Tesina CSMN

A.A. 2021/2022

Elena Puggioni, matr. 65527
e.puggioni11@studenti.unica.com

Questo elaborato contiene la descrizione delle esercitazioni di laboratorio.

Esercitazione 1

Contiene due script molto semplici per esercitarsi con le funzionalità base di Matlab.
Lo script **radice.m** chiede in input all'utente un numero con un valore compreso tra 0 e 50. Se il numero inserito è valido, calcola la radice quadrata del numero e la stampa a video. Se il numero non è valido, stampa un messaggio di errore.

Lo script **menu.m** stampa 4 piatti e le calorie correlate ad essi. I nomi dei piatti sono memorizzati in un vettore di stringhe, e le calorie corrispondenti in un array di interi con stesso indice del nome del piatto. Lo script chiede in input all'utente un numero da 1 a 4 e stampa il piatto scelto e le sue calorie.

Esercitazione 2

Lo script **approx.m** prende in input tre numeri a, b, c e calcola

$d1 = (a + b) + c$

$d2 = a + (b + c)$

in un sistema in virgola mobile con 3 cifre significative. Poi stampa a video i risultati e gli errori relativi dei calcoli con le operazioni macchina.
Se la variabile *debug* è impostata a true, usa dei valori di default invece di chiederli all'utente.

Quando si esegue lo script con dei numeri in input casuali scelti dall'utente spesso non si nota nessuna differenza tra i calcoli normali e quelli con arrotondamento. Tuttavia scegliendo i numeri in input un un certo modo è possibile far aumentare di molto l'errore.

Dal libro a pagina 64, sulla somma di due numeri $x + y$:
Se x e y hanno lo stesso segno il condizionamento è pari a 1, ma il valore aumenta quando i due numeri hanno segno discorde. In particolare, sei il segno è discorde e il modulo è molto simile il condizionamento può crescere in maniera incontrollata. Si parla in questo caso di cancellazione. La cancellazione è però dannosa solo quando gli operandi sono affetti da errore, mentre porta a un risultato esatto quando si opera su numeri di macchina.

Quindi, nel primo caso usando i numeri non arrotondati, l'errore sarà nullo se i numeri utilizzati sono numeri di macchina. Quando si usano invece i numeri arrotondati, anche il risultato sarà soggetto a errore poiché i numeri utilizzati per i calcoli non sono numeri di macchina, e in particolare l'errore sarà molto alto se facendo le somme si hanno due numeri simili in modulo ma con segno discorde.

Esercitazione 3

Lo script **matprod.m** chiede in input all'utente una dimensione n, poi crea:

- una matrice quadrata A di dimensione n che contiene tutti gli elementi uguali a 0
- una matrice quadrata B di dimensione n che contiene tutti gli elementi uguali a 1
- un vettore colonna z di lunghezza n che contiene tutti gli elementi uguali a 2

Poi calcola $\underline{b} = A\underline{z}$ e $\underline{c} = \underline{z}^T A$. Nel calcolo di c, è necessario usare z trasposto per poter eseguire il prodotto, in modo che le dimensioni interne tra vettore e matrice siano uguali.
Il vettore b sarà un vettore colonna ($n \times 1$) e avrà sempre tutti gli elementi pari a zero, dato che viene moltiplicato per la matrice nulla A che ha elementi tutti uguali a zero. Il vettore c sarà un vettore riga di dimensione ($1 \times n$) e tutti i suoi elementi saranno pari a $2n$.

Lo script **vettnorm.m** chiede all'utente una dimensione n in input e crea una matrice quadrata R di dimensione n con valori compresi tra 0 e 1, estrae la diagonale di R e la salva in un vettore colonna x. Crea poi:

- una matrice diagonale D che ha gli elementi di x nella diagonale
- una matrice triangolare superiore U estraendo il triangolo superiore di R
- una matrice triangolare inferiore L estraendo il triangolo inferiore di R.

Infine stampa a video i controlli che verificano:

- che D sia una matrice diagonale utilizzando la funzione *isdiag*;
- che U sia una matrice triangolare superiore utilizzando la funzione *istriu*;
- che L sia una matrice triangolare inferiore utilizzando la funzione *istril*.

Lo script **eigmat.m** prende in input dall'utente una dimensione n, e poi crea una matrice S di dimensione n di elementi interi compresi tra 10 e 20. Poi con la funzione *issymmetric* di Matlab controlla che la matrice S sia simmetrica e se non lo è la simmetrizza utilizzando la formula

$$\frac{S + S^T}{2}$$

A questo punto prende la matrice S resa simmetrica e ne calcola gli autovalori usando la funzione *eig* di Matlab, e li inserisce in un vettore d.
Infine calcola la norma 1, la norma 2, e la norma infinito del vettore d utilizzando la funzione *norm* di Matlab, e stampa i risultati.

Esercitazione 4

Lo script **test_gauss_lu.m** risolve un sistema lineare tramite la fattorizzazione $A = LU$, chiamando la funzione *gauss_lu.m*. Lo script crea 10 matrici di dimensione crescente tra 100 e 1000 con passo 100. Per testare la risoluzione dei sistemi e trovare l'errore relativo, si impone una soluzione x in modo che sia un vettore con tutti gli elementi pari a 1; si sceglie questa soluzione perché gli elementi sono uniformi e non sono né molto piccoli né molto grandi, e si può quindi minimizzare l'errore sulla soluzione. Si risolve poi il sistema con la soluzione scelta, eseguendo A*x e ottenendo il vettore dei termini noti b. A questo punto, si fa finta di non conoscere la soluzione x, e usando il vettore b calcolato in precedenza, si ricalcola la soluzione attraverso la fattorizzazione $A = LU$, risolvendo il sistema

$$\begin{cases} \mathrm{L}\underline{\mathrm{y}}=\underline{\mathrm{b}} \\ \mathrm{U}\underline{\mathrm{x}}=\underline{\mathrm{y}} \end{cases}$$

Lo script **gauss_lu.m** esegue la fattorizzazione di A scomponendola nelle matrici L e U; U è la matrice triangolare superiore che si ottiene applicando l'algoritmo di Gauss ad A, e L è la matrice triangolare inferiore costruita con i moltiplicatori dell'algoritmo di Gauss. Si risolve poi il sistema riportato sopra per trovare la soluzione. Infine si calcola l'errore relativo tra la soