline{z} - \underline{z}^T\alpha\underline{e}_1 - \underline{z}\alpha\underline{e}_1^T + \alpha^2\underline{e}_1^T\underline{e}_1} = \frac{2v_1^2}{||\underline{z}||_2^2 - \alpha a_{1,1} - \alpha a_{1,1} + \alpha^2} =$$
$$= \frac{v_1^2}{\alpha^2 - \alpha a_{1,1}} = \frac{(\alpha a_{1,1})^2}{\alpha(\alpha - a_{1,1})} = -\frac{a_{1,1} - \alpha}{\alpha}$$

Esercizio 3.29. *Scrivere una* `function` MATLAB *che implementi efficientemente l'algoritmo di fattorizzazione QR, mediante il metodo di Householder (vedi l'Algoritmo descritto nell'Esercizio precedente).*

Soluzione.
Nella soluzione proposta (Codice 3.11 a pagina 101), la matrice A viene riscritta con le informazioni della matrice R e della matrice Q: in particolare, la porzione triangolare superiore di $\hat{R}$ e la parte significativa dei vettori di Householder grazie ai quali si può successivamente ricostruire la matrice Q.

Esercizio 3.30. *Scrivere una* `function` MATLAB *che, avendo in ingresso la matrice A prodotta dalla* `function` *del precedente Esercizio, contenente la fattorizzazione QR della matrice originaria, e un corrispondente vettore di termini noti* $\underline{b}$*, calcoli efficientemente la soluzione del sistema lineare sovradeterminato (3.29).*

Soluzione.
Nel Codice 3.12 a pagina 102, la matrice A in input viene innanzitutto fattorizzata QR, quindi viene ricostruita la matrice Q^T a partire dalle informazioni presenti nella matrice A riscritta sui vettori di Householder. Viene allora moltiplicata Q^T per il vettore $\underline{b}$ $(= \underline{g})$, per risolvere infine il sistema lineare $\hat{R}\underline{x} = \underline{g_1}$, dove $\hat{R}$ viene estratto come parte triangolare superiore di A e $\underline{g_1}$ è il vettore formato dalle prime n componenti di $\underline{g}$.

Esercizio 3.31. *Utilizzare le* `function` *degli Esercizi 3.29 e 3.30 per calcolare la soluzione ai minimi quadrati di (3.29), ed il corrispondente residuo, nel caso in cui*

$$A = \begin{pmatrix} 3 & 2 & 1 \\ 1 & 2 & 3 \\ 1 & 2 & 1 \\ 2 & 1 & 2 \end{pmatrix}, \qquad \underline{b} = \begin{pmatrix} 10 \\ 10 \\ 10 \\ 10 \end{pmatrix}.$$

Soluzione.
Eseguando la `function risolviSistemaQR` con A e $\underline{b}$ come input, si ottiene che il risultato ai minimi quadrati del sistema $A\underline{x} = \underline{b}$ è

$$\underline{x} = \begin{pmatrix} 1.4 \\ 2.8 \\ 1.4 \end{pmatrix}.$$

Inoltre si ricava che il vettore residuo minimo ottenuto, e la relativa norma Euclidea, sono:

$$\underline{r} \equiv A\underline{x} - \underline{b} = \begin{pmatrix} 1.2 \\ 1.2 \\ -1.6 \\ -1.6 \end{pmatrix}, \qquad ||\underline{r}||_2^2 = 8.$$

Riferimenti MATLAB
Codice 3.19 (pagina 125)

Esercizio 3.32 (Retta ai minimi quadrati)**.** *Calcolare i coefficienti della equazione della retta*

$$r(x) = a_1 x + a_2,$$

che meglio approssima i dati prodotti dalle seguenti istruzioni MATLAB*:*

```
x    = linspace(0,10,101)'
gamma    = 0.1
y    = 10*x+5+(sin(x*pi))*gamma
```

Riformulare il problema come minimizzazione della norma Euclidea di un corrispondente vettore residuo. Calcolare la soluzione utilizzando le `function` *sviluppate negli Esercizi 3.29 e 3.30, e confrontarla con quella ottenuta dalle seguenti istruuzioni* MATLAB*:*

```
A = [x x.^0]
a = A\y
r = A*a-y
```

Confrontare i risultati che si ottengono per i seguenti valori del parametro `gamma`*:*

$$0.5, 0.1, 0.05, 0.01, 0.005, 0.001.$$

Quale è la soluzione che si ottiene nel limite `gamma` $\to 0$*?*

Soluzione.
Poichè $-1 < sin(\pi x) < 1$ e $\gamma < 1$ si ha

$$y(x) = 10x + 5 + \sin(\pi x)\gamma \approx 10x + 5 = r(x)$$

quindi

$$\underline{a} = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} 10 \\ 5 \end{pmatrix}.$$

Il problema può essere formulato come minimizzazione del vettore residuo $r(x) - y(x)$, si utilizza il metodo di Householder per risolvere il sistema:

$$\begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_{101} & 1 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix} = \begin{pmatrix} y(x_1) \\ y(x_2) \\ \vdots \\ y(x_{101}) \end{pmatrix}.$$

Nella tabella sono riportati i valori di $\underline{a}$ e il resiuduo r al variare di γ:

γ	$\underline{a}$	r
0.5	$\begin{pmatrix} 9.9816 \\ 5.0919 \end{pmatrix}$	3.4943
0.1	$\begin{pmatrix} 9.9636 \\ 5.0184 \end{pmatrix}$	0.6989
0.05	$\begin{pmatrix} 9.9982 \\ 5.0092 \end{pmatrix}$	0.3494
0.01	$\begin{pmatrix} 9.9996 \\ 5.0018 \end{pmatrix}$	0.0699
0.005	$\begin{pmatrix} 9.9998 \\ 5.0009 \end{pmatrix}$	0.0349
0.001	$\begin{pmatrix} 10.0000 \\ 5.0002 \end{pmatrix}$	0.0070

Si nota che, per $\gamma \to 0$, $r \to 0$ infatti, $\sin(\pi x)\gamma \to 0$ e dunque anche $y(x) \to 10x + 5 = r(x)$ con $\underline{a} = \begin{pmatrix} 5 \\ 2 \end{pmatrix}$.

Riferimenti MATLAB
Codice 3.20 (pagina 125)

Esercizio 3.33. *Determinare il punto di minimo della funzione $f(x_1, x_2) = x_1^4 + x_1(x_1 + x_2) + (1 - x_2)^2$, utilizzando il metodo di Newton (3.40) per calcolarne il punto stazionario.*

Soluzione.
Un punto stazionario $(\hat{x_1}, \hat{x_2})$ è tale per cui $f'(\hat{x_1}, \hat{x_2}) = 0$. Si ottiene quindi il sistema non lineare:

$$F(\underline{y}) = \underline{0} \text{ con } F = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{pmatrix} = \begin{pmatrix} 4x_1^3 + 2x_1 + x_2 \\ x_1 + 2x_2 + 2 \end{pmatrix}.$$

Applicando il metodo di Newton si va a risolvere:

$$J_F(\underline{y}^{(k)})\underline{d}^{(k)} = -F(\underline{y}^{(k)}), \quad \underline{y}^{(k+1)} = \underline{y}^{(k)} + \underline{d}^{(k)} \text{ con } J_F = \begin{pmatrix} 12x_1^2 + 2 & 1 \\ 1 & 2 \end{pmatrix}.$$

Inseriti i valori iniziali $x_1^{(0)}, x_2^{(0)}$, il numero massimo d'iteazioni e la tolleranza (`8-11`), per ogni iterazione, si va a valutare lo jacobiano (`19`) e a risolvere il sistema mediante fattorizzazione LU con pivoting (`20-21`). Al termine delle iterazioni, si mostra il punto di minimo ed il valore ivi assunto dalla funzione (`24-25`). Risulta

$$\min f(x_1, x_2) \approx -0.2573 \text{ in } (0.4398, -1.2199).$$

Riferimenti MATLAB
Codice 3.21 (pagina 126)

Esercizio 3.34. *Uno dei metodi di base per la risoluzione di equazioni differenziali ordinarie, $y'(t) = f(t, y(t))$, $t \in [t_0, T]$, $y(t_0) = y_0$ (problema di Cauchy di prim'ordine), è il metodo di Eulero implicito,*

$$y_n = y_{n-1} + hf_n, \qquad n = 1, 2, \ldots, N \equiv \frac{T - t_0}{h},$$

in cui $y_n \approx y(t_n)$, $f_n = f(t_n, y_n)$, $t_n = t_0 + nh$. Utilizzare questo metodo nel caso in cui $t_0 = 0$, $T = 10$, $N = 100$, $y_0 = (1, 2)^T$ e, se $y = (x_1, x_2)^T$, $f(t, y) = (-10^3 x_1 + \sin x_1 \cos x_2, -2x_2 + \sin x_1 \cos x_2)^T$. Utilizzare il metodo (3.41) per la risoluzione dei sistemi nonlineari richiesti.

Soluzione.
Definita $y'(t) = f(t, y(t))$, $y_0 \equiv y(t_0)$, si approssima $y'(t)$ con il seguente rapporto incrementale all'indietro:

$$y'(t_i) = \frac{y(t_i) - y(t_{i-1})}{h} \equiv y_i$$

da cui segue

$$\frac{y_i - y_{i-1}}{h} = f(t_i, y_i) \quad \Rightarrow \quad \frac{y_i - y_{i-1}}{h} = f_i \quad \Rightarrow \quad y_i - y_{i-1} - hf_i = 0.$$

Nell'esercizio si ha

$$y_0 = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

e

$$F_i = \begin{pmatrix} x_1^{(i)} - x_1^{(i-1)} - h(-10^3 x_1 + \sin x_1 \cos x_2) \\ x_2^{(i)} - x_2^{(i-1)} - h(-2x_2 + \sin x_1 \cos x_2) \end{pmatrix}$$

Dove $x_1^{(i-1)}$ e $x_2^{(i-1)}$ ovvero il valore di y al passo $(i-1)$-esimo sono costanti note. Ad ognuno degli N passi eseguiamo l'equivalente del metodo delle corde per i sistemi non lineari

$$J_F(z_0)d_k = -F(z_k), \quad z_{k+1} = z_k + d_k, \quad k = 1, 2, \ldots;$$

calcoliamo lo Jacobiano con $h = 0.1$:

$$J_F = \begin{pmatrix} 1 + h(10^3 - \cos x_1 \cos x_2) & h \sin x_1 \sin x_2 \\ -h(\cos x_1 \cos x_2) & 1 + h(2 + \sin x_1 \sin x_2) \end{pmatrix} \quad \Rightarrow$$

$$J_F \equiv \begin{pmatrix} 101 - \alpha & \beta \\ -\alpha & 1.2 + \beta \end{pmatrix} \quad \text{dove} \quad \alpha = \frac{\cos 1 \cos 2}{10} \text{ e } \beta = \frac{\sin 1 \sin 2}{10}.$$

Ad ogni iterazione si invoca la `function CordeNonLin` per ottenere l'approssimazione (entro una tolleranza ε prefissata) di y_i a partire dal vettore y_{i-1} ove calcoliamo lo Jacobiano. Risulta $y_{100} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$ per $\varepsilon = 10^{-5}$.

Riferimenti MATLAB
Codice 3.22 (pagina 126)

Codice degli esercizi

Codice 3.14: Esercizio 3.18.

```
% Esercizio 3.18
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:11 CET

A = [1 1 1 1; 1 2 2 2; 1 2 1 1; 1 2 1 2]
A = fattorizzaLDLt(A)
```

Codice 3.15: Esercizio 3.23.

```
% Esercizio 3.23
%
% Autore: Tommaso Papini,
```

```
% Ultima modifica: 4 Novembre 2012, 11:11 CET

A = [0 0 0 1; 2 0 0 0; 0 3 0 0; 0 0 4 0], b=[4; 58; 2; 7]
x = risolviSistemaLUpivoting(A, b)
A*x-b

A = [-23 5 -21 8; 0 0 5 7; 1 54 7 9; 0 -8 12 4], b=[10; -4; 76; 23]
x = risolviSistemaLUpivoting(A, b)
A*x-b
```

Codice 3.16: Esercizio 3.24.

```
% Esercizio 3.24
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:11 CET

format short e
A=[ eps , 1; 1, 0], b =[1; 1/4]
A\b
L=[1 , 0; 1/ eps , 1], U=[ eps , 1; 0, -1/ eps]
L*U-A
U\(L\b)
```

Codice 3.17: Esercizio 3.25.

```
% Esercizio 3.25
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:11 CET

format short e
A=eye(10)+100*[ zeros(1, 10); eye(9) zeros(1, 9)']
inv(A)
cond (A)
```

Codice 3.18: Esercizio 3.26.

```
% Esercizio 3.26
%
```

```
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:11 CET

format short e
A=eye (10)+100*[ zeros(1 ,10); eye (9) zeros(1 ,9)'];
b=[1 101* ones(1 ,9)]';
x (1)= b (1); for i=2:10 , x(i)=b(i) -100*x(i -1); end
x=x(:)

c =0.1*[1 101* ones(1 ,9)]';
y (1)= c (1); for i=2:10 , y(i)=c(i) -100*y(i -1); end
y=y(:)

norm (A)* norm (inv(A))* norm (c-b)/ norm (b)
```

Codice 3.19: Esercizio 3.31.

```
% Esercizio 3.31
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:11 CET

A = [3 2 1; 1 2 3; 1 2 1; 2 1 2]
b = [10; 10; 10; 10]
x = risolviSistemaQR(A,b)
r = A*x-b
disp('norma di r: '), norm(r, 2)^2
```

Codice 3.20: Esercizio 3.32.

```
% Esercizio 3.32
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:12 CET

format short

x = linspace(0 ,10 ,101)';
A=[x, x .^0];
gamma = [0.5 0.1 0.05 0.01 0.005 0.001];

for i =1:6
y =10* x +5+( sin(x*pi ))* gamma (i);
a = A\y;
r = A*a-y;
```

```
fprintf ('gamma = %4.3 f',gamma (i))
fprintf ('\n\ tSenza fattorizzazione : '), a, norm (r)
[a,r]= risolviSistemaQR ( fattorizzaQR (A),y);
fprintf ('\ tCon fattorizzazione QR: '), a, r
end
```

Codice 3.21: Esercizio 3.33.

```
% Esercizio 3.33
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:12 CET

format short
x (1)= input ('Valore iniziale x1 = ');
x (2)= input ('Valore iniziale x2 = ');
imax = input ('Numero massimo d''iterazioni : imax = ');
tolx = input ('Tolleranza : tolx = ');

F= inline ('[4*x (1)^3+2* x (1)+ x(2) , x (1)+2* x (2)+2] ');

i=0;
while (i< imax )&&( norm (x- xold )> tolx )
i=i+1;
xold =x;
J =[12* x (1)^2+2 , 1; 1, 2];
[J, p] = fattorizzaLUpivoting (J);
x=x+ risolviSistemaLUpivoting (J, p, -feval (F,x));
end
disp ('Punto di minimo : '), disp (x)
disp ('Valore assunto : '), disp (x (1)^4+ x (1)*( x (1)+ x (2))+(1+ x
    (2))^2)
```

Codice 3.22: Esercizio 3.34.

```
% Esercizio 3.34
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:12 CET

format long
x =[1;2;1;2];
f= inline ('[x(1) -x (3)+(10^3* x(1) - sin(x (1))* cos(x (2)))/10; x(2)
     -x (4)+(2* x(2) - sin(x (1))* cos(x (2)))/10]');

```

```
for i =1:100
alpha = cos(x (1))* cos(x (2))/10;
beta = sin(x (1))* sin(x (2))/10;
J = [101 - alpha , beta ; -alpha , 1.2+ beta ];
y = CordeNonLin ( f, J, x, 10^ -5);
x = x +0.1* y;
end
[x (1); x (2)]
```

Capitolo 4

Approssimazione di funzioni

Indice

Questo capitolo tratta dell'approssimazione di funzioni tramite polinomi. Spesso infatti è necessario approssimare, all'interno di un calcolatore, la forma funzionale di una certa funzione $f : [a, b] \subset \mathbb{R} \to \mathbb{R}$ in quanto la sua forma funzionale potrebbe essere troppo complessa o addirittura non nota.
In generale, assumeremo di essere a conoscenza di un insieme di $n + 1$ ascisse distinte,

$$a \leq x_0 < x_1 < \cdots < x_n \leq b, \tag{4.1}$$

dette **ascisse di interpolazione**, e dei valori assunti dalla funzione in tali punti.

4.1 Interpolazione polinomiale

Siano note le coppie

$$(x_i, f_i), \quad i = 0, 1, \ldots, n, \quad f_i \equiv f(x_i),$$

ricerchiamo un **polinomio interpolante** la funzione $f(x)$ sulle ascisse (4.1), detto $p(x) \in \Pi_n$, che assuma gli stessi valori di f sulle ascisse di interpolazione (4.1), ovvero tale che

$$p(x_i) = f_i, \qquad i = 0, 1, \ldots, n. \tag{4.2}$$

Teorema 4.1. *Date le ascisse (4.1), esiste ed è unico il polinomio $p(x) \in \Pi_n$ interpolante f sulle ascisse (4.1).*

Per determinare tale polinomio interpolante, essendo unico, si tratterà di determinare gli $n+1$ coefficienti dei monomi presenti nel polinomio, che equivale a risolvere il seguente sistema lineare:

$$V\underline{a} = \underline{f}, \tag{4.3}$$

$$\begin{pmatrix} x_0^0 & x_0^1 & \ldots & x_0^n \\ x_1^0 & x_1^1 & \ldots & x_1^n \\ \vdots & \vdots & & \vdots \\ x_n^0 & x_n^1 & \ldots & x_n^n \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} f_0 \\ f_1 \\ \vdots \\ f_n \end{pmatrix}.$$

Infatti un generico polinomio avrà la forma

$$p(x) = \sum_{k=0}^{n} a_k x^k, \tag{4.4}$$

e la matrice V, che è la trasposta della **matrice di Vandermonde**, risulta essere nonsingolare. Tuttavia, una delle proprietà note della matrice di Vandermonde, è che essa diviene rapidamente *malcondizionata* al crescere di n, ovvero del numero di ascisse di interpolazione. Quindi esamineremo metodi alternativi per la determinazione del polinomio $p(x)$ interpolante $f(x)$.

4.2 Forma di Lagrange e forma di Newton

Il problema (4.3) deriva dall'aver scelto, come base per lo spazio vettoriale dei polinomi Π_n, la regolare **base delle potenze**, $\{x^0, x^1, \ldots, x^n\}$. Per riformulare il problema, ottenendo proprietà più favorevoli, è necessario quindi cambiare base per Π_n.

Base di Lagrange:
La *base di Lagrange* è costituita da i seguenti polinomi:

$$L_{k,n}(x) = \prod_{\substack{j=0 \\ j \neq k}}^{n} \frac{x - x_j}{x_k - x_j}, \qquad k = 0, 1, \ldots, n, \tag{4.5}$$

dove $L_{k,n}(x)$ indica il k-esimo polinomio della base di Lagrange di grado n calcolato in x. Osserviamo che, essendo le ascisse (4.1) tutte distinte tra loro, i polinomi (4.5) risultano essere ben definiti.

Lemma 4.1. *Dati i polinomi di Lagrange (4.5) definiti sulle ascisse (4.1),*

$$L_{k,n}(x_i) = \begin{cases} 1, & \text{se } k = i, \\ 0, & \text{se } k \neq i. \end{cases}$$

Inoltre risulta che:

- *essi hanno grado esatto n ed il* ***coefficiente principale*** *(ovvero il coefficiente del termine di grado massimo) di $L_{k,n}(x)$ è*

$$c_{k,n} = \frac{1}{\prod_{\substack{j=0 \\ j\neq k}}^{n}(x_k - x_j)}, \qquad k = 0, 1, \ldots, n; \tag{4.6}$$

- *essi sono linearmente indipendenti tra loro, costituendo quindi una base per Π_n.*

Teorema 4.2 (Forma di Lagrange)**.** *Il polinomio*

$$p(x) = \sum_{k=0}^{n} f_k L_{k,n}(x) \tag{4.7}$$

appartiene a Π_n e soddisfa i vincoli di interpolazione (4.2), ovvero interpola $f(x)$ sulle ascisse (4.1).

Quindi la forma di Lagrange (4.7) consente di ottenere in modo immediato il polinomio interpolante di grado richiesto (infatti i coefficienti del polinomio interpolante, rispetto alla base di Lagrange, risultano essere esattamente i valori $\{f_i\}$). Tuttavia questa apparente semplicità formale non si presta a soddisfare un requisito, spesso richiesto in questo tipo di problema, che è quello di generare il polinomio interpolante in maniera *incrementale*. In particolare vorremmo poter ottenere, dal polinomio $p_r(x)$, già calcolato sulle ascisse $x_0, x_1, \ldots x_r$, il polinomio $p_{r+1}(x)$ di grado successivo, calcolato sulle ascisse precedenti più un'ulteriore ascissa x_{r+1}, in maniera relativamente semplice.
Definiamo, allora, a questo scopo, un'ulteriore base di Π_n.

Base di Newton:
La *base di Newton* è definita da i seguenti polinomi:

$$\begin{aligned} &w_0(x) \equiv 1, \\ &w_{k+1}(x) = (x - x_k)w_k(x), \qquad k = 0, 1, 2, \ldots. \end{aligned} \tag{4.8}$$

Si verifica facilmente che (vedi –>refes4.4) la base di Newton gode delle seguenti proprietà:

Lemma 4.2. *Con riferimento ai polinomi (4.8) della base di Newton, per ogni $k = 0, 1, 2, \ldots$, si ha che:*

- $w_k(x) \in \Pi'_k$, *dove* Π'_k *indica lo spazio vettoriale dei* ***polinomi monici*** *di grado* k *(un polinomio si dice* ***monico*** *se il suo coefficiente principale è pari a* 1*),*
- $w_{k+1}(x) = \prod_{j=0}^{k}(x - x_j)$,
- $w_{k+1}(x_j) = 0$, *per* $i \leq k$,
- $w_0(x), \ldots, w_k(x)$ *costituiscono una base per* Π_k.

Teorema 4.3 (Forma di Newton)**.** *La famiglia dei polinomi di Newton interpolanti* $\{p_r(x)\}_{r=0}^{n}$ *tali che:*

$$p_r(x) \in \Pi_r, \qquad p_r(x_i) = f_i, \qquad i = 0, \ldots, r,$$

è generata ricorsivamente come segue:

$$\begin{aligned} p_0(x) &= f_0 w_0(x), \\ p_r(x) &= p_{r-1}(x) + f[x_0, x_1, \ldots, x_r] w_r(x), \qquad r = 1, \ldots, n, \end{aligned} \tag{4.9}$$

dove il fattore $f[x_0, x_1, \ldots, x_r]$ *è detto* ***differenza divisa*** *di ordine* r *della funzione* $f(x)$ *sulle ascisse* $x_0, x_1, \ldots, x_r$*, e vale*

$$f[x_0, x_1, \ldots, x_r] = \sum_{k=0}^{r} \frac{f_k}{\prod_{\substack{j=0 \\ j \neq k}}^{r}(x_k - x_j)}. \tag{4.10}$$

Codice 4.1: Calcolo di una differenza divisa.

```
% dd = differenzaDivisa(ptx, fi)
% Calcola la differenza divisa relativa ad un set di ascisse.
%
% Input:
%   -ptx: vettore contenente le ascisse su cui calcolare la differenza
%   divisa;
%   -fi: vettore contenente i valori assunti dalla funzione in
%   corrispondenza dei punti in ptx.
% Output:
%   -dd: il valore della differenza divisa risultante.
%
% Autore: Tommaso Papini,
% Ultima modifica: 24 Ottobre 2012, 12:36 CEST.

function [dd] = differenzaDivisa(ptx, fi)
    dd = 0;
    for i=1:length(ptx)
        prod = 1;
        for j=1:length(ptx)
            if j~=i
                prod = prod*(ptx(i)-ptx(j));
            end
```

```
        end
        dd = dd+fi(i)/prod;
    end
end
```

Data la natura iterativa della generazione dei polinomi (4.9) di Newton possiamo utilizzare la seguente forma della forma di Newton soddisfacente le (4.2):

$$p(x) \equiv p_n(x) = \sum_{r=0}^{n} f[x_0, \ldots, x_r] w_r(x), \tag{4.11}$$

infatti $f[x_0] = f_0$.
Possiamo quindi calcolare la forma di Newton del polinomio interpolante tramite il seguente Codice:

Codice 4.2: Forma di Newton del polinomio interpolante.

```
% p = formaNewton(ptx, fi)
% Calcolo dell'espressione del polinomio interpolante una funzione in
% forma di Newton.
%
% Input:
%   -ptx: vettore contenente le ascisse di interpolazione;
%   -fi: i valori assunti dalla funzione da interpolare in
%   corrispondenza delle ascisse di interpolazione in ptx.
% Output:
%   -p: espressione (come funzione inline) del polinomio interpolante.
%
% Autore: Tommaso Papini,
% Ultima modifica: 28 Ottobre 2012, 19:00 CET

function [p] = formaNewton(ptx, fi)
    n = length(ptx)-1;
    dd = differenzeDivise(ptx, fi);
    syms x;
    p = dd(1);
    for i=2:n+1
        prod = dd(i);
        for j=1:i-1
            prod = prod*(x-ptx(j));
        end
        p = p + prod;
    end
    p = inline(p);
end
```

Si osservi che il denominatore del k-esimo polinomio della base di Lagrange di grado n (4.5) e del corrispondente coefficiente principale (4.6) è dato da $w'_{n+1}(x)$, mentre il denominatore di ogni termine delle differenze divise (4.10) è dato da

$w'_{r+1}(x_k)$. Ad esempio, per n (o k) uguale a 4 abbiamo

$$\begin{aligned} w_4(x) &= (x-x_3)(x-x_2)(x-x_1)(x-x_0), \\ w'_4(x) &= \partial[(x-x_3)(x-x_2)]\cdot(x-x_1)(x-x_0) + (x-x_3)(x-x_2)\cdot\partial[(x-x_1)(x-x_2)] = \\ &= (2x-x_3-x_2)(x-x_1)(x-x_0) + (x-x_3)(x-x_2)(2x-x_1-x_0), \end{aligned}$$

e per, ad esempio, $k=1$ otteniamo

$$\begin{aligned} w'_4(x_1) &= (2x_1-x_3-x_2)(x_1-x_1)(x_1-x_0) + (x_1-x_3)(x_1-x_2)(2x_1-x_1-x_0) = \\ &= (x_1-x_3)(x_1-x_2)(x_1-x_0). \end{aligned}$$

Quindi risulta che le differenze divise (4.10) costituiscono i coefficienti del polinomio interpolante (4.2) rispetto alla base di Newton (ma non, ad esempio, rispetto alla base delle potenze!).

Teorema 4.4. *Le differenze divise (4.10) godono delle seguenti proprietà:*

1. *siano $\alpha, \beta \in \mathbb{R}$ e $f(x), g(x)$ due funzioni in una variabile reale, allora*

$$(\alpha\cdot f + \beta\cdot g)[x_0,\dots,x_r] = \alpha\cdot f[x_0,\dots,x_r] + \beta\cdot g[x_0,\dots,x_r];$$

2. *per ogni $\{i_0,\dots,i_r\}$ permutazione di $\{0,\dots,r\}$,*

$$f[x_{i_0},\dots,x_{i_r}] = f[x_0,\dots,x_r],$$

   ovvero non conta l'ordine con cui compaiono le ascisse nella differenza divisa;

3. *sia $f(x) = \sum_{i=0}^{k} a_i x^i \in \Pi_k$, allora,*

$$f[x_0,\dots,x_r] = \begin{cases} a_k, & \text{se } r = b, \\ 0, & \text{se } r > b; \end{cases} \tag{4.12}$$

4. *più in generale, se $f(x) \in C^{(r)}$, allora*

$$f[x_0,\dots,x_r] = \frac{f^{(r)}(\xi)}{r!}, \qquad \xi \in [\min_i x_i, \max_i x_i]; \tag{4.13}$$

5.

$$f[x_0,x_1,\dots,x_{r-1},x_r] = \frac{f[x_1,\dots,x_r] - [x_0,\dots,x_{r-1}]}{x_r - x_0}. \tag{4.14}$$

Si può osservare che, per la (4.12), se $x_0 = \cdots = x_n$, il polinomio (4.11) coincide con il polinomio di Taylor di $f(x)$ di punto iniziale x_0.
Grazie alla proprietà (4.14), possiamo costruire una differenza divisa di ordine r se conosciamo due differenze divise di ordine $r-1$ che differiscano di una sola ascissa. Ricordando allora che

$$f[x_i] = f_i, \qquad i = 0,1,\dots,n,$$

possiamo costruire, in modo iterativo, la seguente tabella triangolare inferiore:

	0	1	...	$n-1$	n
x_0	$\boxed{f[x_0]}$				
x_1	$f[x_1]$	$\boxed{f[x_0,x_1]}$			
x_2	$f[x_2]$	$f[x_1,x_2]$	$\ddots$		
$\vdots$	$\vdots$	$\vdots$			
x_{n-1}	$f[x_{n-1}]$	$f[x_{n-2},x_{n-1}]$	...	$\boxed{f[x_0,\ldots,x_{n-1}]}$	
x_n	$f[x_n]$	$f[x_{n-1},x_n]$	...	$f[x_1,\ldots,x_n]$	$\boxed{f[x_0,x_1,\ldots,x_n]}$

La diagonale principale (elementi messi in evidenza) contiene i coefficienti del polinomio interpolante nella forma di Newton (4.11). Le colonne di tale tabella devono essere calcolate da sinistra verso destra: la prima coincide con i valori assunti dalla funzione nelle ascisse di interpolazione ed è, quindi, data; ogni elemento delle colonne successive alla prima viene calcolato applicando la (4.14) con l'elemento nella colonna immediatamente precedente (sulla stessa riga) e l'elemento nella colonna e nella riga immediatamente precedenti. Inoltre si nota che gli elementi di una determinata colonna non sono più utilizzati una volta calcolati gli elementi della colonna successiva (sulla stessa riga), e quindi per il calcolo delle differenze divise può essere utilizzato un unico vettore che viene riscritto di volta in volta fino ad ottenere l'ultima differenza divisa.
Con questi accorgimenti, possiamo definire la seguente implementazione in MATLAB per il calcolo delle differenze divise:

Codice 4.3: Calcolo delle differenze divise.

```
% f = differenzeDivise(x, f)
% Calcola le differenze divise che costituiscono i coefficienti della
% forma di Newton (rispetto alla base di Newton).
%
% Input:
%   -f: vettore contenente i valori della funzione sulle ascisse di
%   interpolazione;
%   -x: vettore contenente le n+1 ascisse di interpolazione.
% Output:
%   -f: vettore contenente le n+1 differenze divise.
%
% Autore: Tommaso Papini,
% Ultima modifica: 27 Ottobre 2012, 10:36 CEST.

function [f] = differenzeDivise(x, f)
    for i=1:length(x)-1
        for j=length(x):-1:i+1
            f(j) = (f(j) - f(j-1))/(x(j)-x(j-i));
        end
    end
end
```

4.2.1 Interpolazione di Hermite

Supponiamo adesso di avere le ascisse di interpolazione numerate nel modo seguente:

$$a \leq x_0 < x_{\frac{1}{2}} < x_1 < x_{1+\frac{1}{2}} < \ldots x_n < x_{n+\frac{1}{2}} \leq b,$$

in modo tale che le condizioni di interpolazione per il polinomio interpolante $p(x) \in \Pi_{2n+1}$ diventano:

$$p(x_i) = f_i, \qquad p(x_{i+\frac{1}{2}} = f_{i+\frac{1}{2}}, \qquad i = 0, 1, \ldots, n).$$

Facciamo allora tendere le ascisse di indice con indice frazionario verso quelle immediatamente precedenti di indice intero:

$$x_{i+\frac{1}{2}} \to x_i, \qquad i = 0, 1, \ldots, n.$$

Quindi, assumendo (fino alla fine della sezione) che $f(x) \in C^{(1)}$, otteniamo che

$$\frac{f(x_{i+\frac{1}{2}}) - f(x_i)}{x_{i+\frac{1}{2}} - x_i} = f[x_i, x_{i+\frac{1}{2}}] \overset{1}{\to} f[x_i, x_i] =$$

$$= \frac{f'(\xi)}{1!} \overset{2}{\equiv} f'(x_i), \quad i = 0, 1, \ldots, n.$$

Quindi abbiamo che le ascisse di interpolazione divengono

$$a \leq x_0 = x_0 < x_1 = x_1 < \cdots < x_n = x_n \leq b, \tag{4.15}$$

così che il polinomio interpolante (4.11) risulta ancora ben definito e si vede che soddisfa i seguenti vincoli di interpolazione:

- $p(x_i) = f(x_i)$,
- $p'(x_i) = f'(x_i), \quad i = 0, 1, \ldots, n.$

Questo polinomio allora, detto **polinomio interpolante di Hermite**, risulta interpolare sia la funzione che la sua derivata nelle ascisse di interpolazione distinte in (4.15).
Notare che, per $n = 0$, il polinomio di Hermite coincide con il polinomio di Taylor di primo grado e punto iniziale x_0:

$$p(x) = f[x_0] + f[x_0, x_0](x - x_0) = f(x_0) + f'(x_0)(x - x_0).$$

[1] Per la (4.14).
[2] Con $\xi \in [\max\{x_i\}, \min\{x_i\}]$, per la (4.13).

In generale, per un generico n, il polinomio di Hermite avrà una forma del tipo:

$$\begin{aligned} p(x) = & f[x_0]+ \\ & + f[x_0, x_0](x - x_0)+ \\ & + f[x_0, x_0, x_1](x - x_0)^2+ \\ & + f[x_0, x_0, x_1, x_1](x - x_0)^2(x - x_1)+ \\ & + f[x_0, x_0, x_1, x_1, x_2](x - x_0)^2(x - x_1)^2+ \\ & \vdots \\ & + f[x_0, x_0, x_1, x_1, \ldots, x_n, x_n](x - x_0)^2 \ldots (x - x_{n-1})^2(x - x_n). \end{aligned}$$

I coefficienti del polinomio di Hermite vengono calcolati tramite una versione ottimizzata del Codice 4.3. Il Codice 4.4 prende in input il vettore

$$f(x_0), f'(x_0), f(x_1), f'(x_1), \ldots, f(x_n), f'(x_n),$$

che viene riscritto con le differenze divise. Si osservi che le derivate della funzione sulle ascisse di interpolazione non devono essere riscritte al primo ciclo, essendo $f[x_i, x_i] = f'(x_i)$, per $i = 0, 1, \ldots, n$.

Codice 4.4: Calcolo delle differenze divise per il polinomio di Hermite.

```
% f = differenzeDiviseHermite(x, f)
% Calcola le differenze divise che costituiscono i coefficienti della
% forma di Newton (rispetto alla base di Newton) nel caso di ascisse di
% Hermite.
%
% Input:
%   -f: vettore contenente i valori della funzione e la derivata prima
%   sulle ascisse di interpolazione, nella forma [f0, f'0, f1, f'1,
%   ..., fn, f'n];
%   -x: vettore contenente le 2n+2 ascisse di interpolazione di
%   Hermite.
% Output:
%   -f: vettore contenente le 2n+2 differenze divise.
%
% Autore: Tommaso Papini,
% Ultima modifica: 27 Ottobre 2012, 17:55 CEST.

function [f] = differenzeDiviseHermite(x, f)
    for i = length(x)-1:-2:3
        f(i) = (f(i)-f(i-2))/(x(i)-x(i-2));
    end
    for i=2:length(x)-1
        for j=length(x):-1:i+1
            f(j) = (f(j)-f(j-1))/(x(j)-x(j-i));
        end
    end
end
```

Codice 4.5: Calcolo del polinomio interpolante di Hermite.

```
% p = hermite(ptx, fi)
% Calcolo dell'espressione del polinomio interpolante di Hermite.
%
% Input:
%   -ptx: vettore contenente le ascisse di interpolazione;
%   -fi: vettore contenente i valori della funzione e la derivata prima
%   sulle ascisse di interpolazione, nella forma [f0, f'0, f1, f'1,
%   ..., fn, f'n].
% Output:
%   -p: espressione (come funzione inline) del polinomio
%   interpolante.
%
% Autore: Tommaso Papini,
% Ultima modifica: 28 Ottobre 2012, 19:00 CET

function [p] = hermite(ptx, fi)
    n = length(ptx)-1;
    ptx2 = zeros(2*n+2, 1);
    ptx2(1:2:2*n+1)=ptx;
    ptx2(2:2:2*n+2)=ptx;
    dd = differenzeDiviseHermite(ptx2, fi);
    syms x;
    p = dd(1);
    for i=2:2*n+2
        prod = dd(i);
        for j=1:i-1
            prod = prod*(x-ptx2(j));
        end
        p = p + prod;
    end
    p = inline(p);
end
```

4.3 Errore nell'interpolazione

Studiamo adesso di quanto si sbaglia utilizzando il polinomio interpolante al posto della funzione originale.

Definizione 4.1. *Sia $p(x) \in \Pi_n$ il polinomio interpolante soddisfacente le condizioni d'interpolazione (4.2). Si dice* ***errore di interpolazione*** *la quantità*

$$e(x) = f(x) - p(x), \tag{4.16}$$

ovvero l'errore commesso nell'approssimare $f(x)$ mediante il suo polinomio interpolante $p(x)$.

Ovviamente, per le condizioni (4.2), risulta che

$$e(x_i) = 0, \qquad i = 0, 1, \ldots, n,$$

ovvero l'errore è nullo sulle ascisse di interpolazione.

Teorema 4.5. *Sia $p(x)$ il polinomio il polinomio (4.11) soddisfacente le (4.2). Il corrispondente errore di interpolazione (4.16) vale*

$$e(x) = f[x_0, x_1, \ldots, x_n, x]w_{n+1}(x), \tag{4.17}$$

ovvero il termine "mancante" in $p(x)$ se si fosse interpolato su un'ulteriore ascissa corrispondente ad x.

Corollario 4.1. *Se $f(x) \in C^{(n+1)}$, allora*

$$e(x) = \frac{f^{(n+1)}(\xi_x)}{(n+1)!}w_{n+1}(x), \qquad \xi_x \in [\min\{x_0, x\}, \max\{x_n, x\}].$$

L'ascissa minima risulta essere x_0 o x, in quanto le ascisse (4.1) sono ordinate in senso crescente. Analogamente per la massima.

Quindi la struttura dell'errore d'interpolazione ci dice che esso è composto sostanzialmente da due parti:

- l'una, $\frac{f^{(n+1)}(\xi_x)}{(n+1)!}$, dipende dalle proprietà di regolarità della funzione $f(x)$,
- l'altra, $w_{n+1}(x)$, che dipende esclusivamente dalla scelta delle ascisse di interpolazione.

Risulta che $w_{n+1}(x)$ oscilla per $x \in [x_0, x_n]$, annullandosi nelle ascisse di interpolazione (4.1). Al contrario, $|w_{n+1}(x)|$ cresce con la stessa velocità di x^{n+1} al di fuori dell'intervallo di interpolazione, ovvero per $x < x_0$ o $x > x_n$. Se ne conclude che il polinomio $p(x)$ può essere convenientemente utilizzato per approssimare $f(x)$ soltanto all'interno dell'intervallo $[x_0, x_n]$. Per questo si parla di **interpolazione** polinomiale anziché di *estrapolazione* (che invece si occupa dell'utilizzo di $p(x)$ al di fuori di tale intervallo). Inoltre si osserva che, aumentando opportunamente il numero delle ascisse di interpolazione, l'errore di interpolazione commesso nell'intervallo $[a, b]$ decresce.

4.4 Condizionamento del problema

Come tutti i problemi fin'ora affrontati, dedichiamoci adesso allo studio del *condizionamento del problema* della valutazione del polinomio interpolante. Consideriamo quindi, come unici dati di ingresso, i valori $\{f_i\}$ assunti dalla funzione $f(x)$ sulle ascisse di interpolazione (4.1). L'analisi del condizionamento verrà condotta sugli errori assoluti.
Abbiamo quindi che il polinomio interpolante *esatto*, in forma di Lagrange, è dato da

$$p(x) = \sum_{k=0}^{n} f_k L_{k,n}(x),$$

mentre il polinomio interpolante costruito a partire dai *dati perturbati* è

$$\tilde{p}(x) = \sum_{k=0}^{n} \tilde{f}_k L_{k,n}(x),$$

dove $\tilde{f}_i \equiv \tilde{f}(x_i)$, essendo $\tilde{f}(x)$ una perturbazione di $f(x)$. Si ottiene quindi

$$\begin{aligned} |p(x) - \tilde{p}(x)| &= |\sum_{k=0}^{n} f_k L_{k,n}(x) - \sum_{k=0}^{n} \tilde{f}_k L_{k,n}(x)| = |\sum_{k=0}^{n} (f_k - \tilde{f}_k) L_{k,n}(x)| = \\ &= |\sum_{k=0}^{n} (f_k - \tilde{f}_k) \cdot L_{k,n}(x)| \leq \sum_{k=0}^{n} |(f_k - \tilde{f}_k)| \cdot |L_{k,n}(x)| \leq {}^{1} \\ &\leq \left(\sum_{k=0}^{n} |L_{k,n}(x)| \right) \max_k |f_k - \tilde{f}_k| \equiv {}^{2} \lambda_n(x) \max_k |f_k - \tilde{f}_k|, \end{aligned}$$

dove la funzione $\lambda_n(x) \equiv (\sum_{k=0}^{n} |L_{k,n}(x)|)$ è detta **funzione di Lebesgue**, che si vede dipende soltanto dalla scelta delle ascisse di interpolazione (in quanto i polinomi $|L_{k,n}(x)|$ che la compongono dipendono soltanto dalle ascisse (4.1)). Considerando, come avevamo già concluso nella Sezione 4.3, il caso in cui $x \in [a, b]$ (vedi (4.1)), definiamo la norma in $C^{(0)}$ come

$$||f|| = \max_{a \leq x \leq b} |f(x)|. \tag{4.18}$$

Volendo quindi stimare una maggiorazione per la quantità $|p(x) - \tilde{p}(x)|$, avremo che

$$||p - \tilde{p}|| \leq ||\lambda_n|| \cdot ||f - \tilde{f}|| \equiv \Lambda_n ||f - \tilde{f}||, \tag{4.19}$$

dove la quantità Λ_n è detta **costante di Lebesgue**, la quale dipende esclusivamente dalla scelta delle ascisse di interpolazione e dall'intervallo $[a, b]$ considerato. Osserviamo che, essendo nella (4.19) la quantità $||f - \tilde{f}||$ una maggiorazione dell'errore assoluto presente sui dati in ingresso e, analogamente, la quantità $||p - \tilde{p}||$ una maggiorazione dell'errore assoluto commesso sul risultato, si ha che la *costante di Lebesgue*, Λ_n, rappresenta il **numero di condizionamento** del problema di valutazione del polinomio interpolante. A questo proposito è noto che:

- $\Lambda_n \geq O(\log n) \to \infty$ per $\to \infty$. Pertanto il problema diviene progressivamente *malcondizionato* al crescere del numero di ascisse di interpolazione n, ed inoltre Λ_n crescerà almeno con velocità *logaritmica*;

- la scelta di ascisse di interpolazione equidistanti

$$x_i = a + i \cdot h, \qquad i = 0, 1, \ldots, n, \qquad h = \frac{b - a}{n}, \tag{4.20}$$

[1] Per la *disuguaglianza triangolare*, il valore assoluto della somma è minore o uguale alla somma dei valori assoluti.

[2] Viene maggiorata la differenza tra i vari f_k ed $\tilde{f}_k$ e raccolta come fattore comune per tutti i polinomi $|L_{k,n}(x)|$.

per quanto possa sembrare logica, genera una successione $\{\Lambda_n\}$ che *diverge* con velocità approssimativamente *esponenziale*, per $n \to \infty$. Non rappresenta quindi la scelta ottimale, specialmente per valori elevati di n.

Codice 4.6: Calcolo delle ascisse di interpolazione equidistanti.

```
% ptx = ascisseEquidistanti(a, b, n)
% Calcolo delle ascisse equidistanti in un dato intervallo.
%
% Input:
%   -a: estremo sinistro dell'intervallo;
%   -b: estremo destro dell'intervallo;
%   -n: numero delle ascisse da produrre (n+1, da 0 ad n).
% Output:
%   -ptx vettore contenente le n+1 ascisse equidistanti.
%
% Autore: Tommaso Papini,
% Ultima modifica: 28 Ottobre 2012, 17:09 CET

function [ptx] = ascisseEquidistanti(a, b, n)
    h = (b-a)/n;
    ptx = zeros(n+1, 1);
    for i=1:n+1
        ptx(i) = a +(i-1)*h;
    end
end
```

Studiamo adesso che relazione intercorre tra l'errore dell'interpolazione ed il condizionamento del problema.

Definizione 4.2. *Data una funzione $f(x)$ continua in $[a,b]$, il polinomio $p^*(x) \in \Pi_n$ tale che*

$$||f - p^*|| = \min_{p \in \Pi_n} ||f - p||, \tag{4.21}$$

*si dice **polinomio di miglior approssimazione** di grado n di $f(x)$ sull'intervallo $[a,b]$.*

Teorema 4.6. *Assegnata una funzione $f(x)$ continua in $[a,b]$, esiste il polinomio $p^*(x) \in \Pi_n$ di miglior approssimazione (vedi (4.21)) di $f(x)$ su $[a,b]$.*

Teorema 4.7. *Sia $p^*(x)$ il polinomio di miglior approssimazione di grado n di $f(x)$. Allora per l'errore di interpolazione (4.16) vale*

$$||e|| \leq (1 + \Lambda_n)||f - p^*||. \tag{4.22}$$

Quindi non è detto che, al crescere di n, l'errore decresca, in quanto la costante di Lebesgue diverge esponenzialmente.
Introduciamo il concetto di **modulo di continuità** di una funzione:

$$\omega(f;h) \equiv \sup\{|f(x) - f(y)| : x, y \in [a,b], |x - y| \leq h\},$$

dove $h > 0$ è un parametro assegnato. Si osserva che:

- se $f \in C^0$, allora $\omega(f;h) \to 0$, per $h \to 0$: infatti diminuendo sempre di più l'intervallo h, ed essendo la funzione continua, i valori della f si avvicinano sempre di più;
- se $f(x)$ è *Lipschitziana* con costante L, allora $\omega(f;h) \leq Lh$. Ad esempio, se $f(x) \in C^{(1)}$, $L = \max_{a \leq x \leq b} |f'(x)| \equiv ||f'||$.

Teorema 4.8 (Jackson)**.** *Per il polinomio di miglio approssimazione (4.21) di una funzione $f(x) \in C^{(0)}$ si ha:*

$$||f - p^*|| \leq \alpha \cdot \omega \left(f; \frac{b-a}{n} \right), \tag{4.23}$$

in cui la costante α è indipendente da n.

Segue infine, per la (4.22) e la (4.23), che, per una generica $f(x)$:

$$||e|| \leq \alpha(1 + \Lambda_n)\omega \left(f; \frac{b-a}{n} \right).$$

Quindi, concludendo e riassumendo questa Sezione e la precedente, è opportuno effettuare una scelta delle ascisse di interpolazione in modo tale che:

1. la costante di Lebesgue Λ_n abbia una crescita moderata, preferibilmente logaritmica (che abbiamo visto essere quella ottimale), rispetto al grado n del polinomio interpolante;
2. sia minimizzata la quantità $||w_{n+1}||$, come avevamo già dedotto in conclusione della Sezione 4.3.

4.5 Ascisse di Chebyshev

Quindi risulta chiaro che si tratta di scegliere le ascisse in modo da minimizzare la norma $||w_{n+1}||$. Ma la norma, per definizione, è il massimo di una funzione su un certo intervallo (vedi (4.18)), ovvero si tratta di minimizzare un valore massimo: si deve allora cercare la soluzione del seguente **problema del minimassimo** (o **minmax**):

$$\min_{a \leq x_0 < \cdots < x_n \leq b} ||w_{n+1}|| \equiv \min_{a \leq x_0 < \cdots < x_n \leq b} \max_{a \leq x \leq b} |w_{n+1}(x)|.$$

Senza perdere di generalità (vedi Esercizio 4.12) assumiamo che

$$[a, b] \equiv [-1, 1].$$

Definiamo allora la famiglia dei **polinomi di Chebyshev di prima specie**:

$$\begin{aligned} &T_0(x) \equiv 1, \\ &T_1(x) \equiv x, \\ &T_{k+1}(x) \equiv 2xT_k(x) - T_{k-1}(x), \qquad k = 1, 2, \ldots. \end{aligned} \tag{4.24}$$

Si possono dimostrare (per induzione) le seguenti proprietà dei polinomi di Chebyshev:

1. $T_k(x)$ è un polinomio di grado esatto k;

2. il coefficiente principale di $T_k(x)$ è 2^{k-1}, per $k = 1, 2, \ldots$;

3. la famiglia dei polinomi $\{\hat{T}_k\}$, dove

$$\hat{T}_0(x) = T_0(x), \quad \hat{T}_k(x) = 2^{1-k}T_k(x) = \frac{T_k(x)}{2^{k-1}}, \quad k = 1, 2, \ldots,$$

è una famiglia di *polinomi monici* (dal coefficiente principale uguale a 1) di grado k, per $k = 0, 1, \ldots$;

4. possiamo parametrizzare i punti dell'intervallo $[-1, 1]$ rispetto a θ, ponendo

$$x = \cos\theta, \qquad \theta \in [0, \pi].$$

Inoltre, considerando che

$$\cos(k\theta + \theta) + \cos(k\theta - \theta) = 2\cos(k\theta)\cos(\theta)$$

si ottiene

$$T_k(x) \equiv T_k(\cos\theta) = \cos(k\theta), \qquad k = 0, 1, \ldots.$$

Sfruttando queste proprietà si ottiene:

Teorema 4.9. *Gli zeri di $T_k(x)$, tra loro distinti, sono dati da*

$$x_i^{(k)} = \cos\left(\frac{(2i+1)\pi}{2k}\right), \qquad i = 0, 1, \ldots, k-1.$$

Si può vedere che , per $x \in [-1, 1]$, i valori estremi del polinomio $T_k(x)$ sono assunti nei punti

$$\xi_i^{(k)} = \cos\left(\frac{i}{k}\pi\right), \qquad i = 0, 1, \ldots, k,$$

nei quali il polinomio assume i valori

$$T_k(\xi_i^{(k)}) = (-1)^i, \qquad i = 0, 1, \ldots, k,$$

quindi risulta che $||T_k|| = 1$ (cioè il valore massimo del polinomio è 1, essendo un coseno). Inoltre, per $k = 1, 2, \ldots$,

$$||\hat{T}_k|| = 2^{1-k} = \min_{\varphi \in \Pi'_k} ||\varphi||, \tag{4.25}$$

ovvero il polinomio monico $\hat{T}_k$ ha norma minima tra tutti i polinomi monici di grado k.

Si ottiene allora che, scegliendo come ascisse di interpolazione sull'intervallo $[-1, 1]$ come

$$x_{n-i} = \cos\left(\frac{2i+1}{2(n+1)}\pi\right), \qquad i = 0, 1, \ldots, n, \tag{4.26}$$

(dove l'indice $n-i$ serve a generare le ascisse in modo che siano ordinate in senso crescente rispetto al loro indice) si ha

$$w_{n+1}(x) = \prod_{i=0}^{n}(x - x_i) \equiv \hat{T}_{n+1}(x),$$

che rappresenta quindi la soluzione al problema del minimassimo, per la (4.25). Per convertire queste ascisse nelle ascisse di interpolazione sull'intervallo $[a, b]$ basta effettuare la trasformazione (vedi Esercizio 4.12):

$$x_i|_{[a,b]} = \frac{a+b}{2} + \frac{b-a}{2}x_i|_{[-1,1]}, \qquad i = 0 \ldots, n. \tag{4.27}$$

Il seguente Codice MATLAB calcola le ascisse di Chebyshev secondo quanto appena visto:

Codice 4.7: Calcolo delle ascisse di Chebyshev.

```
% xi = ascisseChebyshev(a,b,n)
% Vengono generate le ascisse di Chebyshev su un determinato
% intervallo.
%
% Input:
%   -a: l'estremo sinistro dell'intervallo;
%   -b: l'estremo destro dell'intervallo;
%   -n: il numero di ascisse che si vuole generare (n+1, da 0 ad n).
% Output:
%   -xi: vettore contenente le ascisse di Chebyshev generate.
%
% Autore: Tommaso Papini,
% Ultima modifica: 23 Ottobre 2012, 10:47 CEST.

function [xi] = ascisseChebyshev(a,b,n)
    xi = zeros(n+1, 1);
    for i=0:n
        xi(n+1-i) = (a+b)/2 + cos(pi*(2*i+1)/(2*(n+1)))*(b-a)/2;
    end
end
```

Scegliendo le ascisse 4.27 di interpolazione, dette **ascisse di Chebishev**, si ottiene (per funzioni sufficientemente regolari)

$$\begin{aligned} ||e|| &\leq \frac{||f^{(n+1)}||}{(n+1)!}||w_{n+1}|| = \\ &= \frac{||f^{(n+1)}||}{(n+1)!}||\hat{T}_{n+1}|| = \\ &= \frac{||f^{(n+1)}||}{(n+1)!}2^{1-n-1} = \\ &= \frac{||f^{(n+1)}||}{(n+1)!2^n}, \end{aligned}$$

e la corrispondente costante di Lebesgue si vede valere

$$\Lambda_n \approx \frac{2}{\pi}\log n,$$

che risulta quindi avere una *crescita ottimale*, per $n \to \infty$.

Riassumendo, abbiamo concluso che scegliendo come ascisse di interpolazione le *ascisse di Chebyshev* (4.26) si ottengono i polinomi monici $\hat{T}_k$, $k = 0, 1, \ldots, n$, che formano una *base di Newton* sulla quale poter costruire il polinomio interpolante (come visto in Sezione 4.2), e tale che la norma del polinomio della base di grado $n+1$ (il primo polinomio "mancante" dalla base) sia minima (pari a 2^{-n}).
Si osserva che utilizzando le *ascisse di Chebyshev* (4.26) si perde la caratteristica della forma di Newton di poter generare polinomi interpolanti in modo incrementale in quanto le ascisse in questione dipendono dal grado n del polinomio interpolante (compare la n al denominatore) e quindi volendo determinare il polinomio interpolante di grado $n+1$ saremmo costretti a ricalcolare tutte le ascisse.

4.6 Interpolazione mediante funzioni *spline*

Abbiamo visto che, al crescere del grado n del polinomio interpolante, è necessario effettuare una scelta delle ascisse che non faccia crescere troppo velocemente la costante di Lebesgue Λ_n, che comunque crescerà almeno con velocità logaritmica rispetto ad n. Tuttavia se n rimane basso, il *modulo di continuità*

$$w\left(f; \frac{b-a}{n}\right)$$

non può tendere a 0, con l'intervallo $[a, b]$ fissato.
Allora:

1. consideriamo una partizione dell'intervallo originario,

$$\Delta = \{a = x_0 < x_1 < \cdots < x_n = b\}, \tag{4.28}$$

con

$$h = \max_{i=1,\dots,n} (x_i - x_{i-1}) \to 0, \qquad n \to \infty; \tag{4.29}$$

2. su ciascun sottointervallo $[x_{i-1}, x_i]$ della partizione Δ consideriamo un polinomio di grado m fissato interpolante la funzione $f(x)$ nei suoi estremi.

In questo modo il problema del condizionamento passa in secondo piano in quanto il grado m dei polinomi interpolanti rimane fissato mentre, al contempo (se $f \in C^{(0)}$),

$$w(f; h) \to 0, \qquad n \to \infty.$$

Questo nuovo tipo di funzione interpolante si dice **funzione polinomiale a tratti**, in quanto è rappresentabile da un polinomio in ogni sottointervallo, ma non nel suo insieme. Più in particolare:

Definizione 4.3. *La funzione $s_m(x)$ si dice **spline di grado** m sulla partizione Δ se*

1. *$s_m(x) \in C^{(m-1)}$ sull'intervallo $[a, b]$ e*
2. *$s_m|_{[x_{i-1},x_i]}(x) \in \Pi_m$, per $i = 1, \dots, n$.*

Se risulta che

$$s_m(x_i) = f_i, \qquad i = 0, 1, \dots, n, \tag{4.30}$$

allora si dice che la *spline* **interpola** la funzione $f(x)$ nei nodi della partizione Δ.

Teorema 4.10. *Se $s_m(x)$ è una spline di grado m sulla partizione (4.28), allora $s'_m(x)$ è una spline di grado $m-1$ sulla stessa partizione.*

Teorema 4.11. *L'insieme delle funzioni spline di grado m definite sulla partizione (4.28) è uno spazio vettoriale di dimensione $m + n$.*

Quest'ultimo teorema ci dice che sono necessarie $m + n$ condizioni indipendenti per poter individuare *univocamente* la spline interpolante la funzione $f(x)$ sulla partizione Δ assegnata. Essendo allora le *condizioni di interpolazione* (4.30) soltanto $n + 1$, utilizzandolo soltanto queste condizioni possiamo individuare univocamente una spline di grado 1, o **spline lineare**, che coincide con la spezzata congiungente i punti $\{(x_i, f_i)\}_{i=0,\dots,n}$:

$$s_1|_{[x_{i-1},x_i]}(x) = \frac{(x - x_{i-1})f_i + (x_i - x)f_{i-1}}{x_i - x_{i-1}}. \tag{4.31}$$

Risulta quindi necessario, per spline interpolanti di ordine superiore al primo, introdurre ulteriori condizioni (in particolare, $m - 1$ condizioni aggiuntive rispetto alle condizioni di interpolazione (4.30)) per definire la spline interpolante in maniera univoca.

4.7 *Spline* cubiche

Le **spline cubiche** sono spline interpolanti di grado 3. È quindi necessario imporre 2 condizioni aggiuntive, oltre alle (4.30), per poter definire univocamente tale spline interpolante. In particolare, a seconda di quali condizioni sceglieremo di imporre, si avranno diversi tipi di *spline cubica*.

***Spline* naturale:**
La *spline* naturale consiste nel fissare le seguenti condizioni aggiuntive:

$$s_3''(a) = 0, \qquad s_3''(b) = 0,$$

ovvero viene imposto che la derivata seconda della spline negli estremi della partizione Δ si annulli.

***Spline* completa:**
Supponendo di conoscere i valori della derivata prima $f'(x)$ della funzione negli estremi della partizione, nel caso della *spline* completa si impongono le seguenti condizioni.

$$s_3'(a) = f'(a), \qquad s_3'(b) = f'(b).$$

***Spline* periodica:**
Se la funzione da interpolare è periodica e l'intervallo $[a, b]$ contiene un numero intero di periodi della funzione, allora conviene utilizzare la *spline* peridica, che consiste nell'imporre le condizioni

$$s_3'(a) = s_3'(b), \qquad s_3''(a) = s_3''(b).$$

Condizioni *not-a-knot*:
In questo caso, per far sì che le condizioni di interpolazione (4.30) siano sufficienti a determinare la *spline* cubica, si impone che lo stesso polinomio di terzo grado costituisca la restrizione della *spline* sull'intervallo $[x_0, x_1] \cup [x_1, x_2]$ e, allo stesso modo, lo stesso polinomio di Π_3 costituisca la restrizione della *spline* sull'intervallo $[x_{n-2}, x_{n-1}] \cup [x_{n-1}, x_n]$. In poche parole si impone che nel primo e secondo sottointervallo e nel penultimo ed ultimo sottointervallo, il polinomio interpolante la funzione sia lo stesso, ovvero che

$$s_3|_{[x_0,x_2]} \in \Pi_3, \qquad s_3|_{[x_{n-2},x_n]} \in \Pi_3.$$

In particolare i punti (x_1, f_1) ed (x_{n-1}, f_{n-1}) non saranno più nodi, da cui il nome **not-a-knot** ("non un nodo").
Essendo, per definizione di spline, $s_3 \in C^{(2)}$, per avere lo stesso polinomio nei due sottointervalli successivi manca da imporre che la derivata terza nel punto comune ai due sottointervalli sia uguale, quindi le due condizioni aggiuntive sono:

$$s_3'''|_{[x_0,x_1]}(x_1) = s_3'''|_{[x_1,x_2]}(x_1), \quad s_3'''|_{[x_{n-2},x_{n-1}]}(x_{n-1}) = s_3'''|_{[x_{n-1},x_n]}(x_{n-1}).$$

Grazie al Teorema 4.10, $s_3'''|_{[x_{i-1},x_i]}(x) \in \Pi_0$ (ovvero è una retta), quindi queste due condizioni sono esprimibili con i relativi rapporti incrementali, ovvero come:

$$\frac{s_3''(x_1) - s_3''(x_0)}{x_1 - x_0} = \frac{s_3''(x_2) - s_3''(x_1)}{x_2 - x_1},$$
$$\frac{s_3''(x_{n-1}) - s_3''(x_{n-2})}{x_{n-1} - x_{n-2}} = \frac{s_3''(x_n) - s_3''(x_{n-1})}{x_n - x_{n-1}}.$$

Si può osservare che le *spline* cubiche consentono di approssimare efficientemente funzioni regolari senza preoccuparsi più di tanto della scelta dei nodi della partizione (anche una scelta uniforme dei nodi va più che bene). Infatti si può dimostrare che, se $f(x) \in C^{(4)}$, allora (vedi (4.29))

$$||f^{(k)} - s_3^{(k)}|| = O(h^{4-k}), \qquad k = 0, 1, 2,$$

ovvero $s_3(x)$ approssima efficientemente la funzione $f(x)$ e le sue prime due derivate, per $h \to 0$, ovvero all'aumentare dei nodi di interpolazione. Questo risultato vale per le *spline* complete, periodiche e *not-a-knot* ed essenzialmente anche per le *spline* naturali.

4.8 Calcolo di una *spline* cubica

Vediamo in questa Sezione come sia possibile calcolare una *spline* naturale ed una *not-a-knot* (per gli altri due casi si utilizzano calcoli molto simili).
Con riferimento alla partizione (4.28) denotando

$$m_i \equiv s_3''(x_i), \qquad i = 0, 1, \ldots, n, \tag{4.32}$$

le condizioni per le *spline* naturali diventano

$$m_0 = m_n = 0. \tag{4.33}$$

Mentre, ponendo

$$h_i = x_i - x_{i-1}, \qquad i = 1, 2, \ldots, n,$$

si ha che le condizioni per le *spline not-a-knot* diventano

$$\frac{s_3''(x_1) - s_3''(x_0)}{x_1 - x_0} = \frac{s_3''(x_2) - s_3''(x_1)}{x_2 - x_1}, \quad \frac{s_3''(x_{n-1}) - s_3''(x_{n-2})}{x_{n-1} - x_{n-2}} = \frac{s_3''(x_n) - s_3''(x_{n-1})}{x_n - x_{n-1}},$$

$$\frac{m_1 - m_0}{h_1} = \frac{m_2 - m_1}{h_2}, \quad \frac{m_{n-1} - m_{n-2}}{h_{n-1}} = \frac{m_n - m_{n-1}}{h_n},$$

$$m_1 h_2 - m_0 h_2 = m_2 h_1 - m_1 h_1, \quad m_{n-1} h_n - m_{n-2} h_n = m_n h_{n-1} - m_{n-1} h_{n-1},$$

$$h_1 m_2 + h_2 m_0 = (h_1 + h_2) m_1, \quad h_{n-1} m_n + h_n m_{n-2} = (h_{n-1} + h_n) m_{n-1}. \tag{4.34}$$

Per il Teorema 4.10, essendo $s_3(x)$ una *spline* cubica, allora $s_3'(x)$ è una spline di grado 2, mentre $s_3''(x)$ è una spline lineare. Per la (4.31) sappiamo che la *spline* lineare coincide con la spezzata che congiunge i nodi di interpolazione, segue quindi che:

$$\begin{aligned} s_3''|_{[x_{i-1},x_i]} &= \frac{(x-x_{i-1})s_3''(x_i)+(x_i-x)s_3''(x_{i-1})}{x_i-x_{i-1}} \\ &= \frac{(x-x_{i-1})m_i+(x_i-x)m_{i-1}}{h_i}. \end{aligned}$$

Integrando si ottiene:

$$s_3'(x)|_{[x_{i-1},x_i]} = \frac{(x-x_{i-1})^2 m_i-(x_i-x)^2 m_{i-1}}{2h_i}+q_i, \tag{4.35}$$

con q_i opportuna costante d'integrazione. Integrando ulteriormente si ottiene:

$$s_3(x)|_{[x_{i-1},x_i]} = \frac{(x-x_{i-1})^3 m_i+(x_i-x)^3 m_{i-1}}{6h_i}+q_i(x-x_{i-1})+r_i, \tag{4.36}$$

dove r_i è una seconda costante d'integrazione.
Imponendo a questo punto le condizioni d'interpolazione (4.30) si ottiene

$$\begin{aligned} s_3(x_{i-1}) &= \frac{h_i^2}{6}m_{i-1}+r_i \equiv f_{i-1}, \\ s_3(x_i) &= \frac{h_i^2}{6}m_i+q_i h_i r_i \equiv f_i. \end{aligned}$$

Pertanto si ricava:

$$r_i = f_{i-1}-\frac{h_i^2}{6}m_{i-1}, \tag{4.37}$$

$$q_i = \frac{f_i-f_{i-1}}{h_i}-\frac{h_i}{6}(m_i-m_{i-1}). \tag{4.38}$$

Una volta calcolati i fattori $\{m_i\}$, come vedremo di seguito, possiamo ricavare le n espressioni che caratterizzano la *spline*, come descritto dal Codice 4.8.

Codice 4.8: Calcolo delle espressioni di una *spline* (noti i fattori $\{m_i\}$).

```
% s = espressioniSplineCubica(ptx, fi, mi)
% Calcola le espressioni degli n polinomi costituenti una spline
% cubica.
%
% Input:
%   -ptx: vettore contenente gli n+1 nodi di interpolazione;
%   -fi: vettore contenente i valori assunti dalla funzione da
%   approssimare nei nodi in ptx;
%   -mi: fattori m_i calcolati risolvendo il sistema lineare
%   corrispondente.
% Output:
```

```
%   -s: vettore contenente le espressioni degli n polinomi che
%   definiscono la spline cubica.
%
% Autore: Tommaso Papini,
% Ultima modifica: 24 Ottobre 2012, 12:19 CEST.

function [s] = espressioniSplineCubica(ptx, fi, mi)
    s = sym('x', [length(ptx)-1 1]);
    syms x;
    for i=2:length(ptx)
        hi = ptx(i)-ptx(i-1);
        ri = fi(i-1)-((hi^2)/6)*mi(i-1);
        qi = (fi(i)-fi(i-1))/hi -(hi/6)*(mi(i)-mi(i-1));
        s(i-1)=(((x - ptx(i-1))^3)*mi(i) + ((ptx(i) - x)^3)*mi(i-1))
            /(6*hi) +qi*(x - ptx(i-1)) +ri;
    end
end
```

Sostituendo l'espressione (4.38) nella (4.35) si ottiene:

$$s_3'|_{[x_{i-1},x_i]}(x) = \frac{(x-x_{i-1})^2 m_i - (x_i - x)^2 m_{i-1}}{2h_i} + \frac{f_i - f_{i-1}}{h_i} - \frac{h_i}{6}(m_i - m_{i-1}).$$

Quindi, considerando che $s_3'(x)$ deve risultare $\in C^{(1)}$, si deve imporre la continuità in x_i, che è il punto di incontro tra i sottointervalli $[x_{i-1}, x_i]$ e $[x_i, x_{i+1}]$. Ovvero deve valere

$$s_3'|_{[x_{i-1},x_i]}(x_i) = s_3'|_{[x_i,x_{i+1}]}(x_i),$$

che equivale ad imporre (ricordando che, per definizione di *differenza divisa*, $\frac{f_i - f_{i-1}}{h_i} = f[x_{i-1}, x_i]$):

$$\frac{h_i}{2}m_i + f[x_{i-1}, x_i] - \frac{h_i}{6}(m_i - m_{i-1}) = -\frac{h_{i+1}}{2}m_i + f[x_i, x_{i+1}] - \frac{h_{i+1}}{6}(m_{i+1} - m_i),$$

ovvero (per la (4.14)),

$$\varphi_i m_{i-1} + 2m_i + \xi_i m_{i+1} = 6f[x_{i-1}, x_i, x_{i+1}],$$

con

$$\varphi_i = \frac{h_i}{h_i + h_{i+1}}, \qquad \xi_i = \frac{h_{i+1}}{h_i + h_{i+1}}, \qquad i = 1, \ldots, n-1.$$

Si osserva che

$$\varphi_i \xi_i > 0, \qquad \varphi_i + \xi_i = 1, \qquad i = 1, \ldots, n-1. \tag{4.39}$$

Per una *spline* naturale, quindi, tenendo conto delle condizioni aggiuntive (4.33), si ottiene il seguente sistema *tridiagonale*:

$$\begin{pmatrix} 2 & \xi_1 & & & \\ \varphi_2 & 2 & \xi_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \xi_{n-2} \\ & & & \varphi_{n-1} & 2 \end{pmatrix} \begin{pmatrix} m_1 \\ m_2 \\ \vdots \\ \vdots \\ m_{n-1} \end{pmatrix} = 6 \begin{pmatrix} f[x_0, x_1, x_2] \\ f[x_1, x_2, x_3] \\ \vdots \\ \vdots \\ f[x_{n-2}, x_{n-1}, x_n] \end{pmatrix}. \tag{4.40}$$

Dalla (4.39) si vede che la matrice dei coefficienti è *diagonale dominante per righe*, e quindi fattorizzabile LU. Una volta calcolati i valori $\{m_i\}$ incogniti, si sostituiscono, assieme ai valori di r_i e q_i (vedi (4.37) e (4.38)), nella (4.36).
Da questo sistema, e da altre considerazioni (vedi Esercizio 4.16), possiamo ricavare il seguente Codice per il calcolo dei fattori $\{m_i\}$ per una *spline* cubica naturale:

Codice 4.9: Calcolo dei fattori $\{m_i\}$ per una *spline* cubica naturale.

```
% m = risolviSistemaSplineNaturale(phi, xi, dd)
% Risoluzione del sistema lineare di una spline cubica naturale per la
% determinazione dei fattori m_i necessari per la costruzione
% dell'espressione della spline cubica naturale.
%
% Input:
%   -phi: vettore dei fattori phi che definiscono la matrice dei
%   coefficienti (lunghezza n-1);
%   -xi: vettore dei fattori xi che definiscono la matrice dei
%   coefficienti (lunghezza n-1);
%   -dd: vettore delle differenze divise (lunghezza n-1).
% Output:
%   -m: vettore contenente gli n-1 fattori m_i calcolati.
%
% Autore: Tommaso Papini,
% Ultima modifica: 1 Novembre 2012, 17:49 CET.

function [m] = risolviSistemaSplineNaturale(phi, xi, dd)
    dd = 6*dd;
    n = length(xi)+1;
    u = zeros(n-1, 1);
    l = zeros(n-2, 1);
    u(1)=2;
    for i=2:n-1
        l(i)=phi(i)/u(i-1);
        u(i)=2-l(i)*xi(i-1);
    end
    y = zeros(n-1, 1);
    y(1)=dd(1);
    for i=2:n-1
        y(i)=dd(i)-l(i)*y(i-1);
    end
    m = zeros(n-1, 1);
    m(n-1)=y(n-1)/u(n-1);
    for i=n-2:-1:1
        m(i)=(y(i)-xi(i)*dd(i+1))/u(i);
    end
    m = [0; m; 0];
end
```

Analogamente per le *spline not-a-knot*, tenendo conto delle condizioni aggiuntive (4.34), si ottiene il seguente sistema lineare:

$$\begin{pmatrix} \xi_1 & -1 & \varphi_1 & & \\ \varphi_1 & 2 & \xi_1 & & \\ & \ddots & \ddots & \ddots & \\ & & \varphi_{n-1} & 2 & \xi_{n-1} \\ & & \xi_{n-1} & -1 & \varphi_{n-1} \end{pmatrix} \begin{pmatrix} m_0 \\ m_1 \\ \vdots \\ m_{n-1} \\ m_n \end{pmatrix} = 6 \begin{pmatrix} 0 \\ f[x_0, x_1, x_2] \\ \vdots \\ f[x_{n-2}, x_{n-1}, x_n] \\ 0 \end{pmatrix}.$$

Si vede facilmente che quest'ultimo sistema lineare è equivalente a:

$$A\underline{x} = 6\underline{b}, \tag{4.41}$$

con:

$$A = \begin{pmatrix} 1 & 0 & & & & & & \\ \varphi_1 & (2-\varphi_1) & (\xi_1-\varphi_1) & & & & & \\ & \varphi_2 & 2 & \xi_2 & & & & \\ & & \ddots & \ddots & \ddots & & & \\ & & & \varphi_{n-2} & 2 & \xi_{n-2} & & \\ & & & & (\varphi_{n-1}-\xi_{n-1}) & (2-\xi_{n-1}) & \xi_{n-1} \\ & & & & & 0 & 1 \end{pmatrix},$$

$$\underline{x} = \begin{pmatrix} m_0 + m_1 + m_2 \\ m_1 \\ \vdots \\ m_{n-1} \\ m_{n-2} + m_{n-1} + m_n \end{pmatrix},$$

$$\underline{b} = \begin{pmatrix} f[x_0, x_1, x_2] \\ f[x_0, x_1, x_2] \\ \vdots \\ f[x_{n-2}, x_{n-1}, x_n] \\ f[x_{n-2}, x_{n-1}, x_n] \end{pmatrix},$$

il quale, avendo tutti i minori principali non nulli, risulta fattorizzabile LU.
Analogamente a quanto visto per le *spline* cubiche naturali, possiamo implementare efficientemente (vedi Esercizio 4.17) in MATLAB il calcolo degli $\{m_i\}$ nel caso delle *spline* cubiche *not-a-knot* come segue:

Codice 4.10: Calcolo dei fattori $\{m_i\}$ per una *spline* cubica *not-a-knot*.

```
% m = risolviSistemaSplineNotAKnot(phi, xi, dd)
% Risoluzione del sistema lineare di una spline cubica con condizioni
% not-a-knot per la determinazione dei fattori m_i necessari per la
% costruzione dell'espressione della spline cubica not-a-knot.
%
```

```
% Input:
%   -phi: vettore dei fattori phi che definiscono la matrice dei
%   coefficienti (lunghezza n-1);
%   -xi: vettore dei fattori xi che definiscono la matrice dei
%   coefficienti (lunghezza n-1);
%   -dd: vettore delle differenze divise (lunghezza n-1).
% Output:
%   -m: vettore riscritto con gli n+1 fattori m_i calcolati.
%
% Autore: Tommaso Papini,
% Ultima modifica: 24 Ottobre 2012, 12:44 CEST.

function [m] = risolviSistemaSplineNotAKnot(phi, xi, dd)
    dd=[6*dd(1); 6*dd; 6*dd(length(dd))];
    l=zeros(length(xi)+1, 1);
    u=zeros(length(xi)+2, 1);
    w=zeros(length(xi)+1, 1);
    u(1)=1;
    w(1)=0;
    l(1)=phi(1)/u(1);
    u(2)=2-phi(1);
    w(2)=xi(1)-phi(1);
    l(2)=phi(2)/u(2);
    u(3)=2-l(2)*w(2);
    for i=4:length(xi)
        w(i-1)=xi(i-2);
        l(i-1)=phi(i-1)/u(i-1);
        u(i)=2-l(i-1)*w(i-1);
    end
    w(length(xi))=xi(length(xi)-1);
    l(length(xi))=(phi(length(xi))-xi(length(xi)))/u(length(xi));
    u(length(xi)+1)=2-xi(length(xi))-l(length(xi)-1)*w(length(xi)-1);
    w(length(xi)+1)=xi(length(xi));
    l(length(xi)+1)=0;
    u(length(xi)+2)=1;

    y=zeros(length(xi)+2, 1);
    y(1)=dd(1);
    for i=2:length(xi)+2
        y(i)=dd(i)-l(i-1)*y(i-1);
    end
    m=zeros(length(xi)+2, 1);
    m(length(xi)+2)=y(length(xi)+2)/u(length(xi)+2);
    for i=length(xi)+1:-1:1
        m(i)=(y(i)-w(i)*m(i+1))/u(i);
    end
    m(1)=m(1)-m(2)-m(3);
    m(length(xi)+2)=m(length(xi)+2)-m(length(xi)+1)-m(length(xi));
end
```

Possiamo quindi, a questo punto, definire il seguente Codice MATLAB per la determinazione di una qualsiasi *spline* cubica (naturale o con condizioni *not-a-knot*):

Codice 4.11: Calcolo delle espressioni di una *spline* (naturale o con condizioni *not-a-knot*).

```
% s = splineCubica(ptx, fi, nak)
% Determina le espressioni degli n polinomi che formano una spline
% cubica naturale o con condizioni not-a-knot.
%
% Input:
%   -ptx: vettore contenente gli n+1 nodi di interpolazione;
%   -fi: vettore contenente i valori assunti dalla funzione da
%   approssimare nei nodi in ptx;
%   -nak: true se la spline implementa condizioni not-a-knot, false se
%   invece è una spline naturale.
% Output:
%   -s: il vettore contenente le n espressioni dei polinomi costituenti
%   la spline.
%
% Autore: Tommaso Papini,
% Ultima modifica: 24 Ottobre 2012, 12:50 CEST.

function [s] = splineCubica(ptx, fi, nak)
    phi = zeros(length(ptx)-2, 1);
    xi = zeros(length(ptx)-2, 1);
    dd = zeros(length(ptx)-2, 1);
    for i=2:length(ptx)-1
        hi = ptx(i) - ptx(i-1);
        hi1 = ptx(i+1) - ptx(i);
        phi(i) = hi/(hi+hi1);
        xi(i) = hi1/(hi+hi1);
        dd(i) = differenzaDivisa(ptx(i-1:i+1), fi(i-1:i+1));
    end
    if nak
        mi = risolviSistemaSplineNotAKnot(phi, xi, dd);
    else
        mi = risolviSistemaSplineNaturale(phi, xi, dd);
    end
    s = espressioniSplineCubica(ptx, fi, mi);
end
```

Data la natura polinomiale a tratti delle *spline* è conveniente definire il seguente Codice, che valuta una *spline* su una serie di punti:

Codice 4.12: Valutazione di una *spline* su una serie di punti.

```
% sx = valutaSpline(ptx, s, pt)
% Valuta una spline su una serie di punti.
%
% Input:
%   -ptx: vettore contenente gli n+1 nodi di interpolazione;
%   -s: vettore contenente le espressioni degli n polinomi che
%   definiscono la spline;
%   -pt: vettore di m punti su cui si vuole valutare la spline.
% Output:
%   -sx: vettore di m valori contenente la valutazione dei punti in pt
%   della spline (NaN se un punto non è valutabile).
```

```
%
% Autore: Tommaso Papini,
% Ultima modifica: 24 Ottobre 2012, 11:57 CEST.

function [sx] = valutaSpline(ptx, s, pt)
    sx = zeros(length(pt), 1);
    for i=1:length(pt)
        if pt(i)<ptx(1) || pt(i)>ptx(length(ptx))
            str=spintf('%5.4f non valutato: punto esterno all''
                intervallo [%5.4f, %5.4f].', pt(i), ptx(1), ptx(length(
                ptx))); disp(str);
            sx(i)=NaN;
        else
            for j=1:length(ptx)
                if pt(i)>=ptx(j-1) && pt(i)<=ptx(j)
                    f = inline(s(j));
                    sx(i)=f(pt(i));
                    break;
                end
            end
        end
    end
end
```

4.9 Approssimazione polinomiale ai minimi quadrati

Supponiamo adesso di avere a disposizione una serie di coppie di ascisse e relative ordinate che descrivono il comportamento di un fenomeno (ovviamente saranno principalmente dati di tipo sperimentale). Vogliamo determinare un polinomio di grado m

$$y(x) = \sum_{k=0}^{m} a_k x^k, \tag{4.42}$$

che meglio approssimi le coppie di dati sperimentali

$$(x_i, y_i), \qquad i = 0, 1, \ldots, n, \qquad n \geq m. \tag{4.43}$$

Supporremo, di seguito, che almeno $m+1$ ascisse x_i delle coppie (4.43) siano tra loro *distinte*.
Definiamo allora i vettori

$$\underline{y} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix}, \qquad z = \begin{pmatrix} z_0 \\ z_1 \\ \vdots \\ z_n \end{pmatrix} \equiv \begin{pmatrix} \sum_{k=0}^{m} a_k x_0^k \\ \sum_{k=0}^{m} a_k x_1^k \\ \vdots \\ \sum_{k=0}^{m} a_k x_n^k \end{pmatrix}, \tag{4.44}$$

dove $\underline{y}$ è il vettore dei *valori misurati*, mentre $\underline{z}$ è il vettore dei *valori previsti* (ovvero quelli calcolati utilizzando il polinomio approssimante "attuale") in corrispondenza delle ascisse x_i, $i = 0, 1, \ldots, n$. Quindi si tratta di determinare il

vettore

$$\underline{a} = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{pmatrix},$$

che minimizzi la quantità

$$||y - z||_2^2 = \sum_{i=0}^{n} |y_i - z_i|^2,$$

la quale corrisponde al residuo (3.30) di un sistema lineare sovradeterminato, come visto in Sezione 3.8. Quindi questo vettore $\underline{a}$ definisce il polinomio (4.42) di approssimazione ai *minimi quadrati*. Si vede facilmente che il vettore $\underline{z}$ in (4.44) può essere scritto come

$$\underline{z} = V\underline{a},$$

con la matrice $V \in \mathbb{R}^{n+1\times m+1}$ che è una matrice di tipo *Vandermonde* (in realtà la trasposta di una matrice di tipo *Vandermonde*), ovvero

$$\begin{pmatrix} x_0^0 & x_0^1 & \dots & x_0^m \\ x_1^0 & x_1^1 & \dots & x_1^m \\ \vdots & \vdots & & \vdots \\ x_n^0 & x_n^1 & \dots & x_n^m \end{pmatrix}.$$

Teorema 4.12. *Se almeno $m+1$ ascisse x_i delle coppie (4.43) sono tra loro distinte, allora la matrice V ha rango massimo pari ad $m+1$.*

Quindi risulta che il problema della determinazione del polinomio approssimante (4.42) equivale alla risoluzione, nel senso dei *minimi quadrati*, del sistema lineare determinato

$$V\underline{a} = \underline{y},$$

in cui la matrice dei coefficienti V ha rango massimo.
Quindi, per quanto già visto in Sezione 3.8, ed in particolare per il Teorema 3.9, possiamo fattorizzare

$$V = QR \equiv \begin{pmatrix} \hat{R} \\ O \end{pmatrix}, \qquad \hat{R} \in \mathbb{R}^{m+1\times m+1},$$

con Q ortogonale ed $\hat{R}$ triangolare superiore e nonsingolare. Quindi per le stesse argomentazioni già viste in Sezione 3.8 (in riferimento alla minimizzazione del residuo), otteniamo che la soluzione, nel senso dei minimi quadrati, è data da

$$\underline{a} = \hat{R}^{-1}\underline{g_1},$$

con

$$Q^T\underline{y} \equiv \underline{g} = \begin{pmatrix} \underline{g_1} \\ \underline{g_2} \end{pmatrix}, \qquad \underline{g_1} \in \mathbb{R}^{m+1}.$$

Si evince quindi che la soluzione $\underline{a}$ esiste ed è unica e, di conseguenza, tale è il polinomio di approssimazione ai minimi quadrati (4.42). Per quanto riguarda il costo computazionale e l'occupazione di memoria valgono le stesse considerazioni fatte in Sezione 3.8 per il costo della fattorizzazione QR di Householder.
Osserviamo che nel caso in cui $m = n$ il polinomio di approssimazione ai minimi quadrati coincide con il polinomio interpolante le $n + 1$ ascisse: infatti il vettore $\underline{g_2}$ risulta vuoto e, pertanto, il polinomio interpola tutti i valori.

Spesso, anziché minimizzare la norma del vettore residuo

$$||V\underline{a} - \underline{y}||_2^2,$$

risulta più utile minimizzare un *vettore pesato*, ovvero

$$||D(V\underline{a} - \underline{y})||_2^2,$$

dove

$$D = \begin{pmatrix} w_0 & & \\ & \ddots & \\ & & w_n \end{pmatrix}$$

è la matrice diagonale dei pesi, con $w_i > 0$. Assegnando un peso maggiore in corrispondenza di un'ascissa piuttosto che un'altra si da maggior peso a quell'ascissa, facendo sì che la funzione venga approssimata con più accuratezza vicino a quel valore. Si può utilizzare una tecnica del genere quando si hanno dati sperimentali più affidabili di altri (perché raccolti con strumenti migliori, in condizioni più favorevoli, ecc...).

Esercizi

Esercizio 4.1. *Sia* $f(x) = 4x^2 - 12x + 1$. *Determinare* $p(x) \in \Pi_4$ *che interpola* $f(x)$ *sulle ascisse* $x_i = i$, $i = 0, \ldots, 4$.

Soluzione.
Poiché la funzione è un polinomio di grado 2 e $\Pi_2 \subset \Pi_4$, il polinomio interpolante sulle ascisse $x_i = i$, $i = 0, \ldots, 4$ coincide con la funzione stessa: $p(x) = f(x)$.

Esercizio 4.2 (Algoritmo di Horner)**.** *Dimostrare che il seguente algoritmo,*

```
p = a(n+1)
for k = n:-1:1
    p = p*x +a(k)
end
```

valuta il polinomio (4.4) nel punto x, *se il vettore* $\underline{a}$ *contiene i coefficienti del polinomio* $p(x)$ *(Osservare che in* MATLAB *i vettori hanno indice che parte da* 1, *invece che da* 0*).*

Soluzione.
Un generico polinomio è rappresentabile, rispetto alla base delle potenze, come

$$\sum_{k=0}^{n} a_k x^k,$$

con a_k coefficiente reale del k-esimo polinomio costituente la base delle potenze, x^k. Raccogliendo di volta in volta un fattore x si ottiene

$$\begin{aligned}\sum_{k=0}^{n} a_k x^k &= a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n = \\ &= a_0 + x(a_1 + x(a_2 + \cdots + x(a_{n-1} + a_n x)\dots)).\end{aligned}$$

Quindi si vede facilmente che, sostituendo ad x il punto in cui si vuol valutare il polinomio ed eseguendo le operazioni (prodotti e somme) nell'ordine indicato dalle parentesi, si ottiene esattamente l'Algoritmo di Horner, che quindi risulta essere esatto ai fini della valutazione di polinomio in un dato punto conoscendo i coefficienti $\{a_i\}$ rispetto alla base delle potenze.

Esercizio 4.3. *Dimostrare il Lemma 4.1.*

Soluzione.

1. Si distinguono i due casi:

   - se $k = i$, $L_{i,n}(x_i) = \prod_{j=0, j\neq k}^{n} \frac{x_i - x_j}{x_i - x_j} = \prod_{j=0, j\neq k}^{n} 1 = 1$:
   - se $k \neq i$, $\exists j = k$ tale che $x_k - x_j = 0$ quindi $L_{i,n}(x_k) = 0$.

2. Gli n elementi della produttoria sono polinomi di grado 1 quindi $L_{k,n}(x)$ ha grado n; ognuno di questi polinomi è monico quindi il coefficiente di principale di $L_{k,n}(x)$ è $\frac{1}{\prod_{j=0, j\neq k}^{n} (x_k - x_j)}$.

3. Si prova che $\sum_{k=0}^{n} c_k L_{k,n}(x) = 0$ se $c_k = 0$ $\forall x \in \mathbb{R}$, $k = 0, \dots, n$. Nelle ascisse di interpolazione si ha $L_{k,k}(x_k) = 1$ quindi, affinché la quantità si annulli è necessario che $c_k = 0$ quindi sono linearmente indipendenti; essendo n costituiscono una base per Π_n.

Esercizio 4.4. *Dimostrare il Lemma 4.2.*

Soluzione.

1. Per induzione: $w_0(x) = 1 \in \Pi'_0$, $w_{k+1}(x) = (x - x_k)w_k \in \Pi'_{k+1}$ in quanto $w_k \in \Pi'_k$ per induzione e $(x - x_k) \in \Pi'_1$.
2. Risulta $w_{k+1}(x) = (x - x_k)w_k = (x - x_k)(x - x_{k-1})w_{k-1} = \ldots = (x - x_k)(x - x_{k-1}) \ldots (x - x_0) = \prod_{j=0}^{k} (x - x_j)$.
3. Dal punto precedente, per $j \leq k$, un fattore della produttoria è del tipo $(x - x_j)$ quindi $w_{k+1}(x_j) = 0$.
4. I vari $w_i(x)$ sono linearmente indipendenti: $c_i w_i(x_j) = 0$ implica $c_i = 0$ per $j \neq i$ in virtù del punto precedente; essendo k, costituiscono una base per Π_k.

Esercizio 4.5. *Dimostrare il Teorema 4.4.*

Soluzione.

1. Linearità delle differenze divise:

$$\begin{aligned}(\alpha f + \beta g)[x_0, \ldots, x_r] &= \sum_{k=0}^{r} \frac{\alpha f_k + \beta g_k}{\prod_{j=0, j\neq k}^{r} (x_k - x_j)} = \\ &= \sum_{k=0}^{r} \frac{\alpha f_k}{\prod_{j=0, j\neq k}^{r} (x_k - x_j)} + \sum_{k=0}^{r} \frac{\beta g_k}{\prod_{j=0, j\neq k}^{r} (x_k - x_j)} = \\ &= \alpha \sum_{k=0}^{r} \frac{f_k}{\prod_{j=0, j\neq k}^{r} (x_k - x_j)} + \beta \sum_{k=0}^{r} \frac{g_k}{\prod_{j=0, j\neq k}^{r} (x_k - x_j)} = \\ &= \alpha f[x_0, \ldots, x_r] + \beta g[x_0, \ldots, x_r].\end{aligned}$$

2. La permutazione degli indici è possibile poiché somme e prodotti sono operazioni commutative.
3. Differenze divise e polinomi: dato che $f(x)$ è un polinomio ed il polinomio interpolante è unico,

$$f(x) = \sum_{i=0}^{k} a_i x^i = \sum_{i=0}^{k} f[x_0, \ldots, x_k] w_k(x) = p_k(x).$$

Il termine k-esimo è $a_k x^k = f[x_0, \ldots, x_k] w_k(x)$ quindi $f[x_0, \ldots, x_k] = a_k$ poiché $w_k(x) \in \Pi'_k$. I termini con $r > k$ non compaiono nella funzione dunque hanno coefficiente $f[x_0, \ldots, x_r] = 0$.

4. Derivabilità: sia $p(x)$ il polinomio interplante allora $e(x) = f(x) - p(x) \in C^{(r)}([a,b])$ ed $e(x_i) = 0$ per $i = 0, \ldots, r$. Per il teorema di Rolle sulle funzioni continue, $\exists \xi_i^{(1)} \in [x_i, x_{i+1}]$ tali che $e'(\xi_i^{(1)}) = 0$ per $i = 0, \ldots, r-1$. Reiterando, $\exists \xi_i^{(2)} \in \left[\xi_i^{(1)}, \xi_{i+1}^{(1)}\right]$ tali che $e''(\xi_i^{(2)}) = 0$ per $i = 0, \ldots, r-2$ e così via; infine $\exists \xi^{(r)} \in [a,b]$ tali che $e^{(r)}(\xi^{(r)}) = 0$. Segue $e^{(r)}(x) = f^{(r)}(x) - p^{(r)}(x) = f^{(r)}(x) - r! f[x_0, \ldots, x_r] = 0$ ovvero $f[x_0, \ldots, x_r] = \frac{f^{(r)}(\xi^{(r)})}{r!}$.

5. Ricorsività:

$$
\begin{aligned}
&\frac{f[x_1, \ldots, x_r] - f[x_0, \ldots, x_{r-1}]}{x_r - x_0} = \\
&= \left[\sum_{k=1}^{r} \frac{f_k}{\prod_{j=1, j\neq k}^{r} (x_k - x_j)} + \sum_{k=0}^{r-1} \frac{f_k}{\prod_{j=0, j\neq k}^{r-1} (x_k - x_j)} \right] \frac{1}{x_r - x_0} = \\
&= \left[\sum_{k=1}^{r} \frac{f_k (x_j - x_0)}{\prod_{j=0, j\neq k}^{r} (x_k - x_j)} - \sum_{k=0}^{r-1} \frac{f_k (x_j - x_k)}{\prod_{j=0, j\neq k}^{r} (x_k - x_j)} \right] \frac{1}{x_r - x_0} = \\
&= \left[\sum_{k=0}^{r} \frac{f_k (x_j - x_0)}{\prod_{j=0, j\neq k}^{r} (x_k - x_j)} - \sum_{k=0}^{r} \frac{f_k (x_j - x_k)}{\prod_{j=0, j\neq k}^{r} (x_k - x_j)} \right] \frac{1}{x_r - x_0} = \\
&= \left[\sum_{k=0}^{r} \frac{f_k (x_j - x_0) - f_k (x_j - x_k)}{\prod_{j=0, j\neq k}^{r} (x_k - x_j)} \right] \frac{1}{x_r - x_0} = \\
&= \left[\sum_{k=0}^{r} \frac{f_k (x_r - x_0)}{\prod_{j=0, j\neq k}^{r} (x_k - x_j)} \right] \frac{1}{x_r - x_0} = \\
&= (x_r - x_0) \left[\sum_{k=0}^{r} \frac{f_k}{\prod_{j=0, j\neq k}^{r} (x_k - x_j)} \right] \frac{1}{x_r - x_0} =
\end{aligned}
$$

$$= \sum_{k=0}^{r} \frac{f_k}{\prod_{j=0, j\neq k}^{r}(x_k - x_j)} = f[x_0, \ldots, x_r].$$

Esercizio 4.6. *Costruire una* `function` *in* MATLAB *che implementi in modo efficiente l'algoritmo del calcolo delle differenze divise.*

Soluzione.
Per l'implementazione in MATLAB dell'algoritmo per il calcolo delle differenze divise si veda il Codice 4.3 a pagina 135.

Esercizio 4.7 (Algoritmo di Horner generalizzato)**.** *Dimostrare che il seguente algoritmo, che riceve in ingresso i vettori* $\underline{x}$ *ed* $\underline{f}$ *prodotti dalla* `function` *dell'Esercizio 4.6, valuta il corrispondente polinomio interpolante di Newton in un punto* xx *assegnato.*

```
p = f(n+1)
for k = n:-1:1
    p = p*(xx-x(k)) +f(k)
end
```

Qual'è il suo costo computazionale? Confrontarlo con quello dell'Algoritmo dell'Esercizio 4.6. Costruire, quindi, una corrispondente `function` MATLAB *che lo implementi efficientemente (contemplare la possibilità che* xx *sia un vettore).*

Soluzione.
Essendo un polinomio in forma di Newton esprimibile come

$$p_n(x) = \sum_{k=0}^{n} f[x_0, \ldots, x_k] w_k(x)$$

ed essendo il $(k+1)$-esimo polinomio della base di Newton esprimibile come

$$w_{k+1} = \prod_{j=0}^{k} (x - x_j),$$

vediamo che, come nel caso dell'Esercizio 4.2, possiamo raggruppare i vari $(x - x_i)$ e considerare le differenze divise come coefficienti rispetto alla base di Newton. Si ottiene quindi:

$$\sum_{k=0}^{n} f[x_0, \ldots, x_k] w_k(x) = f[x_0] + (x - x_0)(f[x_0, x_1] + \ldots$$
$$\cdots + (x - x_{n-2})(f[x_0, \ldots, x_{n-1}] + f[x_0, \ldots, x_n](x - x_{n-1})) \ldots).$$

Quindi, eseguendo le operazioni nell'ordine indicato, otteniamo l'algoritmo di Horner generalizzato, il cui costo risulta essere di $3n$ `flop`, in quanto ad ogni iterazione vengono eseguiti una sottrazione, un prodotto ed una somma. Il costo dell'Algoritmo per il calcolo delle differenze divise risulta invece essere pari a $(\frac{3}{2}n^2 - 3n)$ `flop`, in quanto per calcolare ogni differenza divisa vengono eseguite due sottrazioni ed una divisione (3 `flop`) ed il numero di differenze divise da calcolare è pari al numero di elementi triangolari inferiori di una matrice $n \times n$ (quindi $\frac{1}{2}n^2$) meno gli elementi della prima colonna (che sono n) che sono dati.
Nell'implementazione MATLAB, proposta di seguito, si è considerato il caso in cui `xx` sia un vettore di punti semplicemente reiterando m volte, con m dimensione del vettore `xx`, l'algoritmo di Horner generalizzato. in questo caso il costo computazionale risulta essere $3mn$ `flop`.

Codice 4.13: Algoritmo di Horner generalizzato.

```
% p = hornerGeneralizzato(x, f, xx)
% Algoritmo di Horner generalizzato che valuta un polinomio in forma di
% Newton su un vettore di punti.
%
% Input:
%   -x: vettore contenente le ascisse di interpolazione;
%   -f: vettore contenente le differenze divise;
%   -xx: vettore di punti sui quali valutare il polinomio.
% Output:
%   -p: vettore contenente le valutazioni del polinomio sui punti in
%   xx.
%
% Autore: Tommaso Papini,
% Ultima modifica: 26 ottobre 2012, 18:59 CEST

function [p] = hornerGeneralizzato(x, f, xx)
    n = length(x)-1;
    p = zeros(length(xx), 1);
    for i=1:length(xx)
        p(i) = f(n+1);
        for k=n:-1:1
            p = p*(xx(i)-x(k)) +f(k);
        end
    end
end
```

Esercizio 4.8. *Costruire una* `function` MATLAB *che implementi in modo efficiente l'algoritmo del calcolo delle differenze divise per il polinomio di Hermite.*

Soluzione.
Consultare il Codice 4.4 a pagina 137 per il file corrispondente all'implementazione

in MATLAB dell'algoritmo per il calcolo delle differenze divise ottimizzato nel caso di ascisse di Hermite.

Esercizio 4.9. *Si consideri la funzione*

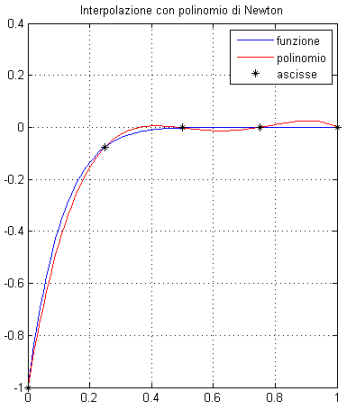
$$f(x) = (x-1)^9.$$

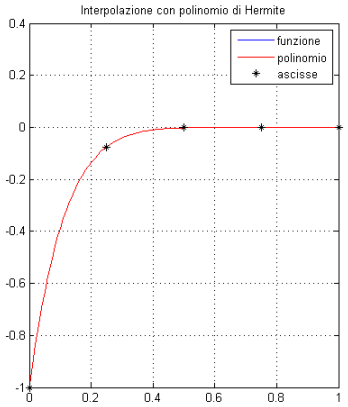
Utilizzando le `function` *degli Esercizi 4.6 e 4.8, valutare i polinomi interpolanti di Newton e di Hermite sulle ascisse*

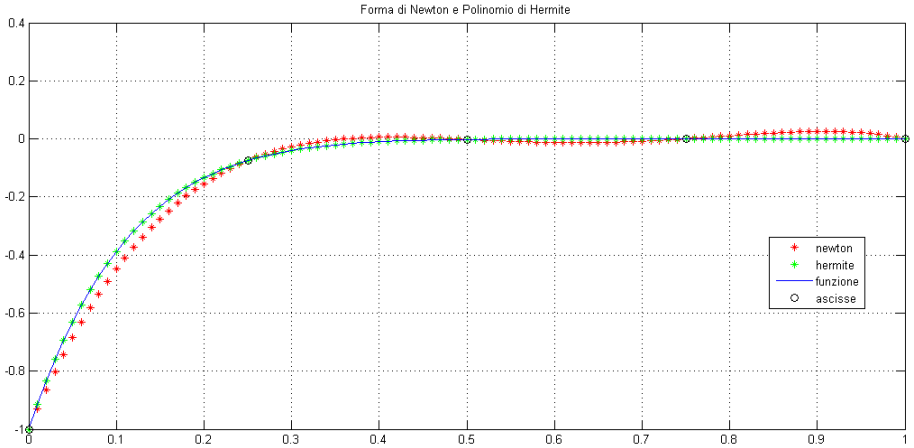
$$0, 0.25, 0.5, 0.75, 1,$$

per `x=linspace(0,1,101)`*. Raffigurare, quindi, (e spiegare) i risultati.*

Soluzione.
I seguenti grafici mostrano i risultati ottenuti interpolando la funzione sulle ascisse indicate con i polinomi di Newton e di Hermite:







Dai grafici ottenuti si vede che l'interpolazione tramite polinomio di Newton approssima abbastanza bene la funzione, mentre utilizzando il polinomio di Hermite si ha una totale coincidenza tra la funzione originaria ed il polinomio interpolante. Questo è dovuto al fatto che il polinomio di Hermite interpola la funzione su un totale di 10 ascisse, il che lo rende a tutti gli effetti un polinomio di grado 9, esattamente come la funzione originaria. Il polinomio di newton, invece, interpolando la funzione su 5 sole ascisse, risulta essere un polinomio di quarto grado.

Riferimenti MATLAB
Codice 4.14 (pagina 177)

Esercizio 4.10. *Quante ascisse di interpolazione equidistanti sono necessarie per approssimare la funzione* $\sin(x)$ *sull'intervallo* $[0, 2\pi]$*, con un errore di interpolazione inferiore a* 10^{-6}*?*

Soluzione.
Considerando l'espressione dell'errore d'interpolazione (4.16) si può massimizzare $\left|f^{(n+1)}(\xi_x)\right| \leq 1$ in quando le derivate di $f(x)$ sono $\pm\sin x$, $\pm\cos x$ quindi limitate; inoltre $w_{n+1}(x) = \prod_{j=0}^{n}(x - x_j) \leq \prod_{j=0}^{n} 2\pi = 2\pi^{n+1}$ poiché $|x - x_j| \leq 2\pi$, $\forall j$. Segue $e(x) = \frac{1}{(n+1)!}(2\pi)^{n+1}$; eseguendo lo script MATLAB, risulta `e(26) = 3.265e-007`$\leq 10^{-6}$.

Riferimenti MATLAB
Codice 4.15 (pagina 177)

Esercizio 4.11. *Verificare sperimentalmente che, considerando le ascisse di interpolazione equidistanti (4.20) su cui si definisce il polinomio* $p(x)$ *interpolante* $f(x)$*, l'errore* $||f - p||$ *diverge, al crescere di* n*, nei seguenti due casi:*

1. *esempio di Runge:*

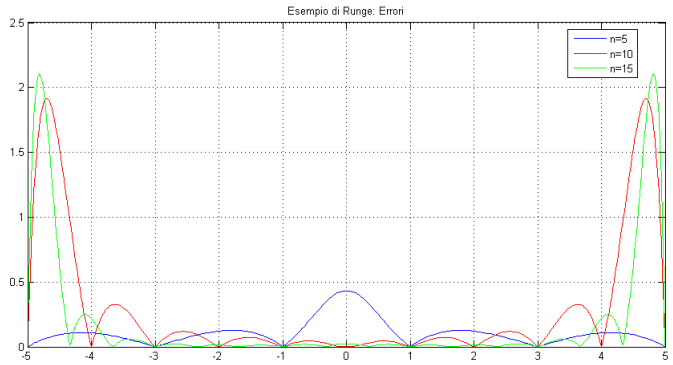
$$f(x) = \frac{1}{1 + x^2}, \qquad [a, b] \equiv [-5, 5];$$

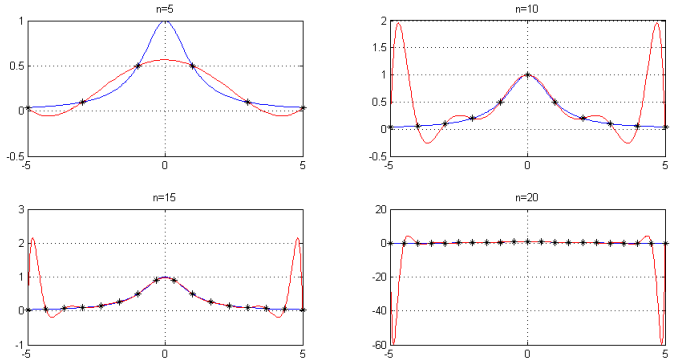
2. *esempio di Bernstein:*

$$f(x) = |x|, \qquad [a, b] \equiv [-1, 1].$$

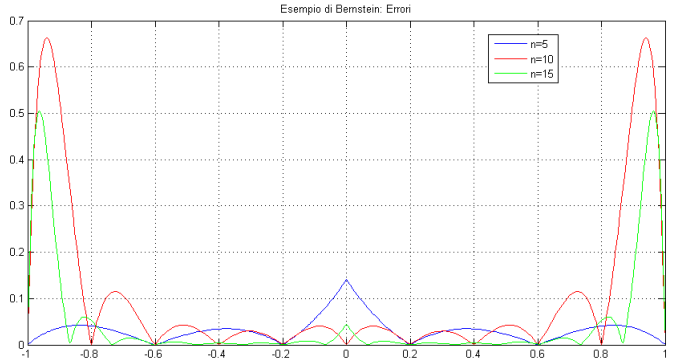
Soluzione.

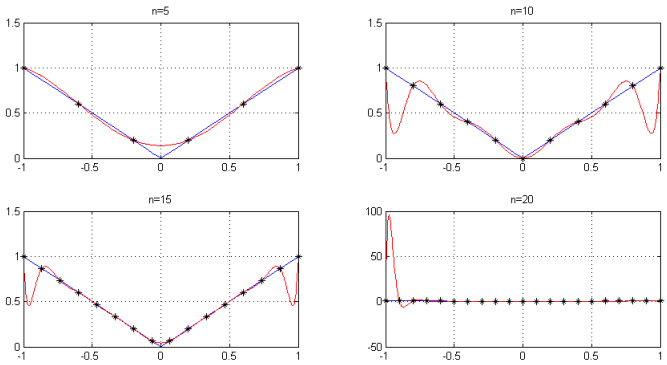
- Esempio di Runge:
  Osserviamo i seguenti grafici che mostrano, rispettivamente, l'errore commesso con $n = 5, 10, 15$ e l'interpolazione della funzione per $n = 5, 10, 15, 20$:





- Esempio di Bernstein: Come per l'esempio di Runge, proponiamo i grafici dell'errore e dell'interpolazione per l'esempio di Bernstein:





Da i grafici proposti per le due funzioni d'esempio deduciamo che l'errore commesso tende a divergere all'aumentare di n con le ascisse di interpolazione scelte equidistanti, ovvero uniformemente distribuite sull'intervallo di interpolazione. Vediamo infine i valori dell'errore massimo di interpolazione commesso nei due esempi per $n = 5, 10, 15, 20$:

n	**Errore**	
	Runge	**Bernstein**
5	0.4327	0.0422
10	0.3276	0.1149
15	2.1076	0.5054
20	59.8223	95.1889

Riferimenti MATLAB
Codice 4.16 (pagina 178)

Esercizio 4.12. *Dimostrare che, se $x \in [-1, 1]$, allora:*

$$\tilde{x} \equiv \frac{a+b}{2} + \frac{b-a}{2} x \in [a, b].$$

Viceversa, se $\tilde{x} \in [a, b]$, allora:

$$x \equiv \frac{2\tilde{x} - a - b}{b-1} \in [-1, 1].$$

Concludere che è sempre possibile trasformare il problema di interpolazione (4.1)-(4.2) in uno definito sull'intervallo $[-1, 1]$, e viceversa.

Soluzione.

- $x \in [-1, 1] \Rightarrow \tilde{x} \in [a, b]$:

  - se $x = -1$, $\tilde{x} = \frac{a+b}{2} - \frac{b-a}{2} = a$;
  - se $x = 1$, $\tilde{x} = \frac{a+b}{2} + \frac{b-a}{2} = b$.

- $\tilde{x} \in [a, b] \Rightarrow x \in [-1, 1]$:
  - se $\tilde{x} = a$, $x = \frac{2a-a-b}{b-a} = -1$;
  - se $\tilde{x} = b$, $x = \frac{2b-a-b}{b-a} = 1$;

Esercizio 4.13. *Dimostrare le proprietà dei polinomi di Chebyshev di I specie (4.24) elencate nel Teorema 4.9.*

Soluzione.
Prime proprietà

1. $T_k(x)$ è un polinomio di grado esatto k:
   - $k = 0$: $T_0(x) = 1$ polinomio di grado 0;
   - $k > 0$: $T_{k+1}(x) = 2xT_k(x) - T_{k-1}(x)$ dove, per ipotesi induttiva, $T_k(x)$ è un polinomio di grado k quindi $T_{k+1}(x)$ è un polinomio di grado $k+1$.
2. Il coefficiente principale di $T_k(x)$ è 2^{k-1}, $k = 1, 2, \ldots$:
   - $k = 1$: $T_1(x) = x$ e $2^{1-1} = 1$;
   - $k > 1$: $T_{k+1}(x) = 2xT_k(x) - T_{k-1}(x)$ dove, per ipotesi induttiva, il coefficiente principale di $T_k(x)$ è 2^{k-1} quindi il coefficiente principale di $T_{k+1}(x)$ è $2 \cdot 2^{k-1} = 2^k$.
3. La famiglia di polinomi $\{\hat{T}_k\}$, in cui
$$\hat{T}_0(x) = T_0(x), \quad \hat{T}_k(x) = 2^{1-k}T_k(x), \quad k = 1, 2, \ldots,$$
è una famiglia di polinomi monici di grado k, $k = 1, 2, \ldots$:
   - grado k: il grado di $\hat{T}_k$ coincide con il grado di $T_k(x)$ che è k;
   - monici: il coefficiente principale di $\hat{T}_k$ è 2^{1-k} dunque $2^{1-k}2^{k-1} = 1$ il polinomio è monico.
4. Ponendo $x = \cos\theta$, $\theta \in [0, \pi]$, si ottiene $T_k(x) = T_k(\cos\theta) = \cos k\theta, \quad k = 0, 1, \ldots$:
   - $k = 0$: $T_0(\cos\theta) = \cos 0\theta = 1 = T_0(x)$;
   - $k = 1$: $T_0(\cos\theta) = \cos\theta = x = T_1(x)$:

- $k > 1$: $T_{k+1}(\cos\theta) = 2\cos\theta T_k(\cos\theta) - T_{k-1}(\cos\theta)$ per ipotesi induttiva, $T_k(\cos\theta) = \cos k\theta$ e $T_{k-1}(\cos\theta) = \cos(k-1)\theta$ dunque $T_{k+1}(\cos\theta) = 2\cos\theta\cos k\theta - \cos(k-1)\theta = \cos(k+1)\theta + \cos(k-1)\theta - \cos(k-1)\theta = \cos(k+1)\theta = T_{k+1}(x)$.

Teorema 4.9

- Radici del polinomio: $T_k(x) = T_k(\cos\theta) = \cos k\theta = 0$ se $\cos k\theta = 0$ ovvero per $k\theta = \frac{\pi}{2} + i\pi$; segue $\theta_i = \frac{\frac{\pi}{2}+i\pi}{k} = \frac{(2i+1)\pi}{2k}$ cioè gli zeri del polinomio sono dati da $x_i^{(k)} = \cos\frac{(2i+1)\pi}{2k}$.
- Estremi: Agli estremi $\cos k\theta = \pm 1$ ovvero $k\theta = i\pi$; segue $\theta_i = \frac{i}{k}\pi$ dunque gli estremi sono assunti in $\xi_i^{(k)} = \cos\frac{i}{k}\pi$; in tali punti, poiché $\cos k\pi = (-1)^i$ la funzione vale $T_k(\xi_i^{(k)}) = (-1)^i$.
- Norme: dal punto precedente segue inoltre $||T_k|| = 1$ e $||\hat{T}_k|| = ||2^{1-k}T_k|| = ||2^{1-k}||$.
- Minima norma per $\hat{T}_k(x)$: supponiamo per assurdo che $\exists p \neq \hat{T}_k(x)$ monico tale che $||p|| < 2^{1-k} = ||\hat{T}_k||$, quindi $g(x) = \hat{T}_k - p(x) \in \Pi_{k-1}$ poiché entrambi monici di grado K. Studiando il segno di g si nota $sign(g(x_i)) = (-1)^i$ per $i = 0, \ldots, k$ ovvero ci sono k cambiamenti di segno quindi k radici cioè $g(x) \in \Pi_k$ ma, per ipotesi, $g(x) \in \Pi_{k-1}$ quindi $g(x) = 0$.

Esercizio 4.14. *Quali diventano le ascisse di Chebyshev (4.26), per un problema definito su un generico intervallo $[a, b]$?*

Soluzione.
Le ascisse di Chebyshev sono

$$x_{n-i} = \cos\frac{(2i+1)\pi}{2(n+1)} \in [-1, 1];$$

utilizzando il risultato dell'Esercizio 4.12 si ha

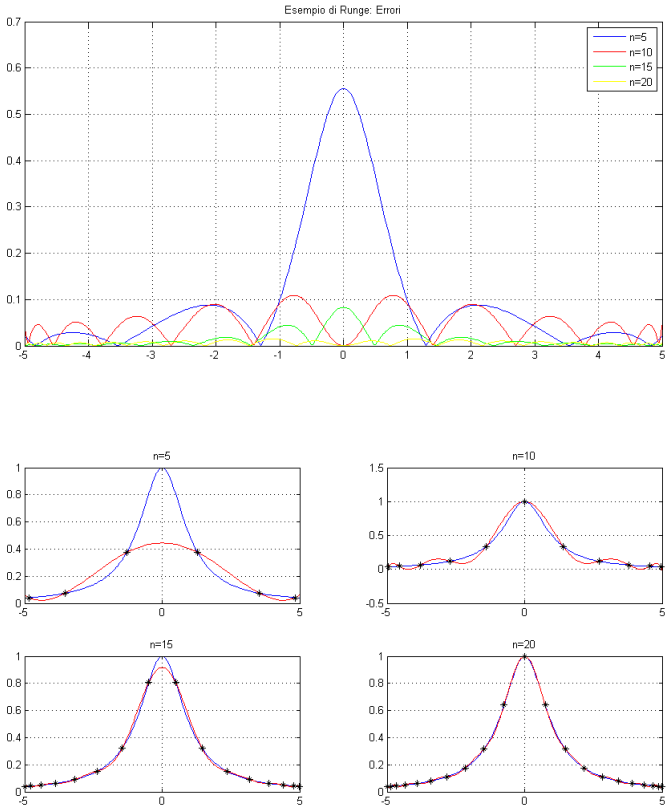
$$\tilde{x}_{n-i} = \frac{a+b}{2} - \frac{b-a}{2}\cos\frac{(2i+1)\pi}{2(n+1)} \in [a, b].$$

Tali $\tilde{x}_i$ definiscono le ascisse di Chebyshev per un generico intervallo $[a, b]$.

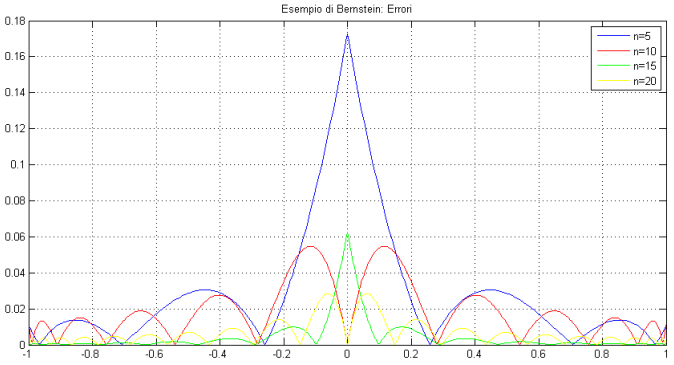
Esercizio 4.15. *Utilizzare le ascisse di Chebyshev (4.26) per approssimare gli esempi visti nell'Esercizio 4.11, per $n = 2, 4, 6, \ldots, 40$.*

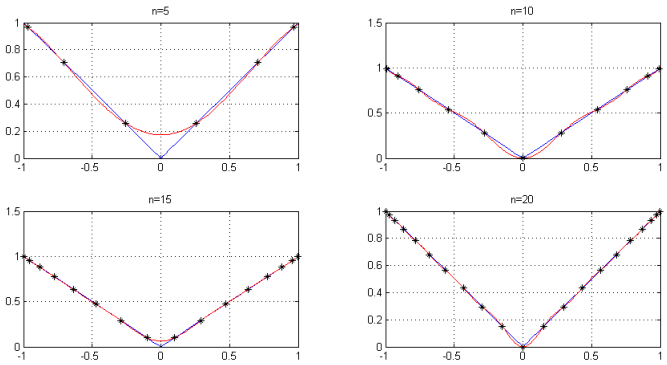
Soluzione.
Similmente a quanto visto nell'Esercizio 4.11 mostriamo di seguito i grafici degli errori e di interpolazione per $n = 5, 10, 15, 20$, rispetto agli esempi di Runge e Bernstein.

- Esempio di Runge:



- Esempio di Bernstein:





Di seguito proponiamo anche gli errori di interpolazione massimi commessi nei due esempi per $n = 5, 10, 15, 20$:

n	**Errore**	
	Runge	**Bernstein**
5	0.0881	0.0305
10	0.0897	0.0275
15	0.0831	0.0628
20	0.0153	0.0140

Si evince quindi, dai grafici e dai valori riportati, che la scelta delle ascisse di Chebyshev al posto di ascisse equidistanti è estremamente più conveniente, in quanto evita la divergenza dell'errore di interpolazione all'aumentare di n. Infatti l'errore commesso, all'aumentare di n risulta essere in generale molto stabile e, molto spesso, risulta anche in diminuzione.

Riferimenti MATLAB
Codice 4.17 (pagina 181)

Esercizio 4.16. *Verificare che la fattorizzazione LU della matrice dei coefficienti del sistema tridiagonale (4.40) è dato da:*

$$L = \begin{pmatrix} 1 & & & \\ l_2 & 1 & & \\ & \ddots & \ddots & \\ & & l_{n-1} & 1 \end{pmatrix}, \qquad U = \begin{pmatrix} u_1 & \xi_1 & & \\ & u_2 & \ddots & \\ & & \ddots & \xi_{n-2} \\ & & & u_{n-1} \end{pmatrix},$$

con

$$\begin{aligned} u_1 &= 2, \\ l_i &= \frac{\varphi_i}{u_{i-1}}, \\ u_i &= 2 - l_i \xi_{i-1}, \qquad i = 2, \ldots, n-1. \end{aligned}$$

Scrivere una `function` MATLAB *che implementi efficientemente la risoluzione della (4.40).*

Soluzione.
La matrice dei coefficienti

$$A = \begin{pmatrix} 2 & \xi_1 & & & \\ \varphi_2 & 2 & \xi_2 & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & \xi_{n-2} \\ & & & \varphi_{n-1} & 2 \end{pmatrix}$$

si vede essere *dominante per righe* in quanto, essendo

$$\varphi_i = \frac{h_i}{h_i + h_{i+1}}, \qquad \xi_i = \frac{h_{i+1}}{h_i + h_{i+1}}, \qquad i = 1, \ldots, n-1,$$

si vede facilmente che $\varphi_i + \xi_i = 1$. Quindi la matrice è fattorizzabile LU.
Moltiplichiamo adesso i termini L ed U e vediamo che il risultato sono proprio i termini della matrice A:

- $a_{11} = 1 \cdot u_1 = u_1 = 2$;
- $a_{12} = 1 \cdot \xi_1 + 0 \cdot u_2 = \xi_1$;
- $a_{i,i-1} = 0 \cdot \xi_{i-2} + l_i \cdot u_{i-1} + 1 \cdot 0 = l_i \cdot u_{i-1} = \frac{\varphi_i}{u_{i-1}} u_{i-1} = \varphi_i$, per $i > 1$;
- $a_{ii} = l_i \cdot \xi_{i-1} + 1 \cdot u_i = l_i \xi_{i-1} + 2 - l_i \xi_{i-1} = 2$, per $i > 1$;
- $a_{i,i+1} = l_i \cdot 0 + 1 \cdot \xi_i + 0 \cdot u_{i+1} = \xi_i$, per $i > 1$.

Quindi il sistema $A\underline{m} = \underline{d}$ si risolve come segue:

- si risolve $L\underline{y} = 6\underline{d}$:
  - $y_1 = 6d_1$,
  - $y_i = 6d_i - l_i y_{i-1}$ per $i = 2, \ldots, n-1$;
- si risolve il sistema $U\underline{m} = \underline{y}$:
  - $m_{n-1} = \frac{y_{n-1}}{u_{n-1}}$,
  - $m_i = \frac{y_i - \xi_i m_{i+1}}{u_i}$, per $i = n-2, \ldots, 1$.

Per quanto riguarda l'implementazione MATLAB dell'algoritmo per la risoluzione di suddetto sistema lineare, fare riferimento al Codice 4.9 a pagina 151. In questa implementazione si è deciso di non utilizzare la fattorizzazione LU vista in Sezione 3.2 (la cui implementazione in MATLAB è consultabile a pagina 78), ma se ne propone una versione ottimizzata basata sulle osservazioni appena fatte.

Esercizio 4.17. *Generalizzare la fattorizzazione del precedente Esercizio 4.16 al caso della matrice dei coefficienti del sistema lineare (4.41). Scrivere una corrispondente* `function` MATLAB *che risolva efficientemente questo sistema.*

Soluzione.
Generalizzando il risultato ottenuto nell'Esercizio 4.16, la fattorizzazione è della forma

$$L = \begin{pmatrix} 1 & & & \\ l_2 & 1 & & \\ & \ddots & \ddots & \\ & & l_{n+1} & 1 \end{pmatrix}, \qquad U = \begin{pmatrix} u_1 & w_1 & & \\ & u_2 & \ddots & \\ & & \ddots & w_n \\ & & & u_{n+1} \end{pmatrix}.$$

Se come prima moltiplichiamo i fattori L ed U ricaviamo le espressioni degli l_i, u_i e w_i:

- per $i = 4, \ldots, n-1$:
$$\begin{cases} u_i = 2 - l_i w_{i-1} \\ w_{i-1} = \xi_{i-2} \\ l_i = \frac{\varphi_{i-1}}{u_{i-1}} \end{cases};$$
- per $i = 3$:
$$\begin{cases} u_3 = 2 - l_3 w_2 \\ w_2 = \xi_1 - \varphi_1 \\ l_3 = \frac{\varphi_2}{u_2} \end{cases};$$
- per $i = 2$:
$$\begin{cases} u_2 = 2 - \varphi_1 \\ w_1 = 0 \\ l_2 = \frac{\varphi_1}{u_1} \end{cases};$$
- per $i = 1$:
$$u_1 = 1;$$
- per $i = n$:
$$\begin{cases} u_n = 2 - \xi_{n-1} - l_n w_{n-1} \\ w_{n-1} = \xi_{n-2} \\ l_n = \frac{\varphi_{n-1} - \xi_{n-1}}{u_{n-1}} \end{cases};$$
- per $i = n+1$:
$$\begin{cases} u_{n+1} = 1 \\ w_n = \xi_{n-1} \\ l_{n+1} = 0 \end{cases}$$

.

Quindi come prima avremo:

- risoluzione del sistema $L\underline{y} = 6\underline{d}$:
  - $y_1 = 6d_1$,
  - $y_i = 6d_i - l_i y_{i-1}$ per $i = 2, \ldots, n+1$;
- risoluzione del sistema $U\underline{\hat{m}} = \underline{y}$:
  - $\hat{m}_{n+1} = \frac{y_{n+1}}{u_{n+1}}$,
  - $\hat{m}_i = \frac{y_i - w_i \hat{m}_{i+1}}{u_i}$, per $i = n, \ldots, 1$;
- calcolo della soluzione:
  - $m_1 = \hat{m}_1 - \hat{m}_2 - \hat{m}_3$,
  - $m_i = \hat{m}_i$, per $i = 2, \ldots, n$,
  - $m_{n+1} = \hat{m}_{n+1} - \hat{m}_n - \hat{m}_{n-1}$.

Per l'implementazione in MATLAB della `function` relativa, si veda il Codice 4.10 a pagina 152.

Esercizio 4.18. *Scrivere una* `function` MATLAB *che, noti gli* $\{m_i\}$ *in (4.32), determini l'espressione, polinomiale a tratti, della spline cubica (4.36).*

Soluzione.
Per l'implementazione MATLAB dell'algoritmo che determina l'espressione polinomiale a tratti di una spline cubica si veda il Codice 4.8 a pagina 149. In questo algoritmo viene creato un vettore contenente le espressioni degli n polinomi che definiscono la spline cubica, applicando la (4.36), passando come input i nodi di interpolazione, i valori assunti dalla funzione in tali nodi e i fattori m_i, eventualmente calcolati utilizzando gli Algoritmi degli Esercizi 4.16 e 4.17.

Esercizio 4.19. *Costruire una* `function` MATLAB *che implementi le spline cubiche naturali e quelle definite dalle condizioni not-a-knot.*

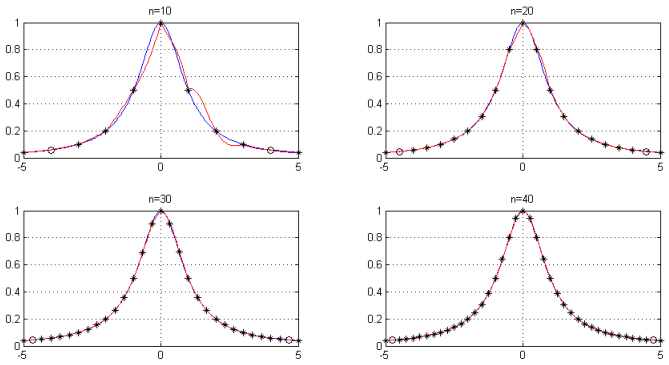
Soluzione.
Per l'implementazione si veda il Codice 4.11 a pagina 153, nel quale la scelta tra spline naturale e con condizioni not-a-knot dipende dal valore dell'input `nak` (rispettivamente, false e true). Quello che viene fatto è creare, a partire dai nodi di interpolazione e dai valori assunti dalla funzione in tali punti, i vettori $\underline{\varphi}$, $\underline{\xi}$ e delle differenze divise (si è costruita una nuova `function` che calcola una singola differenza divisa in modo indipendente dalle altre, in quanto la `function` che era stata scritta per la forma di Newton non è chiaramente riutilizzabile).

Viene quindi risolto il sistema lineare corrispondente per determinare i fattori m_i ed infine viene determinata l'espressione della spline, ovvero le espressioni degli n polinomi costituenti la spline cubica. È stata scritta, inoltre, una `function` MATLAB che, prendendo in input i nodi di interpolazione, l'espressione della spline ed un set di punti, restituisce la valutazione della spline in corrispondenza di tali punti.

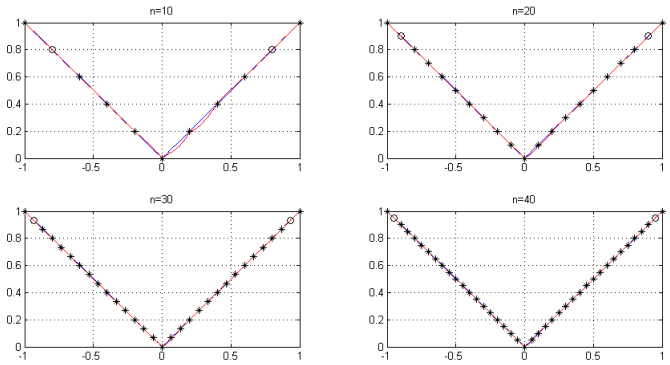
Esercizio 4.20. *Utilizzare la* `function` *dell'Esercizio 4.19 per approssimare, su partizioni (4.28) uniformi con* $n = 10, 20, 30, 40$*, gli esempi proposti nell'Esercizio 4.11.*

Soluzione.
I grafici ottenuti mostrano l'interpolazione delle due funzioni d'esempio tramite spline cubiche con condizioni *not-a-knot* (si osservi che nei grafici i due punti definiti, appunto, *not-a-knot*, sono indicati da un cerchio nero al posto di un asterisco come per tutti gli altri nodi). Non sono stati riportati i grafici riguardanti l'interpolazione tramite spline cubiche naturali in quanto tra i due tipi di grafici non sono presenti sostanziali differenze (eseguendo il file MATLAB relativo vengono disegnati entrambi i grafici per i due esempi).

- Esempio di Runge:



- Esempio di Bernstein:



Riferimenti MATLAB
Codice 4.18 (pagina 184)

Esercizio 4.21. *Interpretare la retta dell'Esercizio 3.32 come retta di approssimazione ai minimi quadrati dei dati.*

Soluzione.
Il problema ai minimi quadrati è dato da

$$\min_{a_1,a_2\in\mathbb{R}} \sum_{k=0}^{n} |y_i - a_1 x_i - a_2|^2$$

dove $\underline{y}$ è il vettore dei valori previsti per la retta. Il problema è esprimibile in forma matriciale come

$$\begin{pmatrix} 1 & x_0 \\ 1 & x_1 \\ \vdots & \vdots \\ 1 & x_n \end{pmatrix} \begin{pmatrix} a_2 \\ a_1 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_n \end{pmatrix};$$

si tratta di risolvere un sistema sovradeterminato che ha soluzione se almeno 2 ascisse sono distinte in quanto $r(x) \in \Pi_1$.

Esercizio 4.22. *È noto che un fenomeno ha un decadimento esponenziale, modellizzato come*

$$y = \alpha \cdot e^{-\lambda t},$$

in cui α e λ sono parametri positivi e incogniti. Riformulare il problema in modo che il modello sia di tipo polinomiale. Supponendo inoltre di disporre delle seguenti misure,

t_1	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
y_i	*5.22*	*4.00*	*4.28*	*3.89*	*3.53*	*3.12*	*2.73*	*2.70*	*2.20*	*2.08*	*1.94*

calcolare la stime ai minimi quadrati dei due parametri incogniti. Valutare il residuo e raffigurare, infine, i risultati ottenuti.

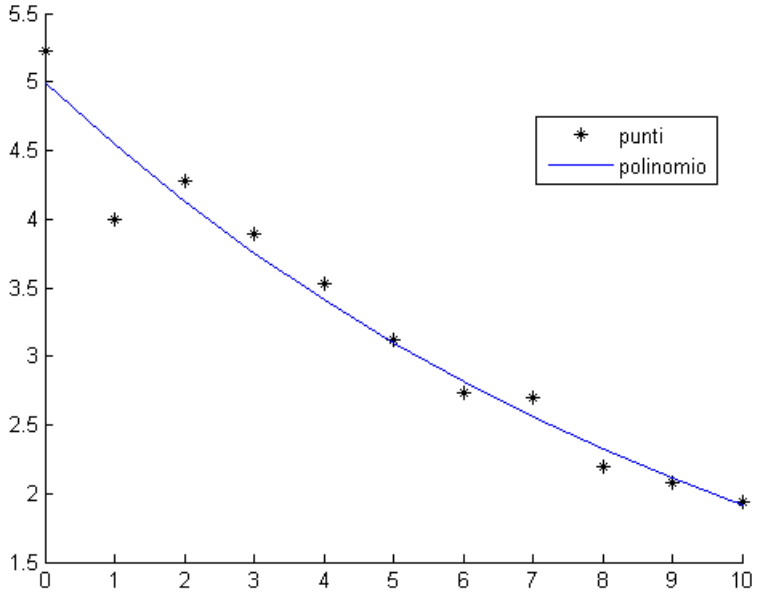
Soluzione.
Il problema in forma polinomiale è

$$\bar{y} = \bar{\alpha} + \bar{\lambda}t, \qquad \text{con } \bar{y} = \log y,\ \bar{\alpha} = \log \alpha \text{ e } \bar{\lambda} = -\lambda$$

infatti si ha $y = \alpha \cdot e^{-\lambda t} \Rightarrow \log y = \log(\alpha \cdot e^{-\lambda t}) \Rightarrow \log y = \log \alpha - \lambda t \Rightarrow \bar{y} = \bar{\alpha} + \bar{\lambda}t$.
Si tratta di risolvere il sistema lineare sovradeterminato

$$\begin{pmatrix} 1 & t_0 \\ 1 & t_1 \\ \vdots & \vdots \\ 1 & t_1 0 \end{pmatrix} \begin{pmatrix} \bar{\alpha} \\ \bar{\lambda} \end{pmatrix} = \begin{pmatrix} \bar{y}_0 \\ \bar{y}_1 \\ \vdots \\ \bar{y}_n \end{pmatrix}$$

mediante fattorizzazione QR (possibile poiché tutte le ascisse sono distinte) e ricavare $\begin{pmatrix} \bar{\alpha} \\ \bar{\lambda} \end{pmatrix}$ da qui $\begin{pmatrix} \alpha \\ \lambda \end{pmatrix} = \begin{pmatrix} e^{\bar{\alpha}} \\ -\bar{\lambda} \end{pmatrix}$.
I risultati ottenuti dallo `scritp` MATLAB sono $\begin{pmatrix} \alpha \\ \lambda \end{pmatrix} = \begin{pmatrix} 5.008 \\ 0.0959 \end{pmatrix}$ con un residuo $r = 0.1708$.



Riferimenti MATLAB
Codice 4.19 (pagina 185)

Codice degli esercizi

Codice 4.14: Esercizio 4.9.

```
% Esercizio 4.9
%
% Autore: Tommaso Papini,
% Ultima modifica: 28 Ottobre 2012, 16:15 CET

format short

f = inline('(x-1).^9');
f1 = inline('9.*(x-1).^8');
ascisse = [0; 0.25; 0.5; 0.75; 1];
fi = f(ascisse);
fiHermite = zeros(2*length(ascisse), 1);
fiHermite(1:2:length(fiHermite)-1) = fi;
fiHermite(2:2:length(fiHermite)) = f1(ascisse);

pn = formaNewton(ascisse, fi);
ph = hermite(ascisse, fiHermite);

xtest = linspace(0,1,101);
figure (1)
h = subplot(1,2,1);
plot(xtest, f(xtest), 'b', xtest, pn(xtest), 'r', ascisse, fi, 'black *
    ');
grid on
legend('funzione', 'polinomio', 'ascisse')
title('Interpolazione con polinomio di Newton')
subplot(1,2,2);
plot(xtest, f(xtest), 'b', xtest, ph(xtest), 'r', ascisse, fi, 'black *
    ');
grid on
axis ([get(h, 'XLim') get(h, 'YLim')])
legend('funzione', 'polinomio', 'ascisse')
title('Interpolazione con polinomio di Hermite')

figure (2)
plot(xtest, pn(xtest), 'r *', xtest, ph(xtest), 'g *', xtest, f(xtest),
     'b', ascisse, fi, 'black O');
grid on
legend('newton', 'hermite', 'funzione', 'ascisse')
title('Forma di Newton e Polinomio di Hermite')
```

Codice 4.15: Esercizio 4.10.

```
% Esercizio 4.10
```

```
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:12 CET

e= inline ('((2* pi ).^( n +1))./( factorial (n +1)) ');
n=0;
while (e(n)>= 10^ -6)
n=n+1;
end
fprintf ('e(%d) = %6.3 e\n', n, e(n));
```

Codice 4.16: Esercizio 4.11.

```
% Esercizio 4.11
%
% Autore: Tommaso Papini,
% Ultima modifica: 31 Ottobre 2012, 16:57 CET

fRunge = inline('1./(1.+x.^2)');
fBernstein = inline('abs(x)');
e = inline('abs( feval(f,x) - feval(p,x) )');

disp('Esempio di Runge:')
a=-5; b=5;
n = 5;
ascisse = ascisseEquidistanti(a, b, n);
pRunge = formaNewton(ascisse, fRunge(ascisse));
figure(1)
fplot(@(x)(e(fRunge, pRunge, x)), [a b], 'b');
hold on
grid on
legend('n=5')
title('Esempio di Runge: Errori')
maxErr = e(fRunge, pRunge, fminbnd(@(x)(-e(fRunge, pRunge, x)), a, b));
str = sprintf('Errore massimo con n = %d: %5.4f', n, maxErr); disp(str)
    ;
figure (2)
subplot(2,2,1)
fplot(fRunge, [a b], 'b')
hold on
grid on
fplot(pRunge, [a b], 'r')
plot(ascisse, fRunge(ascisse), 'k *')
title(sprintf('n=%d', n))

n = 10;
ascisse = ascisseEquidistanti(a, b, n);
pRunge = formaNewton(ascisse, fRunge(ascisse));
figure(1)
fplot(@(x)(e(fRunge, pRunge, x)), [a b], 'r');
```

```
legend('n=5', 'n=10')
maxErr = e(fRunge, pRunge, fminbnd(@(x)(-e(fRunge, pRunge, x)), a, b));
str = sprintf('Errore massimo con n = %d: %5.4f', n, maxErr); disp(str)
    ;
figure (2)
subplot(2,2,2)
fplot(fRunge, [a b], 'b')
hold on
grid on
fplot(pRunge, [a b], 'r')
plot(ascisse, fRunge(ascisse), 'k *')
title(sprintf('n=%d', n))

n = 15;
ascisse = ascisseEquidistanti(a, b, n);
pRunge = formaNewton(ascisse, fRunge(ascisse));
figure(1)
fplot(@(x)(e(fRunge, pRunge, x)), [a b], 'g');
legend('n=5', 'n=10', 'n=15')
maxErr = e(fRunge, pRunge, fminbnd(@(x)(-e(fRunge, pRunge, x)), a, b));
str = sprintf('Errore massimo con n = %d: %5.4f', n, maxErr); disp(str)
    ;
figure (2)
subplot(2,2,3)
fplot(fRunge, [a b], 'b')
hold on
grid on
fplot(pRunge, [a b], 'r')
plot(ascisse, fRunge(ascisse), 'k *')
title(sprintf('n=%d', n))

n = 20;
ascisse = ascisseEquidistanti(a, b, n);
pRunge = formaNewton(ascisse, fRunge(ascisse));
maxErr = e(fRunge, pRunge, fminbnd(@(x)(-e(fRunge, pRunge, x)), a, b));
str = sprintf('Errore massimo con n = %d: %5.4f', n, maxErr); disp(str)
    ;
figure (2)
subplot(2,2,4)
fplot(fRunge, [a b], 'b')
hold on
grid on
fplot(pRunge, [a b], 'r')
plot(ascisse, fRunge(ascisse), 'k *')
title(sprintf('n=%d', n))

disp(' '), disp('Esempio di Bernstein:')
a=-1; b=1;
n = 5;
ascisse = ascisseEquidistanti(a, b, n);
pBernstein = formaNewton(ascisse, fBernstein(ascisse));
figure(3)
fplot(@(x)(e(fBernstein, pBernstein, x)), [a b], 'b');
hold on
grid on
```

```
legend('n=5')
title('Esempio di Bernstein: Errori')
maxErr = e(fBernstein, pBernstein, fminbnd(@(x)(-e(fBernstein,
    pBernstein, x)), a, b));
str = sprintf('Errore massimo con n = %d: %5.4f', n, maxErr); disp(str)
    ;
figure (4)
subplot(2,2,1)
fplot(fBernstein, [a b], 'b')
hold on
grid on
fplot(pBernstein, [a b], 'r')
plot(ascisse, fBernstein(ascisse), 'k *')
title(sprintf('n=%d', n))

n = 10;
ascisse = ascisseEquidistanti(a, b, n);
pBernstein = formaNewton(ascisse, fBernstein(ascisse));
figure(3)
fplot(@(x)(e(fBernstein, pBernstein, x)), [a b], 'r');
legend('n=5', 'n=10')
maxErr = e(fBernstein, pBernstein, fminbnd(@(x)(-e(fBernstein,
    pBernstein, x)), a, b));
str = sprintf('Errore massimo con n = %d: %5.4f', n, maxErr); disp(str)
    ;
figure (4)
subplot(2,2,2)
fplot(fBernstein, [a b], 'b')
hold on
grid on
fplot(pBernstein, [a b], 'r')
plot(ascisse, fBernstein(ascisse), 'k *')
title(sprintf('n=%d', n))

n = 15;
ascisse = ascisseEquidistanti(a, b, n);
pBernstein = formaNewton(ascisse, fBernstein(ascisse));
figure(3)
fplot(@(x)(e(fBernstein, pBernstein, x)), [a b], 'g');
legend('n=5', 'n=10', 'n=15')
maxErr = e(fBernstein, pBernstein, fminbnd(@(x)(-e(fBernstein,
    pBernstein, x)), a, b));
str = sprintf('Errore massimo con n = %d: %5.4f', n, maxErr); disp(str)
    ;
figure (4)
subplot(2,2,3)
fplot(fBernstein, [a b], 'b')
hold on
grid on
fplot(pBernstein, [a b], 'r')
plot(ascisse, fBernstein(ascisse), 'k *')
title(sprintf('n=%d', n))

n = 20;
ascisse = ascisseEquidistanti(a, b, n);
```

```
pBernstein = formaNewton(ascisse, fBernstein(ascisse));
maxErr = e(fBernstein, pBernstein, fminbnd(@(x)(-e(fBernstein,
    pBernstein, x)), a, b));
str = sprintf('Errore massimo con n = %d: %5.4f', n, maxErr); disp(str)
    ;
figure (4)
subplot(2,2,4)
fplot(fBernstein, [a b], 'b')
hold on
grid on
fplot(pBernstein, [a b], 'r')
plot(ascisse, fBernstein(ascisse), 'k *')
title(sprintf('n=%d', n))
```

Codice 4.17: Esercizio 4.15.

```
% Esercizio 4.15
%
% Autore: Tommaso Papini,
% Ultima modifica: 31 Ottobre 2012, 18:26 CET

fRunge = inline('1./(1.+x.^2)');
fBernstein = inline('abs(x)');
e = inline('abs( feval(f,x) - feval(p,x) )');

disp('Esempio di Runge:')
a=-5; b=5;
n = 5;
ascisse = ascisseChebyshev(a, b, n);
pRunge = formaNewton(ascisse, fRunge(ascisse));
figure(1)
fplot(@(x)(e(fRunge, pRunge, x)), [a b], 'b');
hold on
grid on
legend('n=5')
title('Esempio di Runge: Errori')
maxErr = e(fRunge, pRunge, fminbnd(@(x)(-e(fRunge, pRunge, x)), a, b));
str = sprintf('Errore massimo con n = %d: %5.4f', n, maxErr); disp(str)
    ;
figure (2)
subplot(2,2,1)
fplot(fRunge, [a b], 'b')
hold on
grid on
fplot(pRunge, [a b], 'r')
plot(ascisse, fRunge(ascisse), 'k *')
title(sprintf('n=%d', n))

n = 10;
ascisse = ascisseChebyshev(a, b, n);
```

```
pRunge = formaNewton(ascisse, fRunge(ascisse));
figure(1)
fplot(@(x)(e(fRunge, pRunge, x)), [a b], 'r');
legend('n=5', 'n=10')
maxErr = e(fRunge, pRunge, fminbnd(@(x)(-e(fRunge, pRunge, x)), a, b));
str = sprintf('Errore massimo con n = %d: %5.4f', n, maxErr); disp(str)
    ;
figure (2)
subplot(2,2,2)
fplot(fRunge, [a b], 'b')
hold on
grid on
fplot(pRunge, [a b], 'r')
plot(ascisse, fRunge(ascisse), 'k *')
title(sprintf('n=%d', n))

n = 15;
ascisse = ascisseChebyshev(a, b, n);
pRunge = formaNewton(ascisse, fRunge(ascisse));
figure(1)
fplot(@(x)(e(fRunge, pRunge, x)), [a b], 'g');
legend('n=5', 'n=10', 'n=15')
maxErr = e(fRunge, pRunge, fminbnd(@(x)(-e(fRunge, pRunge, x)), a, b));
str = sprintf('Errore massimo con n = %d: %5.4f', n, maxErr); disp(str)
    ;
figure (2)
subplot(2,2,3)
fplot(fRunge, [a b], 'b')
hold on
grid on
fplot(pRunge, [a b], 'r')
plot(ascisse, fRunge(ascisse), 'k *')
title(sprintf('n=%d', n))

n = 20;
ascisse = ascisseChebyshev(a, b, n);
pRunge = formaNewton(ascisse, fRunge(ascisse));
figure(1)
fplot(@(x)(e(fRunge, pRunge, x)), [a b], 'y');
legend('n=5', 'n=10', 'n=15', 'n=20')
maxErr = e(fRunge, pRunge, fminbnd(@(x)(-e(fRunge, pRunge, x)), a, b));
str = sprintf('Errore massimo con n = %d: %5.4f', n, maxErr); disp(str)
    ;
figure (2)
subplot(2,2,4)
fplot(fRunge, [a b], 'b')
hold on
grid on
fplot(pRunge, [a b], 'r')
plot(ascisse, fRunge(ascisse), 'k *')
title(sprintf('n=%d', n))

disp(' '), disp('Esempio di Bernstein:')
a=-1; b=1;
n = 5;
```

```
ascisse = ascisseChebyshev(a, b, n);
pBernstein = formaNewton(ascisse, fBernstein(ascisse));
figure(3)
fplot(@(x)(e(fBernstein, pBernstein, x)), [a b], 'b');
hold on
grid on
legend('n=5')
title('Esempio di Bernstein: Errori')
maxErr = e(fBernstein, pBernstein, fminbnd(@(x)(-e(fBernstein,
    pBernstein, x)), a, b));
str = sprintf('Errore massimo con n = %d: %5.4f', n, maxErr); disp(str)
    ;
figure (4)
subplot(2,2,1)
fplot(fBernstein, [a b], 'b')
hold on
grid on
fplot(pBernstein, [a b], 'r')
plot(ascisse, fBernstein(ascisse), 'k *')
title(sprintf('n=%d', n))

n = 10;
ascisse = ascisseChebyshev(a, b, n);
pBernstein = formaNewton(ascisse, fBernstein(ascisse));
figure(3)
fplot(@(x)(e(fBernstein, pBernstein, x)), [a b], 'r');
legend('n=5', 'n=10')
maxErr = e(fBernstein, pBernstein, fminbnd(@(x)(-e(fBernstein,
    pBernstein, x)), a, b));
str = sprintf('Errore massimo con n = %d: %5.4f', n, maxErr); disp(str)
    ;
figure (4)
subplot(2,2,2)
fplot(fBernstein, [a b], 'b')
hold on
grid on
fplot(pBernstein, [a b], 'r')
plot(ascisse, fBernstein(ascisse), 'k *')
title(sprintf('n=%d', n))

n = 15;
ascisse = ascisseChebyshev(a, b, n);
pBernstein = formaNewton(ascisse, fBernstein(ascisse));
figure(3)
fplot(@(x)(e(fBernstein, pBernstein, x)), [a b], 'g');
legend('n=5', 'n=10', 'n=15')
maxErr = e(fBernstein, pBernstein, fminbnd(@(x)(-e(fBernstein,
    pBernstein, x)), a, b));
str = sprintf('Errore massimo con n = %d: %5.4f', n, maxErr); disp(str)
    ;
figure (4)
subplot(2,2,3)
fplot(fBernstein, [a b], 'b')
hold on
grid on
```

```
fplot(pBernstein, [a b], 'r')
plot(ascisse, fBernstein(ascisse), 'k *')
title(sprintf('n=%d', n))

n = 20;
ascisse = ascisseChebyshev(a, b, n);
pBernstein = formaNewton(ascisse, fBernstein(ascisse));
figure(3)
fplot(@(x)(e(fBernstein, pBernstein, x)), [a b], 'y');
legend('n=5', 'n=10', 'n=15', 'n=20')
maxErr = e(fBernstein, pBernstein, fminbnd(@(x)(-e(fBernstein,
    pBernstein, x)), a, b));
str = sprintf('Errore massimo con n = %d: %5.4f', n, maxErr); disp(str)
    ;
figure (4)
subplot(2,2,4)
fplot(fBernstein, [a b], 'b')
hold on
grid on
fplot(pBernstein, [a b], 'r')
plot(ascisse, fBernstein(ascisse), 'k *')
title(sprintf('n=%d', n))
```

Codice 4.18: Esercizio 4.20.

```
% Esercizio 4.20
%
% Autore: Tommaso Papini,
% Ultima modifica: 1 Novembre 2012, 11:20 CET

fRunge = inline('1./(1.+x.^2)');
fBernstein = inline('abs(x)');

for i=1:4
    n=i*10;

    a=-5; b=5;
    ascisse = ascisseEquidistanti(a, b, n);
    fi = fRunge(ascisse);
    sN = splineCubica(ascisse, fi, false);
    sNaK = splineCubica(ascisse, fi, true);
    figure (1)
    h = subplot(2,2,i);
    fplot(fRunge, [a b], 'b')
    lims = [get(h, 'XLim') get(h, 'YLim')];
    hold on
    grid on
    for j=1:n
        fplot(inline(sN(j)), [ascisse(j) ascisse(j+1)], 'r')
    end
```

```
    axis(lims)
    plot(ascisse, fi, 'k *')
    title(sprintf('n=%d', n))
    figure (2)
    h = subplot(2,2,i);
    fplot(fRunge, [a b], 'b')
    lims = [get(h, 'XLim') get(h, 'YLim')];
    hold on
    grid on
    for j=1:n
        fplot(inline(sNaK(j)), [ascisse(j) ascisse(j+1)], 'r')
    end
    axis(lims)
    plot(ascisse(1), fi(1), 'k *', ascisse(n+1), fi(n+1), 'k *')
    plot(ascisse(2), fi(2), 'k O', ascisse(n), fi(n), 'k O')
    plot(ascisse(3:n-1), fi(3:n-1), 'k *')
    title(sprintf('n=%d', n))

    a=-1; b=1;
    ascisse = ascisseEquidistanti(a, b, n);
    fi = fBernstein(ascisse);
    sN = splineCubica(ascisse, fi, false);
    sNaK = splineCubica(ascisse, fi, true);
    figure (3)
    h = subplot(2,2,i);
    fplot(fBernstein, [a b], 'b')
    lims = [get(h, 'XLim') get(h, 'YLim')];
    hold on
    grid on
    for j=1:n
        fplot(inline(sN(j)), [ascisse(j) ascisse(j+1)], 'r')
    end
    axis(lims)
    plot(ascisse, fi, 'k *')
    title(sprintf('n=%d', n))
    figure (4)
    h = subplot(2,2,i);
    fplot(fBernstein, [a b], 'b')
    lims = [get(h, 'XLim') get(h, 'YLim')];
    hold on
    grid on
    for j=1:n
        fplot(inline(sNaK(j)), [ascisse(j) ascisse(j+1)], 'r')
    end
    axis(lims)
    plot(ascisse(1), fi(1), 'k *', ascisse(n+1), fi(n+1), 'k *')
    plot(ascisse(2), fi(2), 'k O', ascisse(n), fi(n), 'k O')
    plot(ascisse(3:n-1), fi(3:n-1), 'k *')
    title(sprintf('n=%d', n))
end
```

Codice 4.19: Esercizio 4.22.

```
% Esercizio 4.22
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:12 CET

hold on
y = [5.22; 4.00; 4.28; 3.89; 3.53; 3.12; 2.73; 2.70; 2.20; 2.08; 1.94];
plot ((0:10) ,y, 'black *')
y = log(y);

A = [ ones(1 ,11)' , (0:10)'];
[y, r] = SistemaQR ( FattQR (A),y);
y (1)= exp(y (1));
y(2)= -y (2);
disp ('Soluzione :'), disp (y)
disp ('Residuo :'), disp (r)

plot ((0:10) , y (1)* exp(-y (2)*(0:10)))

legend ('punti ','polinomio ','Location ','Best ')
```

Capitolo 5

Formule di quadratura

Indice

In questo capitolo analizziamo il problema del calcolo di un **integrale definito**,

$$I(f) = \int_a^b f(x)dx, \qquad a < b, \tag{5.1}$$

con $[a, b]$ intervallo *limitato* ed $f(x)$ funzione abbastanza *regolare* (cioè che non presenta singolarità, ovvero punti di discontinuità, nell'intervallo). Quindi nel seguito assumeremo che $f(x)$ sia *continua* in $[a, b]$.
Tutti i metodi descritti si basano sull'approssimazione della funzione da integrare tramite una funzione polinomiale o una funzione polinomiale a tratti.
Studiamo innanzitutto il condizionamento del problema (5.1), ovvero il problema in cui si calcola l'integrale esatto, dove si ha la perturbazione sulla funzione integranda $f(x)$, indicando con $\tilde{f}(x)$ la funzione perturbata. Ricordando allora la definizione di norma (4.18), si ha:

$$|I(f) - I(\tilde{f})| = |\int_a^b f(x)dx - \int_a^b \tilde{f}(x)dx| = |\int_a^b (f(x) - \tilde{f}(x))dx| \leq$$
$$\leq \int_a^b |f(x) - \tilde{f}(x)|dx \overset{1}{\leq} ||f - \tilde{f}|| \int_a^b dx \overset{2}{=} [x]_a^b ||f - \tilde{f}|| =$$
$$= (b-a)||f - \tilde{f}||.$$

Considerando che $|I(f) - I(\tilde{f})|$ rappresenta l'errore commesso sulla soluzione del problema e $||f - \tilde{f}||$ rappresenta una maggiorazione dell'errore commesso sui dati iniziali, si ottiene che la quantità

$$k = b - a \tag{5.2}$$

definisce il *numero di condizionamento* del problema (5.1).

5.1 Formule di Newton-Cotes

Supponiamo di approssimare la funzione integranda $f(x)$ con un polinomio interpolante su $n+1$ ascisse equidistanti (ovvero uniformemente distribuite sull'intervallo $[a, b]$), ovvero un polinomio tale che

$$p(x_i) = f(x) \equiv f_i, \qquad i = 0, 1, \ldots, n,$$

con

$$x_i = a + ih, \qquad h = \frac{b-a}{n}. \tag{5.3}$$

Considerando adesso la forma di Lagrange (vedi (4.5) e (4.7) del polinomio interpolante:

[1] Per la disuguaglianza triangolare, in quanto l'integrale rappresenta una somma, quindi il valore assoluto dell'integrale è minore o uguale dell'integrale del valore assoluto (esattamente come una sommatoria).

[2] Per definizione di norma (vedi (4.18)), che risulta essere uno scalare (e quindi estraibile dall'integrale).

$$
\begin{aligned}
I_n(f) \approx \int_a^b p(x)dx = \int_a^b \sum_{k=0}^{n} f_k L_{k,n}(x)dx = \\
= \sum_{k=0}^{n} f_k \int_a^b L_{k,n}(x)dx = \sum_{k=0}^{n} f_k \int_a^b \prod_{\substack{j=0 \\ j\neq k}}^{n} \frac{x-x_j}{x_k-x_j}dx = \\
= \sum_{k=0}^{n} f_k \int_0^n \prod_{\substack{j=0 \\ j\neq k}}^{n} \frac{t-j}{k-j} h \cdot dt \overset{1}{=} h \sum_{k=0}^{n} f_k \int_0^n \prod_{\substack{j=0 \\ j\neq k}}^{n} \frac{t-j}{k-j} dt \\
= \frac{b-a}{n} \sum_{k=0}^{n} c_{k,n} f_k, \qquad (5.4)
\end{aligned}
$$

dove

$$
c_{k,n} = \int_0^n \prod_{\substack{j=0 \\ j\neq k}}^{n} \frac{t-j}{k-j} dt, \qquad k = 0, \ldots, n, \qquad (5.5)
$$

con $I_n(f)$ che rappresenta l'integrale definito tra a e b di $f(x)$ utilizzando un'approssimazione di f tramite un polinomio interpolante su n ascisse equidistanti. La (5.4) definisce quindi la generica **formula di quadratura di Newton-Cotes**.
Si osserva che avendo eseguito il cambio di variabile in t, i coefficienti $c_{k,n}$ non dipendono più dall'intervallo $[a, b]$, ma soltanto dal numero n (ovvero il numero di sottointervalli), in quanto quello che viene preso in considerazione sono le distanze reciproche tra le ascisse che, essendo equidistanti, risulteranno (per uno stesso n) sempre alla stessa distanza relativa.
A seconda di quante ascisse vengono utilizzate per l'approssimazione della funzione si hanno differenti istanze della formula di Newton-Cotes generica. In particolare:

- **formula dei trapezi:** $n = 1$
  In questo caso abbiamo che $c_{0,1} = c_{1,1} = 1/2$, infatti

$$
\begin{aligned}
c_{0,1} &= \int_0^1 \frac{t-1}{0-1} dt = \int_0^1 dt - \int_0^1 t\, dt = 1 - \frac{1}{2} = \frac{1}{2}, \\
c_{1,1} &= \int_0^1 \frac{t-0}{1-0} dt = \int_0^1 t\, dt = \frac{1}{2},
\end{aligned}
$$

  e la formula dei trapezi risulta essere

$$
I_1(f) = \frac{b-a}{2}(f(a) + f(b)). \qquad (5.6)
$$

[1] È stato effettuato il cambio di variabile $x = a + ht$, che, derivando, implica $dx = h \cdot dt$ ed il cambio di indici (infatti per avere $x \in [a, b]$ si ha $t \in [0, n]$).

Il nome della formula deriva dal fatto che l'area calcolata con questo metodo coincide con l'area del trapezio avente come vertici $(a, 0)$, $(a, f(a))$, $(b, f(b))$ e $(b, 0)$.

- **formula di Simpson:** $n = 2$
  Si può facilmente verificare che in questo caso i coefficienti risultano valere

$$c_{0,2} = c_{2,2} = 1/2, \quad c_{1,2} = 4/3,$$

e la formula è definita da

$$I_2(f) = \frac{b-a}{6}(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)). \tag{5.7}$$

In generale si osserva che per $n \leq 6$ si hanno tutti i $c_{k,n} \geq 0$. Al contrario per $n \geq 7$ compaiono dei coefficienti negativi, che influiranno sul condizionamento del problema.

Teorema 5.1. *Riguardo ai coefficienti (5.5) vale il seguente risultato:*

$$\frac{1}{n}\sum_{k=0}^{n} c_{k,n} = 1.$$

Possiamo adesso studiare il condizionamento del problema del calcolo della formula (5.4):

$$|I_n(f) - I_n(\tilde{f})| = \frac{b-a}{n}|\sum_{k=0}^{n}(f_k - \tilde{f}_k)c_{k,n}| \leq \frac{b-a}{n}\sum_{k=0}^{n}|f_k - \tilde{f}_k| \cdot |c_{k,n}| \leq {}^{1}$$
$$\leq \left(\frac{b-a}{n}\sum_{k=0}^{n}|c_{k,n}|\right)||f - \tilde{f}||,{}^{2}$$

Quindi si conclude che

$$k_n = \frac{b-a}{n}\sum_{k=0}^{n}|c_{k,n}|$$

è il *numero di condizionamento* delle formule di Newton-Cotes.
Allora, per quanto osservato prima e per il Teorema 5.1 risulta che:

$$k_n \equiv k, \quad n = 1, \ldots, 6,$$
$$k_n > k, \quad n \geq 7,$$

ovvero le formule di Newton-Cotes sono convenientemente utilizzabili fino ad $n = 6$ (ovvero 7 ascisse di interpolazione).

[1] Per la disuguaglianza triangolare.
[2] Per la definizione di norma in $C^{(0)}$ (vedi (4.18)).

5.2 Errore e formule composite

Studiamo adesso l'*errore di quadratura* commesso nell'utilizzare un'approssimazione polinomiale della funzione integranda al posto della funzione esatta. Ricordando la stima fatta sull'errore di interpolazione nelle (4.16) e (4.17) abbiamo:

$$E_n(f) = I(f) - I_n(f) = \int_a^b f(x)\,dx - \int_a^b p_n(x)\,dx = \int_a^b (f(x) - p_n(x))\,dx$$
$$= \int_a^b e(x)\,dx = \int_a^b f[x_0, \ldots, x_n, x] w_{n+1}(x)\,dx. \tag{5.8}$$

Teorema 5.2. *Se* $f(x) \in C^{(n+k)}$, *con*

$$k = \begin{cases} 1, & \text{se } n \text{ è dispari,} \\ 2, & \text{se } n \text{ è pari,} \end{cases}$$

allora l'errore di quadratura (5.8) è dato da

$$E_n(f) = \nu_n \frac{f^{(n+k)}(\xi)}{(x+k)!} \left(\frac{b-a}{n}\right)^{n+k+1},$$

per un opportuno $\xi \in [a, b]$, *dove*

$$\nu_n = \begin{cases} \int_0^n \prod_{j=0}^n (t-j)\,dt, & \text{se } n \text{ è dispari,} \\ \int_0^n t \prod_{j=0}^n (t-j)\,dt, & \text{se } n \text{ è pari.} \end{cases}$$

Corollario 5.1. *Le formule definite su* $n+1$ *punti sono esatte per polinomi fino a grado* n, *se* n *è dispari, e fino a grado* $n+1$ *se* n *è pari.*

Quindi per il metodo dei trapezi si ha:

$$E_1(f) = -\frac{1}{12} f^{(2)}(\xi)(b-a)^3, \qquad \xi \in [a, b], \tag{5.9}$$

mentre per il metodo di Simpson risulta:

$$E_2(f) = -\frac{1}{90} f^{(4)}(\xi) \left(\frac{b-a}{2}\right)^5, \qquad \xi \in [a, b]. \tag{5.10}$$

Per poter decrementare il rapporto $\frac{b-a}{n}$ senza aumentare n (per quanto detto riguardo al condizionamento del problema) si può utilizzare un approccio del tutto analogo a quello fatto per le *spline* nel Capitolo 4: l'intervallo $[a, b]$ viene suddiviso in n sottointervalli di uguale ampiezza e su ognuno di essi viene applicata la stessa formula di Newton-Cotes. Si ottiene così una **formula di Newton-Cotes composita**.
Ad esempio la *formula dei trapezi composita* è data dall'applicazione della formula

dei trapezi su ciascun sottointervallo $[x_{i-1}, x_i]$, per $i = 1, \ldots, n$, sulla partizione uniforme (5.3):

$$I_1^{(n)}(f) \equiv \frac{b-a}{n} \cdot \frac{1}{2} \left((f_0 + f_1) + (f_1 + f_2) + \cdots + (f_{n-1} + f_n) \right) = \\ = \frac{b-a}{2n} \left(f_0 + 2 \sum_{i=1}^{n-1} f_i + f_n \right), \tag{5.11}$$

ed il corrispondente errore di quadratura si vede valere (generalizzando il Teorema (5.2)):

$$E_1^{(n)}(f) = -\frac{n}{12} f^{(2)}(\xi) \left(\frac{b-a}{n} \right)^3, \qquad \xi \in [a, b]. \tag{5.12}$$

Il Codice di seguito mostra un'implementazione in MATLAB della formula dei trapezi composita:

Codice 5.1: Formula dei trapezi composita.

```
% int = trapeziComposita(f, a, b, n)
% Formula dei trapezi composita per l'approssimazione dell'integrale
% definito di una funzione.
%
% Input:
%   -f: la funzione di cui si vuol calcolare l'integrale;
%   -a: estremo sinistro dell'intervallo di integrazione;
%   -b: estremo destro dell'intervallo di integrazione;
%   -n: numero di sottointervalli sui quali applicare la formula dei
%   trapezi semplice.
% Output:
%   -int: l'approssimazione dell'integrale definito della funzione.
%
% Autore: Tommaso Papini,
% Ultima modifica: 24 Ottobre 2012, 17:57 CEST

function [int] = trapeziComposita(f, a, b, n)
    h = (b-a)/n;
    int = 0;
    for i=1:n-1
        int = int+f(a+i*h);
    end
    int = (h/2)*(2*int + f(a) + f(b));
end
```

Per la *formula di Simpson composita* si ha invece (per valori pari di n):

$$I_2^{(n)} \equiv \frac{b-a}{3n} \left(f_0 + 4f_1 + f_2 + f_2 + \cdots + f_{n-2} + 4f_{n-1} + f_n \right) = \\ = \frac{b-a}{3n} \left(4 \sum_{i=1}^{n/2} f_{2i-1} + 2 \sum_{i=0}^{n/2} -f_0 - f_n \right), \tag{5.13}$$

dove la prima sommatoria è per gli indici dispari mentre la seconda è per i pari (f_0 ed f_n vengono sottratti in quanto sono contati due volte nella seconda sommatoria). Il corrispettivo errore di quadratura si vede valere:

$$E_2^{(n)}(f) = -\frac{n}{90} f^{(4)}(\xi) \left(\frac{b-a}{n}\right)^5, \qquad \xi \in [a,b]. \tag{5.14}$$

Codice 5.2: Formula di Simpson composita.

```
% int = simpsonComposita(f, a, b, n)
% Formula di Simpson composita per l'approssimazione dell'integrale
% definito di una funzione.
%
% Input:
%   -f: la funzione di cui si vuol calcolare l'integrale;
%   -a: estremo sinistro dell'intervallo di integrazione;
%   -b: estremo destro dell'intervallo di integrazione;
%   -n: numero, pari, di sottointervalli sui quali applicare la formula
%   di Simpson semplice.
% Output:
%   -int: l'approssimazione dell'integrale definito della funzione.
%
% Autore: Tommaso Papini,
% Ultima modifica: 24 Ottobre 2012, 18:20 CEST

function [int] = simpsonComposita(f, a, b, n)
    h = (b-a)/n;
    int = f(a)-f(b);
    for i=1:n/2
        int = int + 4*f(a+(2*i-1)*h)+2*f((a+2*i*h));
    end
    int = int*(h/3);
end
```

Per le due stime degli errori di quadratura sulle formule composite si vede facilmente che

$$E_k^{(n)}(f) \to 0, \qquad n \to \infty, \qquad k = 1, 2.$$

5.3 Formule adattative

Supponiamo adesso di voler approssimare l'integrale definito di una funzione che, all'interno dell'intervallo $[a, b]$ di integrazione, ha un carattere molto variabile, ovvero ci sono punti in cui la funzione effettua rapide variazioni (per valori alti della derivata in valore assoluto) a causa di un'elevata pendenza, mentre in altri dove la funzione resta pressoché la stessa (ovvero ha la derivata prima circa zero). In questi casi le formule composite di Newton-Cotes possono essere modificate in **formule adattative**, ovvero che sono in grado di scegliere i nodi di interpolazione a seconda dell'andamento della funzione.
Ad esempio, implementando la formula dei trapezi e dei trapezi composita su due

sottointervalli (che come nodi hanno f_0 ed f_1 e f_1 e f_2), si hanno i seguenti errori (vedi (5.9) e (5.12)):

$$I(f) - I_1(f) \approx -\frac{1}{12} f^{(2)}(\xi)(b-a)^3, \quad I(f) - I_1^{(2)}(f) \approx -\frac{n}{12} f^{(2)}(\xi) \left(\frac{b-a}{n}\right)^3$$

Quindi possiamo stimare l'errore dell'utilizzo della formula dei trapezi composita su due sottointervalli come segue:

$$\begin{aligned} I(f) - I_1^{(2)}(f) &\approx \frac{1}{4}\left(I(f) - I_1(f)\right), \\ \frac{3}{4} I(f) - I_1^{(2)} &\approx -\frac{1}{4} I_1(f) \\ I(f) - I_1^{(2)}(f) &\approx \frac{1}{3}\left(I_1^{(2)}(f) - I_1(f)\right), \end{aligned}$$

in modo tale che se l'errore stimato risulta maggiore di una tolleranza fissata tol, l'intera procedura viene reiterata su ognuno dei due sottointervalli $[a, \frac{a+b}{2}]$ e $[\frac{a+b}{2}, b]$ con tolleranza $^{tol}/_{/2}$.
Il Codice 5.3 mostra un'implementazione in MATLAB della formula dei trapezi adattativa.

Codice 5.3: Formula dei trapezi adattativa.

```
% [int, pt] = trapeziAdattativaRicorsiva(f, a, b, tol, pt)
% Formula dei trapezi adattativa per l'approssimazione dell'integrale
% definito di una funzione.
%
% Input:
%   -f: la funzione di cui si vuol calcolare l'integrale;
%   -a: estremo sinistro dell'intervallo di integrazione;
%   -b: estremo destro dell'intervallo di integrazione;
%   -tol: la tolleranza entro la quale si richiede debba rientrare la
%   soluzione approssimata.
% Output:
%   -int: l'approssimazione dell'integrale definito della funzione;
%   -pt: numero di punti generati ricorsivamente.
%
% Autore: Tommaso Papini,
% Ultima modifica: 25 Giugno 2013, 23:54 CEST

function [int, pt] = trapeziAdattativa(f, a, b, tol)
    [int, pt] = trapeziAdattativaRicorsiva(f, a, b, tol, 3);
end

function [int, pt] = trapeziAdattativaRicorsiva(f, a, b, tol, pt)
    h = (b-a)/2;
    m = (a+b)/2;
    int1 = h*(feval(f, a) + feval(f, b));
    int = int1/2 + h*feval(f, m);
    err = abs(int-int1)/3;
    if err>tol
```

```
        [intSx, ptSx] = trapeziAdattativaRicorsiva(f, a, m, tol/2, 1);
        [intDx, ptDx] = trapeziAdattativaRicorsiva(f, m, b, tol/2, 1);
        int = intSx+intDx;
        pt = pt+ptSx+ptDx;
    end
end
```

In modo del tutto analogo si implementa la formula di Simpson e la formula di Simpson composita (con $n = 4$, in quanto per ogni sottointervallo c'è bisogno di un punto intermedio aggiuntivo), che presentano i seguenti errori (vedi (5.10) e (5.14)):

$$I(f) - I_2(f) \approx -\frac{1}{90} f^{(4)}(\xi) \frac{(b-a)^5}{32},$$
$$I(f) - I_2^{(4)}(f) \approx -\frac{1}{90} f^{(4)}(\xi) \frac{(b-a)^5}{512},$$

per poter stimare l'errore commesso con l'utilizzo della formula di Simpson composita:

$$I(f) - I_2^{(4)}(f) \approx \frac{1}{15} \left(I_2^{(4)} - I_2(f) \right),$$

e poter quindi procedere come visto nel caso della formula dei trapezi se l'errore stimato risulta maggiore della tolleranza fissata *tol*.

Codice 5.4: Formula di Simpson adattativa.

```
% [int, pt] = simpsonAdattativa(f, a, b, tol)
% Formula di Simpson adattativa per l'approssimazione dell'integrale
% definito di una funzione.
%
% Input:
%   -f: la funzione di cui si vuol calcolare l'integrale;
%   -a: estremo sinistro dell'intervallo di integrazione;
%   -b: estremo destro dell'intervallo di integrazione;
%   -tol: la tolleranza entro la quale si richiede debba rientrare la
%   soluzione approssimata.
% Output:
%   -int: l'approssimazione dell'integrale definito della funzione;
%   -pt: numero di punti generati ricorsivamente.
%
% Autore: Tommaso Papini,
% Ultima modifica: 25 Giugno 2013, 23:53 CEST

function [int, pt] = simpsonAdattativa(f, a, b, tol)
    [int, pt] = simpsonAdattativaRicorsiva(f, a, b, tol, 5);
end

function [int, pt] = simpsonAdattativaRicorsiva(f, a, b, tol, pt)
    h = (b-a)/6;
    m = (a+b)/2;
    m1 = (a+m)/2;
    m2 = (m+b)/2;
```

```
    int1 = h*(feval(f, a) +4*feval(f, m) + feval(f, b));
28  int = int1/2 + h*(2*feval(f, m1) + 2*feval(f, m2) -feval(f, m));
    err = abs(int-int1)/15;
30  if err>tol
        [intSx, ptSx] = simpsonAdattativaRicorsiva(f, a, m, tol/2, 1);
32      [intDx, ptDx] = simpsonAdattativaRicorsiva(f, m, b, tol/2, 1);
        int = intSx+intDx;
34      pt = pt+ptSx+ptDx;
    end
36 end
```

Esercizi

Esercizio 5.1. *Calcolare il numero di condizionamento dell'integrale*

$$\int_0^{e^{21}} \sin\sqrt{x}\, dx.$$

Questo problema è ben condizionato o è malcondizionato?

Soluzione.
Dalla 5.2 risulta $\kappa = b - a = e^{21} - 0 = e^{21} > 10^9$, il problema è dunque mal condizionato.

Riferimenti MATLAB
Codice 5.5 (pagina 201)

Esercizio 5.2. *Derivare, dalla (5.5), i coefficienti della formula dei trapezi (5.6) e della formula di Simpson (5.7).*

Soluzione.

- Formula dei trapezi, $n = 1$:
  - $c_{1,1} = \int_0^1 \frac{t-0}{1-0}\, dt = \int_0^1 t = \frac{t^2}{2}\Big|_0^1 = \frac{1}{2}$,
  - $c_{0,1} = 1 - c_{1,1} = \frac{1}{2}$ per la (5.5).

  Segue $I_1(f) = (b-a)\left(\frac{1}{2}f(a) + \frac{1}{2}f(b)\right) = \frac{b-a}{2}\left(f(a) + f(b)\right)$.
- Formula di Simpson, $n = 2$:
  - $c_{1,2} = \int_0^2 \frac{t-0}{1-0}\frac{t-2}{1-2}\, dt = \int_2^0 t(t-2) = \left(\frac{t^3}{3} - t^2\right)\Big|_0^2 = \frac{4}{3}$,
  - $c_{2,2} = \int_0^2 \frac{t-0}{2-0}\frac{t-1}{2-1}\, dt = \int_2^0 \frac{t(t-1)}{2} = \left(\frac{t^3}{6} - \frac{t^2}{4}\right)\Big|_0^2 = \frac{1}{3}$,

- $c_{0,2} = 1 - c_{1,2} - c_{2,2} = \frac{1}{3}$ per la (5.5).

Segue $I_2(f) = \frac{b-a}{2}\left(\frac{1}{3}f(a) + \frac{4}{3}f\left(\frac{a+b}{2}\right) + \frac{1}{2}f(b)\right) =$
$= \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right)$.

Esercizio 5.3. *Verificare, utilizzando il risultato del Teorema (5.2) le (5.9) e (5.10).*

Soluzione.

- Formula dei trapezi, $n = 1$:
  - $k = 1$,
  - $\nu_1 = \int_0^1 \prod_{j=0}^1 (t-j)\,dt = \int_0^1 (t(t-1))\,dt = \left(\frac{t^3}{3} - \frac{t^2}{2}\right)\Big|_0^1 = -\frac{1}{6}$.

  Segue $E_1(f) = \nu_1 \frac{f^{(2)}(\xi)}{2!}\left(\frac{b-a}{1}\right)^3 = -\frac{1}{12}f^{(2)}(\xi)(b-a)^3$.
- Formula di Simpson, $n = 2$:
  - $k = 2$,
  - $\nu_2 = \int_0^2 t\prod_{j=0}^2 (t-j)\,dt = \int_0^1 (t^2(t-1)(t-2))\,dt =$
    $= \left(\frac{t^5}{5} - 3\frac{t^4}{4} + 2\frac{t^3}{3}\right)\Big|_0^1 = -\frac{4}{15}$.

  Segue $E_2(f) = \nu_2 \frac{f^{(4)}(\xi)}{4!}\left(\frac{b-a}{2}\right)^5 = -\frac{1}{90}f^{(4)}(\xi)\left(\frac{b-a}{2}\right)^5$.

Esercizio 5.4. *Scrivere una* `function` MATLAB *che implementi efficientemente la formula dei trapezi composita (5.11).*

Soluzione.
Si veda il Codice 5.1 a pagina 192 per l'implementazione in MATLAB.

Esercizio 5.5. *Scrivere una* `function` MATLAB *che implementi efficientemente la formula di Simpson composita (5.13).*

Soluzione.
Si veda il Codice 5.2 a pagina 193 per l'implementazione in MATLAB. Nella soluzione proposta per unificare le due sommatorie (che corrisponderebbero a due cicli `for`) si è portato fuori dalla seconda sommatoria il fattore f_0, quindi l'espressione effettivamente implementata è

$$I_2^{(n)}(f) = \frac{b-a}{3n}\left(\sum_{i=1}^{n/2}(4f_{2i-1} + 2f_{2i}) + f_0 - f_n\right).$$

Esercizio 5.6. *Implementare efficientemente in* MATLAB *la formula adattativa dei trapezi.*

Soluzione.
Per l'implementazione della formula adattativa dei trapezi si veda il Codice 5.3 a pagina 194. L'algoritmo è stato implementato in maniera ricorsiva, utilizzando due funzioni distinte: `trapeziAdattativa` che è la funzione di partenza e `trapeziAdattativaRicorsiva` che è la funzione che si occupa delle ricorsioni.

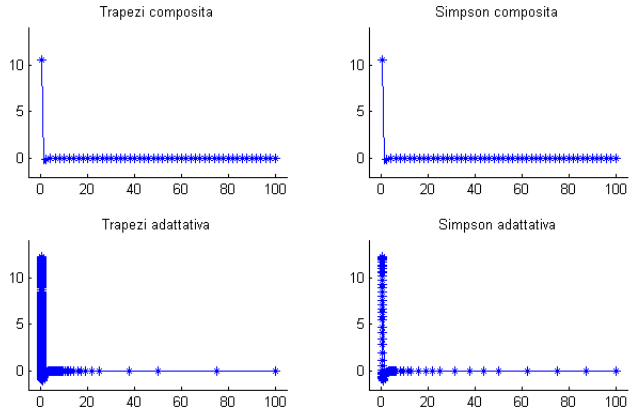
Esercizio 5.7. *Implementare efficientemente in* MATLAB *la formula adattativa di Simpson.*

Soluzione.
Per l'implementazione della formula adattativa di Simpson si veda il Codice 5.4 a pagina 195. L'algoritmo è stato implementato in maniera ricorsiva, utilizzando due funzioni distinte: `simpsonAdattativa` che è la funzione di partenza e `simpsonAdattativaRicorsiva` che è la funzione che si occupa delle ricorsioni.

Esercizio 5.8. *Come è classificabile, dal punto di vista del condizionamento, il seguente problema?*

$$\int_{\frac{1}{2}}^{100} -2x^{-3}\cos\left(x^{-2}\right)\,\mathrm{d}x \equiv \sin\left(10^{-4}\right) - \sin(4)$$

Soluzione.
Per la 5.2 risulta $\kappa = b - a = 100 - 1/2 = 99.5$, il problema è dunque mal condizionato. Visto che la funzione ha rapide variazioni in $(1, 10) \subset [1/2, 100]$ è preferibile utilizzare le formule di adattative invece che le formule composite; gli Esercizi 5.9 e 5.10 tabulano tali risultati.



Riferimenti MATLAB
Codice 5.6 (pagina 201)

Esercizio 5.9. *Utilizzare le* `function` *degli Esercizi 5.4 e 5.6 per il calcolo dell'integrale*

$$\int_{\frac{1}{2}}^{100} -2x^{-3}\cos\left(x^{-2}\right)\,\mathrm{d}x \equiv \sin\left(10^{-4}\right) - \sin(4),$$

indicando gli errori commessi. Si utilizzi $n = 1000, 2000, \ldots, 10000$ *per la formula dei trapezi composita e* $tol = 10^{-1}, 10^{-2}, \ldots, 10^{-5}$ *per la formula dei trapezi adattativa (indicando anche il numero di punti).*

Soluzione.

Formula composita dei trapezi		
n	I	$E_1^{(n)}$
1000	6.6401e-001	9.2897e-002
2000	7.3077e-001	2.6131e-002
3000	7.4507e-001	1.1836e-002
4000	7.5020e-001	6.7007e-003
5000	7.5260e-001	4.3009e-003
6000	7.5391e-001	2.9914e-003
7000	7.5470e-001	2.1998e-003
8000	7.5522e-001	1.6853e-003
9000	7.5557e-001	1.3321e-003
10000	7.5582e-001	1.0793e-003

Formula dei trapezi adattativa			
tol	I	$E_1^{(n)}$	punti
1.0e-001	7.5143e-001	5.4696e-003	159
1.0e-002	7.5563e-001	1.2676e-003	471
1.0e-003	7.5657e-001	3.3005e-004	1567
1.0e-004	7.5684e-001	6.5936e-005	4851

Riferimenti MATLAB
Codice 5.7 (pagina 202)

Esercizio 5.10. *Utilizzare le* `function` *degli Esercizi 5.5 e 5.7 per il calcolo dell'integrale*

$$\int_{\frac{1}{2}}^{100} -2x^{-3}\cos\left(x^{-2}\right)\,\mathrm{d}x \equiv \sin\left(10^{-4}\right) - \sin(4),$$

indicando gli errori commessi. Si utilizzi $n = 1000, 2000, \ldots, 10000$ *per la formula di Simpson composita e* $tol = 10^{-1}, 10^{-2}, \ldots, 10^{-5}$ *per la formula di Simpson adattativa (indicando anche il numero di punti).*

Soluzione.

Formula composita di Simpson		
n	I	$E_1^{(n)}$
1000	7.0132e-001	5.5580e-002
2000	7.5303e-001	3.8753e-003
3000	7.5617e-001	7.2977e-004
4000	7.5668e-001	2.2403e-004
5000	7.5681e-001	9.0209e-005
6000	7.5686e-001	4.3062e-005
7000	7.5688e-001	2.3094e-005
8000	7.5689e-001	1.3479e-005
9000	7.5689e-001	8.3892e-006
10000	7.5690e-001	5.4921e-006

Formula di Simpson adattativa			
tol	I	$E_1^{(n)}$	punti
1.0e-001	7.5701e-001	1.1164e-004	49
1.0e-002	7.5671e-001	1.9384e-004	65
1.0e-003	7.5690e-001	4.8068e-006	93
1.0e-004	7.5688e-001	1.7808e-005	181
1.0e-005	7.5690e-001	4.8337e-006	309

Riferimenti MATLAB
Codice 5.8 (pagina 202)

Codice degli esercizi

Codice 5.5: Esercizio 5.1.

```
% Esercizio 5.01
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:13 CET

format short
k = exp (1)^21
```

Codice 5.6: Esercizio 5.8.

```
% Esercizio 5.08
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:13 CET

f = inline (' -2.*x .^( -3).* cos(x .^( -2)) ');
subplot (2 ,2 ,1)
axis ([ -5 105 -2 14])
title ('Trapezi composita ');
TrapeziComposita (f, 0.5 , 100 , 50, true );

subplot (2 ,2 ,2)
axis ([ -5 105 -2 14])
title ('Simpson composita ');
SimpsonComposita (f, 0.5 , 100 , 50, true );
subplot (2 ,2 ,3)
axis ([ -5 105 -2 14])
title ('Trapezi adattativa ');
TrapeziAdattativa (f, 0.5 , 100 , 10^ -3 , true );

subplot (2 ,2 ,4)
axis ([ -5 105 -2 14])
title ('Simpson adattativa ');
SimpsonAdattativa (f, 0.5 , 100 , 10^ -3 , true );
```

Codice 5.7: Esercizio 5.9.

```
% Esercizio 5.9
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:13 CET

f = inline (' -2*x^( -3)* cos(x^ -2) ');

fprintf ('\n\ tFormula composita dei trapezi \n')
for n =1000:1000:10000
    I = TrapeziComposita (f, 1/2 , 100 , n, false );
    fprintf ('n = %d \t I = %5.4 e \t E = %5.4 e\n', n, I, abs(I -( sin
        (10^ -4) - sin (4))));
end
fprintf ('\n\n\ tFormula dei trapezi adattativa \n')
for i =1:4
    tol = 10^ -i;
    [I, p] = TrapeziAdattativa (f, 1/2 , 100 , tol , false );
    fprintf ('tol = %1.1 e \t I = %5.4 e \t E = %5.4 e \t punti = %d\n'
        , tol , I, abs(I -( sin (10^ -4) - sin (4))) , p);
end
```

Codice 5.8: Esercizio 5.10.

```
% Esercizio 5.10
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:13 CET

f = inline (' -2*x^( -3)* cos(x^ -2) ');

fprintf ('\n\ tFormula composita di Simpson \n')
for n =1000:1000:10000
    I = SimpsonComposita (f, 1/2 , 100 , n, false );
    fprintf ('n = %d \t I = %5.4 e \t E = %5.4 e\n', n, I, abs(I -( sin
        (10^ -4) - sin (4))));
end
fprintf ('\n\n\ tFormula di Simpson adattativa \n')
for i =1:5
    tol = 10^ -i;
    [I, p] = SimpsonAdattativa (f, 1/2 , 100 , tol , false );
    fprintf ('tol = %1.1 e \t I = %5.4 e \t E = %5.4 e \t punti = %d\n'
        , tol , I, abs(I -( sin (10^ -4) - sin (4))) , p);
end
```

Capitolo 6

Calcolo del *Google pagerank*

Indice

La tecnica di ordinamento di Google si basa su l'ordinamento per importanza, detto **Google pagerank**. L'idea di base è che "una pagina è importante se viene citata da molte pagine importanti". Per determinare quindi l'importanza di una pagina viene implementato il cosiddetto **random surfer**, ovvero viene modellizzato un ipotetico utente che naviga sul web cliccando a caso sui link che gli si presentano davanti.
Se pensiamo adesso al web come un enorme grafo orientato $W\langle V, E\rangle$, dove Vè l'insieme dei nodi, ovvero le *pagine web*, con $|V| = n$, mentre E è l'insieme degli archi, ovvero i *link* che collegano tra loro le pagine, allora l'importanza x_i della pagina i-esima viene definita come

$$x_i = \sum_{(j \to i) \in E} \frac{x_j}{f_i}, \qquad i = 1, \ldots, n, \tag{6.1}$$

dove $(j \to i)$ indica il link uscente dalla pagina j ed entrante nella pagina i, mentre f_j denota il numero totale di link uscenti dalla pagina j. Quindi l'importanza di una pagina viene ridistribuita equamente tra le pagine in essa citate.

Più formalmente, definendo la matrice

$$L = (l_{ij}) \in \mathbb{R}^{n\times n}, \qquad l_{ij} = \begin{cases} 1, & \text{se } (j \to i) \in E, \\ 0, & \text{altrimenti,} \end{cases}$$

abbiamo che

$$f_j = \sum_{i=1}^{n} l_{ij}, \qquad j = 1, \ldots, n.$$

Definendo allora la matrice

$$H = LF,$$

con

$$F = \begin{pmatrix} f_1^+ & & \\ & \ddots & \\ & & f_n^+ \end{pmatrix}, \qquad f_i^+ = \begin{cases} \frac{1}{f_i}, & \text{se } f_i > 0, \\ 0, & \text{se } f_i = 0, \end{cases}$$

(ovvero, $h_{ij} = \frac{l_{ij}}{f_j}$, oppure 0 se $f_j = 0$), possiamo definire il vettore del *pagerank*

$$\underline{\hat{x}} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix},$$

e riformulare la (6.1) come segue:

$$\underline{\hat{x}} = H\underline{\hat{x}}, \tag{6.2}$$

ovvero $\underline{\hat{x}}$ è l'*autovettore destro* di H relativo all'autovalore 1.
Si osservi che H è una matrice sparsa (cioè con molti elementi nulli) e quindi può essere convenientemente memorizzata in formato compresso per colonne.
Il modello (6.2) è tuttavia incompleto, in quanto se il *random surfer* si trova in una pagina senza link in uscita non potrebbe più uscirne. Si ipotizza allora che quando il *random surfer* si incontra in una pagina senza link uscenti, questo salta casualmente ad una pagina qualsiasi del web. Definiamo a tal proposito i vettori

$$\underline{v} = \begin{pmatrix} \frac{1}{n} \\ \vdots \\ \frac{1}{n} \end{pmatrix}, \qquad \underline{v} = \frac{1}{n}\underline{e}, \qquad \underline{\delta} = \begin{pmatrix} \delta_1 \\ \vdots \\ \delta_n \end{pmatrix}, \quad \delta_i = 1 - f_i f_i^+ = \begin{cases} 1, & \text{se } f_i = 0, \\ 0, & \text{se } f_i > 0. \end{cases} \tag{6.3}$$

e la matrice

$$S \equiv H + \underline{v}\,\underline{\delta}^T. \tag{6.4}$$

Allora la modifica appena esposta si traduce, algebricamente, nel seguente modello per il *pagerank*:

$$\underline{\hat{x}} = S\underline{\hat{x}}. \tag{6.5}$$

Teorema 6.1. *La matrice S definita nella (6.4) soddisfa le seguenti proprietà (le disuguaglianze sono elemento per elemento):*

1. $S \geq 0$;

2. $\underline{e}^T S = \underline{e}^T$, *dove*

$$\underline{e} = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix};$$

3. $\lambda = 1$ *è il raggio spettrale di* S *(dove con raggio spettrale, indicato dal simbolo* $\rho(S)$*, si intende il massimo autovalore, in valore assoluto di una matrice).*

Tuttavia, affinché il *pagerank* abbia significato fisico, vogliamo che sia a componenti strettamente positive ed unico, utilizzando la normalizzazione:

$$||\underline{\hat{x}}||_1 = \underline{e}^T \underline{\hat{x}} = 1. \tag{6.6}$$

Teorema 6.2 (Perron-Frobenius)**.** *Sia* $A > 0$*, allora:*

1. *esiste* $\lambda > 0$*,* $\lambda \in \sigma(A)$ *(dove* $\sigma(A)$ *indica lo spettro di* A*, ovvero l'insieme dei suoi autovalori), tale che* $\lambda = \rho(A)$*. inoltre,* λ *è un autovalore semplice (ovvero se è una radice semplice del polinomio caratteristico* $p(\lambda) = det(A - \lambda I)$*) e separa in modulo ogni altro autovalore di* A*;*

2. *ad esso corrisponde un autovettore a componenti positive.*

Teorema 6.3 (Perron-Frobenius (forma debole))**.** *Sia* $A \geq 0$*, allora:*

1. *esiste* $\lambda \geq 0$*,* $\lambda \in \sigma(A)$*, tale che* $\lambda = \rho(A)$*;*

2. *ad esso corrisponde un autovettore a componenti non negative.*

Perciò, per il Teorema 6.2, se avessimo $S > 0$, avremo sicuramente l'autovettore del *pagerank* unico e a componenti positive.
Modifichiamo quindi ulteriormente la formulazione del *pagerank*, introducendo una probabilità $p \in (0, 1)$, come segue:

$$\underline{\hat{x}} = S(p)\underline{\hat{x}} \equiv (pS + (1-p)\underline{v}\,\underline{e}^T)\underline{\hat{x}}. \tag{6.7}$$

Questo equivale a dire che, con probabilità p il *random surfer* si comporterà esattamente come descritto nella (6.5), mentre con probabilità $(1 - p)$ salterà casualmente ad una qualsiasi pagina del web.
Si può vedere che la matrice $S(p)$ verifica le ipotesi del Teorema 6.2, ovvero $S(p) > 0$, quindi esiste il vettore del *pagerank* a componenti positive e il calcolo di tale vettore di riduce al calcolo dell'autovettore della matrice $S(p)$ relativo all'autovalore $\lambda = 1$. Inoltre si ha che

$$||S(p)||_1 = ||\underline{e}^T S(p)||_\infty = 1, \qquad p \in (0, 1). \tag{6.8}$$

6.1 Il metodo delle potenze

Studiamo adesso il metodo delle potenze, grazie al quale potremo calcolare l'*autovalore dominante* (*semplice*) ed il corrispondente *autovettore* di una data matrice. Ovvero calcolare, data $A \in \mathbb{R}^{n\times n}$, λ_1 e $\underline{v_1} \neq 0$, tali che

$$A\underline{v_1} = \lambda_1 \underline{v_1}, \tag{6.9}$$

con

$$|\lambda_1| = \rho(A) > |\lambda_2| \geq \cdots \geq |\lambda_n|, \qquad \{\lambda_i\} = \sigma(A).$$

Supponiamo, per semplicità, che gli autovalori siano tutti *distinti* e *reali*:

$$\lambda_i \neq \lambda_j, \qquad i \neq j, \qquad \lambda_i \in \mathbb{R}, \qquad i = 1, \ldots, n,$$

in questo modo anche gli autovettori corrispondenti saranno a componenti reali:

$$0 \neq \underline{v_i} \in \mathbb{R}^n, \qquad i = 1, \ldots, n.$$

Possiamo riscrivere l'equazione (6.9) in forma matriciale come segue:

$$AV = V\Lambda,$$
$$V = (\underline{v_1}, \ldots, \underline{v_n}), \qquad \Lambda = \begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{pmatrix},$$

ovvero con V matrice che presenta gli autovettori lungo le sue colonne e Λ matrice con gli autovettori corrispondenti sulla sua diagonale principale.

Teorema 6.4. *Se gli autovalori di una matrice sono distinti, allora i corrispondenti autovettori sono linearmente indipendenti.*

Essendo, per il teorema appena enunciato, gli autovettori di A linearmente indipendenti, essi possono costituire una base per $\mathbb{R}^n$. Consideriamo quindi un vettore casuale costruito su tale base:

$$\underline{x_0} = V\alpha = (\underline{v_1}, \ldots, \underline{v_n}) \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix} = \sum_{i=1}^{n} \alpha_i \underline{v_i}, \qquad \alpha_i \in \mathbb{R},$$

e costruiamo la successione $\underline{x_k} = A\underline{x_{k-1}} = \cdots = A^k\underline{x_0} = A^kV\alpha$. Per la (6.9)

$$\underline{x_k} = A^kV\alpha = A^{k-1}(AV)\alpha = A^{k-1}V\Lambda\alpha = \cdots = V\Lambda^k\alpha = \sum_{i=1}^{n} \underline{v_i}\lambda_i^k\alpha_i$$
$$= \underline{v_1}\lambda_1^k\alpha_1 + \sum_{i=2}^{n} \underline{v_i}\lambda_i^k\alpha_i = \lambda_1^k \left(\alpha_1\underline{v_1} + \sum_{i=2}^{n} \left(\frac{\lambda_i}{\lambda_1}\right)^k \alpha_i\underline{v_i} \right).$$

Dato che, per il Teorema 6.2, λ_1 separa in modulo tutti gli altri autovalori, si ricava che

$$\left(\frac{\lambda_i}{\lambda_1}\right)^k \to 0, \qquad k \to \infty, \qquad i = 2, \ldots, n.$$

Quindi si può stimare che

$$\underline{x_k} = \lambda_1^k \left(\alpha_1 \underline{v_1} + \sum_{i=2}^{n} \left(\frac{\lambda_i}{\lambda_1}\right)^k \alpha_i \underline{v_i}\right) \approx \lambda_1^k \alpha_1 \underline{v_1}, \qquad k \gg 1.$$

Se proviamo a moltiplicare il vettore k-esimo della successione con il $(k+1)$-esimo, otteniamo:

$$\underline{x_k}^T \underline{x_{k+1}} \approx (\lambda_1^k \alpha_1 \underline{v_1})^T (\lambda_1^{k+1} \alpha_1 \underline{v_1}) = \lambda_1 (\lambda_1^k \alpha_1 \underline{v_1})^T (\lambda_1^k \alpha_1 \underline{v_1}) = \lambda_1 \underline{x_k}^T \underline{x_k} = \lambda_1 ||\underline{x_k}||_2^2,$$

dalla quale si ricava, infine, un'approssimazione per l'autovalore dominante:

$$\lambda_1 \approx \frac{\underline{x_k}^T \underline{x_{k+1}}}{||\underline{x_k}||_2^2}, \qquad k \gg 1,$$

prendendo $\underline{x_{k+1}}$ come approssimazione dell'autovettore dominante corrispondente.
Ovviamente, in fase di implementazione, si deve porre attenzione a possibili *overflow* o *underflow*. Per ovviare a questo tipo di problemi, ad ogni passo del metodo delle potenze si normalizza il vettore $\underline{x_k}$.
Di seguito, proponiamo un'implementazione in MATLAB del metodo delle potenze:

Codice 6.1: Metodo delle potenze.

```
% [l1, x1] = metodoPotenze(A, tol)
% Generico metodo delle potenze che calcola l'autovalore dominante
% semplice e l'autovettore corrispondente di una matrice entro una
% certa tolleranza.
%
% Input:
%   -A: la matrice di cui calcolare l'autovalore dominante e
%   l'autovettore corrisondente;
%   -tol: la tolleranza richiesta per l'approssimazione
%   dell'autovalore.
% Output:
%   -l1: l'approssimazione dell'autovalore dominante;
%   -x1: l'approssimazione dell'autovettore corrispondente
%   all'autovalore dominante.
%
% Autore: Tommaso Papini,
% Ultima modifica: 26 Ottobre 2012, 12:09 CEST

function [l1, x1] = metodoPotenze(A, tol)
    [m,n] = size(A);
    if m~=n
        error('La matrice non è quadrata!');
```

```
    end
    x1 = rand(n, 1);
    l1 = inf;
    err = inf;
    while err>tol
        x0 = x1/norm(x1, 2);
        x1 = A*x0;
        l0 = l1;
        l1 = x0'*x1;
        err = abs(l0-l1);
    end
end
```

6.1.1 Metodo delle potenze per il *Google pagerank*

Nel caso del calcolo del *Google pagerank* sappiamo che l'autovalore corrispondente all'autovettore dominante è $\lambda = 1$, quindi non ci sarà bisogno né di calcolarlo, né di normalizzare di volta in volta il vettore $\underline{x_k}$. Per questo motivo non si rischia di incorrere in *overflow*/*underflow*.
Se inoltre prendiamo come vettore iniziale un vettore $\underline{x_0}$ tale che

$$\underline{x_0} > 0, \qquad \underline{e}^T \underline{x_0} = ||\underline{x_0}||_1 = 1,$$

per induzione (grazie alla (6.8)) si ha:

$$\begin{aligned} &\underline{x_k} = S(p)\underline{x_{k-1}} > 0, \\ &||\underline{x_k}||_1 = \underbrace{\underline{e}^T S(p)}_{\underline{e}^T} \underline{x_{k-1}} = \underline{e}^T \underline{x_{k-1}} = ||\underline{x_{k-1}}||_1 = 1. \end{aligned} \tag{6.10}$$

Quindi, per la (6.7), possiamo ricavare come segue un criterio d'arresto per il metodo delle potenze:

$$\begin{aligned} ||\underline{x_k} - \underline{\hat{x}}||_1 &= ||S(p)\underline{x_{k-1}} - S(p)\underline{\hat{x}}||_1 = ||S(p)(\underline{x_{k-1}} - \underline{\hat{x}})||_1 = \\ &= ||pS(\underline{x_{k-1}} - \underline{\hat{x}}) + (1-p)\underline{v}\ \underbrace{\underline{e}^T(\underline{x_{k-1}} - \underline{\hat{x}})}_{=0}||_1 = ||pS(\underline{x_{k-1}} - \underline{\hat{x}})||_1 \leq \\ &\leq p \underbrace{||S||_1}_{=||\underline{e}^T S||_\infty = 1} \cdot ||\underline{x_{k-1}} - \underline{\hat{x}}||_1 = p||\underline{x_{k-1}} - \underline{\hat{x}}||_1 \leq \cdots \leq \\ &\leq p^k ||\underline{x_0} - \underline{\hat{x}}||_1 \leq 2p^k. \end{aligned}$$

Quindi si ricava che per avere un risultato entro la tolleranza fissata *tol*, basta fare, al più, $\frac{\log tol - \log 2}{\log p}$. Si osserva che valori di p vicini ad 1 riproducono un modello più fedele del web ma degradano la velocità di convergenza del metodo delle potenze. Studi in proposito indicano $p = 0.85$ come un buon compromesso tra *accuratezza* ed *efficienza*.

Per quanto riguarda il costo computazionale del metodo delle potenze, si ha che ogni iterazione si riduce al calcolo di (ricordando che $\underline{v} = \frac{1}{n}\underline{e}$):

$$\begin{aligned} \underline{x_k} &= pS\underline{x_{k-1}} + (1-p)\underline{v}\ \underbrace{\underline{e}^T\underline{x_{k-1}}}_{=1} = \\ &= p(H + \underline{v}\ \underline{\delta}^T)\underline{x_{k-1}} + (1-p)\underline{v} = \\ &= p(H\underline{x_{k-1}}) + \frac{1 + p(\underline{\delta}^T\underline{x_{k-1}} - 1)}{n}\underline{e}. \end{aligned} \tag{6.11}$$

Quest'operazione è composta da:

- un prodotto matrice-vettore, $H\underline{x_{k-1}}$, del costo di $O(n)$ `flops`;
- un prodotto scalare, $\underline{\delta}^T\underline{x_{k-1}}$, del costo di $2n$ `flops`;
- l'operazione complessiva, che è del tipo

  $$\text{vettore} = \text{scalare*vettore} + \text{scalare*vettore},$$

  che costa altri $3n$ `flops`.

Quindi il costo complessivo si vede essere **lineare** rispetto alla dimensione del problema, sia in termini di operazioni eseguite che di memoria occupata (memorizzando la matrice sparsa H in formato compresso per colonne).
Secondo le osservazioni appena fatte, il metodo delle potenze per il calcolo del Google *pagerank* può essere implementato efficientemente come segue:

Codice 6.2: Metodo delle potenze per il calcolo del Google *pagerank*.

```
% x = metodoPotenzeGooglePagerank(L, p, tol)
% Calcolo dell'autovettore corrispondente all'autovalore dominante 1
% utilizzando il metodo delle potenze ottimizzato per il calcolo del
% Google pagerank. La matrice L in input viene riscritta con gli
% elementi della matrice H.
%
% Input:
%   -L: matrice contenente le informazioni sui link presenti tra le
%   diverse pagine web (L(i, j)=1 se esiste un link tra la pagina j e
%   la pagina i, altrimenti 0);
%   -p: probabilità con cui il random surfer sceglie di seguire un link
%   presente nella pagina (con probabilità 1-p salta verso una pagina a
%   caso del web);
%   -tol: tolleranza richiesta.
% Output:
%   -x: approssimazione dell'autovettore relativo all'autovalore
%   dominante.
%
% Autore: Tommaso Papini,
% Ultima modifica: 27 Ottobre 2012, 10:22 CEST
```

```
function [x] = metodoPotenzeGooglePagerank(L, p, tol)
    [m,n] = size(L);
    if m~=n
        error('La matrice non è quadrata!');
    end

    delta = zeros(n, 1);
    for j=1:n
        for i=1:n
            delta(j) = delta(j) + L(i, j);
        end
    end
    for j=1:n
        for i=1:n
            if delta(j)==0
                L(i, j) = 0;
            else
                L(i, j) = L(i, j)/delta(j);
            end
        end
        if delta(j)==0
            delta(j)=1;
        else
            delta(j)=0;
        end
    end

    x = rand(n,1);
    x = x/norm(x);
    imax = (log(tol)-log(2))/(log(p));
    for i=1:imax
        x = p*(L*x)+((1+p*(delta'*x -1))/n)*ones(n,1);
    end
end
```

6.2 Risoluzione iterativa di sistemi lineari

Un approccio differente al metodo delle potenze per il calcolo del *Google pagerank* è la risoluzione iterativa di sistemi lineari.
Si vede che, per le (6.3), (6.6) e (6.7):

$$\underline{\hat{x}} = S(p)\underline{\hat{x}} = pS\underline{\hat{x}} + (1-p)\frac{1}{n}\underline{e}\underbrace{\underline{e}^T\underline{\hat{x}}}_{=1},$$

$$\underline{\hat{x}} - pS\underline{\hat{x}} = \frac{1-p}{n}\underline{e},$$

$$\underbrace{(I - pS)}_{\equiv A}\underline{\hat{x}} = \underbrace{\frac{1-p}{n}\underline{e}}_{\equiv \underline{b}}.$$

Quindi il problema (6.7) può essere riformulato come segue:

$$A\underline{\hat{x}} = \underline{b}. \tag{6.12}$$

Risulta chiaro che, data la sparsità della matrice A, non è conveniente utilizzare i metodi di fattorizzazione visti nel Capitolo 3.Un'importante caratteristica della matrice A è quella di poter essere scritta nella forma

$$A = I - B, \qquad B \geq 0, \qquad \rho(B) < 1, \tag{6.13}$$

infatti risulta che $B = I - I + pS = pS$ e si ha che $S \geq 0$, $\rho(S) = 1$ e $p < 1$.

Teorema 6.5. *Una matrice B ha raggio spettrale minore di 1, ovvero $\rho(B) < 1$, se e solo se*

$$B^i \to O, \qquad i \to \infty.$$

Definizione 6.1. *Per il Teorema 6.5, una matrice avente raggio spettrale minore di 1 si dice **convergente**.*

Definizione 6.2. *Una matrice del tipo αA, con A definita nella (6.13) ed $\alpha > 0$, si dice **M-matrice**.*

Teorema 6.6. *Se A è una M-matrice, allora $A^{-1} \geq 0$*

Le *M-matrici* inoltre risultano essere **matrici monotone**, ovvero tali che

$$A\underline{x} \leq \underline{y} \qquad \Rightarrow \qquad \underline{x} \leq A^{-1}\underline{y},$$

o, più in generale, se C è una matrice delle stesse dimensioni di A, tali che

$$A \leq C \qquad \Rightarrow \qquad I \leq A^{-1}C, \quad I \leq CA^{-1},$$

dove le disuguaglianze sono da intendersi elemento per elemento.

Definizione 6.3. *Si parla di **splitting** della matrice A qualora si possa scrivere*

$$A = M - N, \tag{6.14}$$

con $det(M) \neq 0$.

Lo *splitting* della matrice A viene definito in modo tale che i sistemi lineari con la matrice M come matrice dei coefficienti siano immediatamente risolvibili senza bisogno di fattorizzazione. Questo perché, per le (6.12) e (6.14), si può definire il seguente metodo iterativo per calcolare un'approssimazione del vettore di *pagerank*:

$$M\underline{x_k} = N\underline{x_{k-1}} + \underline{b}, \qquad k \geq 1, \tag{6.15}$$

partendo da un'approssimazione data x_0. Definiamo l'errore al passo k, come

$$\underline{e_k} = \underline{x_k} - \underline{\hat{x}}. \tag{6.16}$$

Osservando che

$$M\underline{\hat{x}} = N\underline{\hat{x}} + \underline{b}$$

e per la (6.15) si ottiene la seguente *equazione dell'errore*:

$$\begin{aligned}\underline{e_k} &= \underline{x_k} - \underline{\hat{x}} = M^{-1}(N\underline{x_{k-1}} + \underline{b}) - M^{-1}(N\underline{\hat{x}} + \underline{b}) = \\ &= M^{-1}(N\underline{x_{k-1}} + \underline{b} - N\underline{\hat{x}} - \underline{b}) = \\ &= M^{-1}N(\underline{x_{k-1}} - \underline{\hat{x}}) = \\ &= M^{-1}N\underline{e_{k-1}}, \qquad k \geq 1,\end{aligned}$$

dalla quale si ricava

$$\underline{e_k} = (M^{-1}N)^k \underline{e_0}, \qquad k \geq 1.$$

Quindi, per il Teorema 6.5, il metodo sarà convergente se e solo se la matrice $M^{-1}N$ è convergente, ovvero se e solo se $\rho(M^{-1}N) < 1$.

Definizione 6.4. *La matrice $M^{-1}N$ dello splitting (6.14) si dice* ***matrice di iterazione*** *del metodo (6.15).*

Per quanto visto nello studio del metodo delle potenze e supponendo che la matrice d'iterazione abbia un autovalore dominante semplice, si ottiene la stima

$$||\underline{e_k}|| \approx \rho(M^{-1}N)^k ||\underline{e_0}||, \qquad k \gg 1.$$

Quindi il raggio spettrale della matrice d'iterazione ci permette di confrontare tra loro metodi iterativi definiti da diversi splitting della matrice A.

6.2.1 *Splitting* regolari di matrici

Definizione 6.5. *Lo splitting (6.14) si dice* ***regolare*** *se*

$$m^{-1} \geq 0, \qquad N \geq 0.$$

Lemma 6.1. *Siano $A, B \in \mathbb{R}^{n\times n}$, $A \geq B \geq 0$. Allora $A^i \geq B^i \geq 0$, $i \geq 0$.*

Lemma 6.2. *Siano $A, B \in \mathbb{R}^{n\times n}$, $A \geq B \geq 0$. Allora $\rho(A) \geq \rho(B)$.*

Teorema 6.7. *Se lo splitting (6.14) è regolare e $A^{-1} \geq 0$, allora il metodo iterativo (6.15) è convergente*

Corollario 6.1. *Se lo splitting (6.14) è regolare ed A è una M-matrice, allora il metodo iterativo (6.15) è convergente (vedi Teorema 6.6).*

Per i Lemmi 6.1 e 6.2 si hanno i seguenti due corollari:

Corollario 6.2. *Se $A = \alpha(I - B)$ è una M-matrice e $A = M - N$, con $0 \leq N \leq \alpha B$, allora M è nonsingolare e lo splitting è regolare. Pertanto, il metodo (6.15) è convergente.*

Corollario 6.3. *Se A è una M-matrice e la matrice M in (6.14) è ottenuta ponendo a 0 gli elementi extradiagonali di A, allora lo splitting (6.14) è regolare. Pertanto il metodo iterativo (6.15) è convergente.*

Infine quest'ultimo Teorema ci consente di comparare tra loro le velocità di due metodi iterativi costruiti su due diversi splitting regolari di una stessa matrice:

Teorema 6.8. *Siano $A = M_1 - N_1 = M_2 - N_2$ due splitting regolati di A, matrice tale che $A^{-1} \geq 0$. Se $N_1 \leq N_2$, allora $\rho(M_1^{-1}N_1) \leq \rho(M_2^{-1}N_2)$, ovvero il primo splitting converge più velocemente del secondo.*

6.2.2 Criteri d'arresto

Per quanto riguarda i criteri d'arresto del metodo iterativo (6.15) si può pensare a controllare la norma $||\underline{x_k} - \underline{x_{k-1}}||$. Un'altra possibilità, invece, è quella di controllare la norma del *vettore residuo*:

$$\underline{r_k} = A\underline{x_k} - \underline{b}. \tag{6.17}$$

Si osserva che il vettore residuo $\underline{r_k}$ ed il vettore d'errore $\underline{e_k}$ sono in stretta relazione tra loro, infatti:

$$A\underline{e_k} = \underline{r_k}.$$

6.2.3 I metodi di Jacobi e Gauss-Seidel

Consideriamo il partizionamento della matrice A

$$A = D - L - U, \tag{6.18}$$

con

- D diagonale,
- L strettamente triangolare inferiore e
- U strettamente triangolare superiore.

Teorema 6.9. *Se la matrice A in (6.18) è una M-matrice, allora $D, L, U \geq 0$. In particolare D ha elementi diagonali positivi ($D > 0$).*

Teorema 6.10. *Se A in (6.18) è una M-matrice, allora tali sono anche le matrici D e $D - L$.*

Definizione 6.6.

- *Lo splitting*
$$A = M_J - N_J \equiv D - (L + U)$$
definisce il metodo iterativo di ***Jacobi****.*
- *Lo splitting*
$$A = M_{GS} - N_{GS} \equiv (D - L) - U$$
definisce il metodo iterativo di ***Gauss-Seidel***

Corollario 6.4. *Se A in (6.18) è una M-matrice, allora gli splitting di Jacobi e di Gauss-Seidel sono regolari e quindi i relativi metodi iterativi sono convergenti.*

Si osserva che ad ogni iterazione il metodo di Jacobi richiede la risoluzione di un sistema diagonale, mentre il metodo di Gauss-Seidel richiede la risoluzione di un sistema triangolare inferiore.

Corollario 6.5. *Se A in (6.18) è una M-matrice, allora*

$$\rho((D-L)^{-1}U) \leq \rho(D^{-1}(L+U)) < 1.$$

Ovvero, in generale, il metodo di Gauss-Seidel converge più velocemente del metodo di Jacobi.

Infatti, essendo $L \geq 0$ per il Teorema 6.9, sicuramente $U \leq (L+U)$ (vedi Teorema 6.8).

Concludendo, osserviamo che, per quanto detto fin'ora sugli splitting regolari di matrici, i metodi iterativi di Jacobi e di Gauss-Seidel (ed in particolare quest'ultimo) possono essere convenientemente utilizzati per calcolare un'approssimazione del vettore di *pagerank* di Google.

Si mostrano di seguito le implementazioni in MATLAB dei metodi di Jacobi e di Gauss-Seidel;

Codice 6.3: Metodo di Jacobi.

```
% [x1, i] = jacobi(A, b, x0, tol)
% Risoluzione iterativa di sistemi lineari tramite il metodo di Jacobi.
%
% Input:
%   -A: la matrice dei coefficienti;
%   -b: vettore dei termini noti;
%   -x0: approssimazione iniziale della soluzione;
%   -tol: la tolleranza richiesta.
% Output:
%   -x1: l'approssimazione calcolata del vettore soluzione;
%   -i: numero di iterazioni eseguite.
%
% Autore: Tommaso Papini,
% Ultima modifica: 13 Maggio 2013, 13:16 CEST

function [x1, i] = jacobi(A, b, x0, tol)
    M = diag(diag(A));
    i = 0;
    r = A*x0-b;
    x1=x0;
    while norm(r)>tol
        i = i+1;
        u = diagonale(M, r);
        x1 = x0-u;
```

```
        x0 = x1;
        r = A*x0-b;
    end
end
```

Codice 6.4: Metodo di Gauss-Seidel.

```
% [x1, i] = gaussSeidel(A, b, x0, tol)
% Risoluzione iterativa di sistemi lineari tramite il metodo di
% Gauss-Seidel.
%
% Input:
%   -A: la matrice dei coefficienti;
%   -b: vettore dei termini noti;
%   -x0: approssimazione iniziale della soluzione;
%   -tol: la tolleranza richiesta.
% Output:
%   -x1: l'approssimazione calcolata del vettore soluzione;
%   -i: numero di iterazioni eseguite.
%
% Autore: Tommaso Papini,
% Ultima modifica: 26 Ottobre 2012, 13:00 CEST

function [x1, i] = gaussSeidel(A, b, x0, tol)
    M = tril(A);
    i = 0;
    r = A*x0-b;
    while norm(r)>tol
        i = i+1;
        u = triangolareInfCol(M, r);
        x1 = x0-u;
        x0 = x1;
        r = A*x0-b;
    end
end
```

Esercizi

Esercizio 6.1 (Teorema di Gershgorin)**.** *Dimostrare che gli autovalori di una matrice* $A = (a_{ij}) \in \mathbb{C}^{n\times n}$ *sono contenuti nell'insieme*

$$\mathcal{D} = \bigcup_{i=1}^{n} \mathcal{D}_i, \qquad \mathcal{D}_i = \left\{ \lambda \in \mathbb{C} : |\lambda - a_{ii}| \leq \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{ij}| \right\}, \quad i = 1, \ldots, n.$$

Soluzione.
Sia $\lambda \in \sigma A$ ed $\underline{x}$ il corrispondente autovettore ($\underline{x} \neq \underline{0}$ e $A\underline{x} = \lambda\underline{x}$) quindi $\underline{e}_i^T A\underline{x} = \lambda \underline{e}_i^T \underline{x}$ per $i = 1, \ldots, n$ cioè $\sum_{j=1}^{n} a_{i,j}x_j = \lambda x_i$; posto i tale che $|x_i| \geq |x_j|$ ($x_i \neq 0$)

risulta $\lambda = \frac{\sum_{j=1}^{n} a_{i,j}x_j}{x_i} = a_{i,i} + \sum_{j=1\ j\neq i}^{n} a_{i,j}\frac{x_j}{x_i}$ ovvero $\lambda - a_{i,i} = \sum_{j=1\ j\neq i}^{n} a_{i,j}\frac{x_j}{x_i}$. Passando ai valori assoluti

$$|\lambda - a_{i,i}| = \left| \sum_{j=1\ j\neq i}^{n} a_{i,j}\frac{x_j}{x_i} \right| \leq \sum_{j=1\ j\neq i}^{n} \left| a_{i,j}\frac{x_j}{x_i} \right| \leq \sum_{j=1\ j\neq i}^{n} |a_{i,j}|$$

poiché $\left|\frac{x_j}{x_i}\right| \leq 1$ in quanto $|x_i| \geq |x_j|$. Segue $\lambda \in \bigcup_{i=1}^{n} \mathcal{D}_i$ da cui $\sigma A \subseteq \mathcal{D}$.

Esercizio 6.2. *Utilizzare il metodo delle potenze per approssimare l'autovalore dominante della matrice*

$$A_n = \begin{pmatrix} 2 & -1 & & \\ -1 & 2 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{pmatrix} \in \mathbb{R}^{n\times n},$$

per valori crescenti di n. *Verificare numericamente che questo è dato da* $2\left(1+\cos\frac{\pi}{n+1}\right)$.

Soluzione.
Prima di applicare il metodo delle potenze, mostriamo che A_n può essere scritta come una M-matrice, infatti: $A_n = 2(I_n - B_n)$ dove

$$B_n = \begin{pmatrix} 0 & 1/2 & & \\ 1/2 & 0 & \ddots & \\ & \ddots & \ddots & 1/2 \\ & & 1/2 & 0 \end{pmatrix} \in \mathbb{R}^{n\times n}.$$

Risulta $\lambda_j(B_n) = \cos\frac{j\pi}{n+1}$, $|\lambda_j| \leq 1$, per $j = 1, \ldots, n$; segue $\lambda_j(A_n) = 2(\lambda_j(I_n) - \lambda_j(B_n)) = 2(1 - \cos\frac{j\pi}{n+1})$; il massimo è $\lambda = 2(1+\cos\frac{\pi}{n+1})$. La verifica numerica di questo risultato è riportata nelle seguenti tabelle.

Tolleranza $tol = 10^{-5}$			
n	λ_1	$2\left(1+\cos\frac{\pi}{n+1}\right)$	scostamento
10	3.9189	3.9190	0.0001
15	3.9614	3.9616	0.0002
20	3.9774	3.9777	0.0003
25	3.9853	3.9854	0.0002
30	3.9891	3.9897	0.0006
35	3.9921	3.9924	0.0003
40	3.9929	3.9941	0.0012
45	3.9938	3.9953	0.0015
50	3.9947	3.9962	0.0015

Tolleranza $tol = 10^{-7}$			
n	λ_1	$2\left(1+\cos\frac{\pi}{n+1}\right)$	scostamento
5	3.7321	3.7321	0.0000
10	3.9190	3.9190	0.0000
15	3.9616	3.9616	0.0000
20	3.9777	3.9777	0.0000
25	3.9854	3.9854	0.0000
30	3.9897	3.9897	0.0000
35	3.9924	3.9924	0.0000
40	3.9941	3.9941	0.0000
45	3.9953	3.9953	0.0000
50	3.9962	3.9962	0.0000

Riferimenti MATLAB
Codice 6.5 (pagina 223)

Esercizio 6.3. *Dimostrare i Corollari 6.2 e 6.3.*

Soluzione.
Corollario 6.2
Sia $A = \alpha(I - B) = M - N$ con $0 \leq N \leq \alpha B$ quindi $M = \alpha I - \alpha B + N = \alpha I - \alpha B + \alpha Q = \alpha(I - (B - Q))$ con $\alpha Q = N \leq \alpha B$ e $0 \leq Q \leq B$. Dato che $0 \leq B - Q \leq B$, per il Lemma 6.2, $\rho(B - Q) \leq \rho(B) \leq 1$. Quindi M è una M-matrice; lo splitting è regolare infatti $M^{-1} \geq 0$ (M è una M-matrice) e $B - Q \geq 0$.

Corollario 6.3
Sia $A = M - N$ M-matrice con $M = diag(A)$ allora la matrice $A - M = B$ avrà elementi nulli sulla diagonale e, altrove, gli elementi di A. Poiché $A = \alpha(I - B) = \alpha I - \alpha B = M - N$, risulta $\alpha B = N$ quindi, per il Corollario 6.2, lo splitting è regolare ed il metodo è convergente.

Esercizio 6.4. *Dimostrare il Teorema 6.9.*

Soluzione.
Poiché A è una M-matrice, essa può essere scritta nella forma $A = \alpha(I - B)$ con $B \geq 0$ e $\rho(B) < 1$; inoltre, per ipotesi, $A = D - L - U$ da cui si deduce che $D = \alpha(I - diag(B))$ e $(L + U) = \alpha(B - diag(B))$.
La matrice $(L + U) = \alpha(B - diag(B))$ risulta maggiore di zero in quanto $B > 0 \rightarrow (B - diag(B)) > 0$. La matrice D ha elementi positivi sulla diagonale: supponiamo per assurdo $a_{i,i} \leq 0$: l'i-esima riga di A, negativa, è data da $A\underline{e}_i \leq \underline{0}$ dato che A è una M-matrice risulta $\underline{e}_i \leq A^{-1}\underline{0} = \underline{0}$ assurdo poiché $\underline{e}_i \geq \underline{0}, \forall i$.

Esercizio 6.5. *Tenendo conto della (6.10), riformulare il metodo delle potenze (6.11) per il calcolo del Google pagerank come metodo iterativo definito da uno splitting regolare.*

Soluzione.
Il problema del *Google pagerank* è $S(p)\underline{\hat{x}} = \underline{\hat{x}}$ dove $S(p) = pS + (1-p)\underline{v}\,\underline{e}^T$. Sostituendo, risulta

$$(pS + (1-p)\underline{v}\,\underline{e}^T)\underline{\hat{x}} = \underline{\hat{x}} \quad \Rightarrow \quad (I - pS)\underline{\hat{x}} = (1-p)\underline{v}\,\underline{e}^T\underline{\hat{x}}$$

Dato che $\underline{v} = \frac{1}{n}\underline{e}$ ed $e^T\underline{\hat{x}} = 1$ si ricava

$$(I - pS)\underline{\hat{x}} = \frac{1-p}{n}\underline{e}.$$

Si può infine definire il seguente metodo iterativo:

$$I\underline{x}_{k+1} = pS\underline{x}_k + \frac{1-p}{n}\underline{e}.$$

Il metodo è convergente in quanto la matrice di iterazione ha raggio spettrale minore di 1: $\rho(I^{-1}pS) = \rho(pS) < 1$ dato che $p \in (0,1)$ e $\rho(S) = 1$.

Esercizio 6.6. *Dimostrare che il metodo di Jacobi converge asintoticamente in un numero minore di iterazioni, rispetto al metodo delle potenze (6.11) per il calcolo del Google pagerank.*

Soluzione.
La matrice di iterazione di Jacobi ha raggio spettrale minore di 1 mentre, nel calcolo del *Google pagerank*, l'autovalore dominante è $\lambda = 1$ quindi il raggio spettrale di tale matrice è esattamente 1. Poiché il raggio spettrale della matrice di iterazione del metodo di Jacobi è minore del raggio spettrale della matrice del *Google pagerank*, il metodo di Jacobi converge in asintoticamente in un numero minore di iterazioni, rispetto al metodo delle potenze per il calcolo del *Google pagerank*.

Esercizio 6.7. *Dimostrare che, se A è diagonale dominante, per riga o per colonna, il metodo di Jacobi è convergente.*

Soluzione.
Nel metodo di Jacobi si ha $A = M - N = D - (L + U)$ quindi

$$M = \begin{pmatrix} a_{1,1} & & & \\ & \ddots & & \\ & & \ddots & \\ & & & a_{n,n} \end{pmatrix} \text{ e } N = \begin{pmatrix} 0 & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & a_{n-1,n} \\ a_{n,1} & \dots & a_{n,n-1} & 0 \end{pmatrix}.$$

Si ha

$$B = M^{-1}N = \begin{pmatrix} \frac{1}{a_{1,1}} & & & \\ & \ddots & & \\ & & \ddots & \\ & & & \frac{1}{a_{n,n}} \end{pmatrix} \begin{pmatrix} 0 & a_{1,2} & \dots & a_{1,n} \\ a_{2,1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & a_{n-1,n} \\ a_{n,1} & \dots & a_{n,n-1} & 0 \end{pmatrix} =$$

$$= \begin{pmatrix} 0 & \frac{a_{1,2}}{a_{1,1}} & \dots & \frac{a_{1,n}}{a_{1,1}} \\ \frac{a_{2,1}}{a_{2,2}} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \frac{a_{n-1,n}}{a_{n-1,n-1}} \\ \frac{a_{n,1}}{a_{n,n}} & \dots & \frac{a_{n,n-1}}{a_{n,n}} & 0 \end{pmatrix},$$

per il Teorema di Gershgorin risulta

$$\mathcal{D}_i = \left\{ \lambda \in \mathbb{C} \,:\, |\lambda - b_{i,i}| \leq \sum_{j=1\, j\neq i}^{n} |b_{i,j}| \right\} = \left\{ \lambda \in \mathbb{C} \,:\, |\lambda| \leq \sum_{j=1\, j\neq i}^{n} \left| \frac{a_{i,j}}{a_{i,i}} \right| \right\};$$

supposta A a diagonale dominante per righe, $|a_{i,i}| \geq \sum_{j=1,\, j\neq i}^{n} |a_{i,j}|$, risulta $|\lambda| \leq \frac{1}{|a_{i,i}|} \sum_{j=1\, j\neq i}^{n} a_{i,j} < 1$. Ogni $\mathcal{D}_i$ è centrato in 0 e ha raggio minore di 1 quindi $\mathcal{D} = \bigcup_{i=1}^{n} \mathcal{D}_i$ è centrato in 0 e ha raggio pari al raggio massimo dei $\mathcal{D}_i$ ma sempre minore di 1. Dato che $\lambda(A) = \lambda(A^T)$, il risultato vale anche se A è a diagonale dominante per colonne; il metodo di Jacobi è dunque convergente per matrici a diagonale dominante.

Esercizio 6.8. *Dimostrare che, se A è diagonale dominante, per riga o per colonna, il metodo di Gauss-Seidel è convergente.*

Soluzione.
Nel metodo di Gauss-Seidel si ha $A = M - N = (D - L) - U$; sia $\lambda \in \sigma(M^{-}1N)$ quindi λ è tale che $\det(M^{-1}N - \lambda I) = \det(M^{-1}(N - \lambda M)) = \det(M^{-1}) \det(N - \lambda M) = 0$. Dato che, per definizione di splitting, $\det(M^{-1}) \neq 0$ deve risultare $\det(N - \lambda M) = 0 = \det(\lambda M - N)$; sia $H = \lambda M - N$ matrice singolare e supponiamo, per assurdo, $|\lambda| \geq 1$. Risulta

$$H = \begin{cases} \lambda a_{i,j} & \text{se } i \geq j, \\ a_{i,j} & \text{altrimenti,} \end{cases};$$

quindi H è a diagonale dominante ma

$$\sum_{j=1\, j\neq i}^{n} |h_{i,j}| \leq |\lambda| \sum_{j=1\, j\neq i}^{n} |a_{i,j}| < |\lambda||a_{i,i}| = |h_{i,i}|.$$

Si ha una contraddizione poiché le matrici a diagonale dominanti sono non singolari, dunque $|\lambda| < 1$ e quindi il metodo di Gauss-Seidel è convergente per matrici a diagonale dominante.

Esercizio 6.9. *Se A è sdp, il metodo di Gauss-Seidel risulta essere convergente. Dimostrare questo risultato nel caso (assai più semplice) in cui l'autovalore di massimo modulo della matrice di iterazione sia reale.*
(Suggerimento: considerare il sistema lineare equivalente

$$(D^{-\frac{1}{2}}AD^{-\frac{1}{2}})(D^{\frac{1}{2}}\underline{x}) = (D^{-\frac{1}{2}}\underline{b}), \qquad D^{\frac{1}{2}} = diag(\sqrt{a_{11}}, \ldots, \sqrt{a_{nn}}),$$

la cui matrice dei coefficienti è ancora sdp ma ha diagonale unitaria, ovvero del tipo $I - L - L^T$. Osservare quindi che, per ogni vettore reale $\underline{v}$ di norma 1, si ha: $\underline{v}^T L\underline{v} = \underline{v}^T L^T \underline{v} = \frac{1}{2}\underline{v}^T(L + L^T)\underline{v} < \frac{1}{2}$.)

Soluzione.
Scriviamo il sistema $A\underline{x} = \underline{b}$ nella forma equivalente

$$\left(D^{-1/2}AD^{-1/2}\right)\left(D^{1/2}\underline{x}\right) = \left(D^{-1/2}\underline{b}\right)$$

con $D = diag(\sqrt{a_{1,1}}, \ldots, \sqrt{a_{n,n}})$. La matrice $C = \left(D^{-1/2}AD^{-1/2}\right)$ ha diagonale unitaria:

$$c_{i,i} = d_i^{-1}a_{i,i}d_i^{-1} = \frac{1}{\sqrt{a_{i,i}}}a_{i,i}\frac{1}{\sqrt{a_{i,i}}} = a_{i,i};$$

inoltre è ancora sdp e scrivibile come $C = I - L - L^T$.
Poiché C è sdp risulta $\underline{v}^T A\underline{v} > 0, \forall \underline{v} \neq \underline{0}$ cioè

$$\underline{v}^T\underline{v} > \underline{v}^T L\underline{v} + \underline{v}^T L^T\underline{v} \quad \Rightarrow \quad \underline{v}^T L\underline{v} = \underline{v}^T L^T\underline{v} < \frac{1}{2}.$$

Sia $|\lambda| = \rho(M_{GS}^{-1}N_{GS}) = \rho\left((I-L)^{-1}L^T\right)$ assunto reale e $\underline{v}$ il corrispondente autovettore, dunque

$$(I-L)^{-1}L^T = \lambda\underline{v} \quad \Rightarrow \quad \lambda\underline{v} = L^T\underline{v} + \lambda L\underline{v}$$

ovvero $\lambda = \underline{v}^T L\underline{v} + \lambda\underline{v}^T L\underline{v} = (1+\lambda)\underline{v}^T L)\underline{v}$ da cui

$$\frac{\lambda}{1+\lambda} = \underline{v}^T L\underline{v} < \frac{1}{2} \quad \Rightarrow \quad -1 < \lambda < 1.$$

Segue $\rho(M_{GS}^{-1}N_{GS}) = |\lambda| < 1$.

Esercizio 6.10. *Con riferimento ai vettori errore (6.16) e residuo (6.17) dimostrare che, se*

$$||\underline{r_k}|| \leq \varepsilon||\underline{b}||, \tag{6.19}$$

allora

$$||\underline{e_k}|| \leq \varepsilon k(A)||\underline{\hat{x}}||,$$

dove $k(A)$ denota, al solito, il numero di condizionamento della matrice A. Concludere che, per sistemi lineari malcondizionati, anche la risoluzione iterativa (al pari di quella diretta) risulta essere più problematica.

Soluzione.
Posto $\underline{e}_k = \underline{x}_k - \hat{\underline{x}}$ e $\underline{r}_k = A\underline{x}_k - \underline{b}$, risulta

$$A\underline{e}_k = A(\underline{x}_k - \hat{\underline{x}}) = A\underline{x}_k - A\hat{\underline{x}} = A\underline{x}_k - b = \underline{r}_k.$$

Segue, passando alle norme,

$$\begin{aligned} ||\underline{e}_k|| = \left|\left|A^{-1}\underline{r}_k\right|\right| \leq \left|\left|A^{-1}\right|\right| \cdot ||\underline{r}_k|| \leq \left|\left|A^{-1}\right|\right| \cdot \varepsilon||\underline{b}|| \leq \\ \leq \left|\left|A^{-1}\right|\right| \cdot ||A|| \cdot ||\hat{\underline{x}}|| = \varepsilon\kappa(A)||\hat{\underline{x}}||, \end{aligned} \tag{6.20}$$

ovvero

$$\frac{||\underline{e}_k||}{||\hat{\underline{x}}||} \leq \varepsilon\kappa(A).$$

La risoluzione iterativa, come la risoluzione diretta, di sistemi lineari è ben condizionata per $\kappa(A) \approx 1$ mentre risulta malcondizionata per $\kappa(A) \gg 1$.

Esercizio 6.11. *Calcolare il polinomio caratteristico della matrice*

$$\begin{pmatrix} 0 & \dots & 0 & \alpha \\ 1 & \ddots & & 0 \\ & \ddots & \ddots & \vdots \\ 0 & & 1 & 0 \end{pmatrix} \in \mathbb{R}^{n\times n}.$$

Soluzione.
Il polinomio caratteristico è dato del determinante della matrice $A - \lambda I$:

$$\begin{aligned} \det(A - \lambda I) = &\det \begin{pmatrix} -\lambda & \dots & 0 & \alpha \\ 1 & \ddots & & 0 \\ & \ddots & \ddots & \vdots \\ 0 & & 1 & -\lambda \end{pmatrix} = \\ &= (-1)^n\lambda^n + (-1)^{n+1}\alpha = (-1)^n(\lambda^n - \alpha). \end{aligned} \tag{6.21}$$

Le radici di tale polinomio sono $\lambda = \sqrt[n]{\alpha}$.

Esercizio 6.12. *Dimostrare che i metodi di Jacobi e Gauss-Seidel possono essere utilizzati per la risoluzione del sistema lineare (gli elementi non indicati sono da intendersi nulli)*

$$\begin{pmatrix} 1 & & & -\frac{1}{2} \\ -1 & 1 & & \\ & \ddots & \ddots & \\ & & -1 & 1 \end{pmatrix} \underline{x} = \begin{pmatrix} \frac{1}{2} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \in \mathbb{R}^n,$$

la cui soluzione è $\underline{x} = (1, \dots, 1)^T \in \mathbb{R}^n$. *Confrontare il numero di iterazioni richieste dai due metodi per soddisfare lo stesso criterio di arresto (6.19), per valori crescenti di* n *e per tolleranze* ε *decrescenti. Riportare i risultati ottenuti in una tabella* (n/ε).

Soluzione.
La matrice è una M-matrice:

$$A = \begin{pmatrix} 1 & & & -1/2 \\ -1 & 1 & & \\ & \ddots & \ddots & \\ & & -1 & 1 \end{pmatrix} = I - B \text{ con } B = \begin{pmatrix} 0 & & & 1/2 \\ 1 & \ddots & & \\ & \ddots & \ddots & \\ & & 1 & 0 \end{pmatrix} > 0.$$

Si dimostra che $\rho(B) < 1$ calcolando gli autovalori della matrice B:

$$\det(B - \lambda I) =$$

$$\det \begin{pmatrix} -\lambda & & & 1/2 \\ 1 & \ddots & & \\ & \ddots & \ddots & \\ & & 1 & -\lambda \end{pmatrix} = -\lambda \det \begin{pmatrix} -\lambda & & \\ 1 & \ddots & \\ & \ddots & \ddots \\ & & 1 & -\lambda \end{pmatrix}_{(n-1)\times(n-1)} +$$

$$+(-1)^{n+1}\frac{1}{2}\det \begin{pmatrix} 1 & -\lambda & & \\ & \ddots & \ddots & \\ & & \ddots & -\lambda \\ & & & 1 \end{pmatrix}_{(n-1)\times(n-1)} = -\lambda(-\lambda)^{n-1} + (-1)^{n+1}\frac{1}{2} =$$

$$= (-1)^{n-2}\lambda^n + (-1)^{n+1}\frac{1}{2} = (-1)^n\frac{1}{2}(2\lambda^n - 1).$$

Quindi $\det(B - \lambda I) = 0$ se e solo se $2\lambda^n - 1 = 0$ se e solo se $\lambda = 2^{-1/n}$. Poiché $|\lambda| < 1$, $\rho(B) < 1$ quindi A è una M-matrice ed è possibile risolvere il sistema lineare tramite i metodo iterativi di Jacobi e Gauss-Seidel in quando lo splitting è regolare ed A converge.
Implementando il criterio d'arresto $||\underline{r}_k|| \leq \varepsilon||\underline{b}||$, si ha convergenza nel numero di iterazioni riportate nelle seguenti tabelle:

Iterazioni del metodo di Jacobi

$n\backslash\varepsilon$	10^{-1}	10^{-2}	10^{-3}	10^{-14}	10^{-5}	10^{-6}	10^{-7}	10^{-8}	10^{-9}	10^{-10}
5	25	34	54	74	85	105	124	139	157	171
10	50	72	116	145	181	211	250	276	304	342
15	87	125	180	230	284	326	379	430	465	524
20	95	175	239	298	370	433	512	573	628	702
25	128	217	299	375	468	549	643	720	800	885
30	162	265	378	475	570	659	762	870	964	1066
35	199	311	436	533	662	775	905	1014	1120	1249
40	214	384	494	635	755	889	1037	1157	1299	1429
45	252	423	556	703	853	1020	1165	1327	1448	1607
50	299	458	622	787	963	1108	1290	1473	1630	1789

Iterazioni del metodo di Gauss-Seidel

$n\backslash\varepsilon$	10^{-1}	10^{-2}	10^{-3}	10^{-14}	10^{-5}	10^{-6}	10^{-7}	10^{-8}	10^{-9}	10^{-10}
5	3	7	10	13	17	20	22	26	30	33
10	1	6	10	13	16	20	23	26	27	33
15	4	6	9	12	17	19	21	27	27	33
20	2	5	10	12	15	20	23	27	28	33
25	4	4	10	12	16	20	24	23	28	33
30	1	7	7	13	17	19	23	27	29	33
35	2	7	10	11	17	19	23	26	30	32
40	1	7	9	12	17	20	18	27	30	25
45	1	3	9	13	15	20	23	27	30	32
50	2	3	10	7	15	19	23	27	27	31

Si nota come il numero di iterazioni richieste dal metodo di Gauss-Seidel sia molto minore del numero richiesto dal metodo di Jacobi, per ogni valore della tolleranza.

Riferimenti MATLAB
Codice 6.6 (pagina 223)

Codice degli esercizi

Codice 6.5: Esercizio 6.2.

```
% Esercizio 6.2
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:13 CET

format short

for n =5:5:50
    A=2* eye(n);
    for i=2:n
        A(i,i -1)= -1;
        A(i -1,i)= -1;
    end
    lambda = MetodoPotenze (A, 10^ -5);
    approx = 2*(1+ cos(pi /(n +1)));
    fprintf ('n = %d\ tlambda = %5.4 f\ tapprox = %5.4 f\ terr = %5.4 f
        \n', n, lambda , approx , abs(lambda - approx ));
end
```

Codice 6.6: Esercizio 6.12.

```
% Esercizio 6.12
```

```
%
% Autore: Tommaso Papini,
% Ultima modifica: 4 Novembre 2012, 11:13 CET

format short
fprintf ('\ nMETODO DI JACOBI \n')
for n =5:5:50
    tol =10^ -1;
    while (tol >10^ -10)
        A=eye(n);
        for i=2:n

        end
        A(1,n )= -1/2;
        b= zeros (n ,1);
        b (1)=1/2;
        [b,i]= Jacobi ( A, b, tol );
        fprintf ('n = %d \t tol = %5.4 e \t i = %d \n', n, tol , i);
        tol=tol /10;
    end
end
fprintf ('\ nMETODO DI GAUSS - SEIDEL \n')
for n =5:5:50
    tol =10^ -1;
    while (tol >10^ -10)
        A=eye(n);
        for i=2:n
            A(i,i -1)= -1;
        end
        A(1,n )= -1/2;
        b= zeros (n ,1);
        b (1)=1/2;
        [b,i]= GaussSeidel ( A, b, tol );
        fprintf ('n = %d \t tol = %5.4 e \t i = %d \n', n, tol , i);
        tol=tol /10;
    end
    fprintf ('\n')
end
```

Bibliografia

[1] Luigi Brugnano, Cecilia Magherini, and Alessandra Sestini. *Calcolo Numerico, seconda edizione ampliata e corretta.* Novembre 2011.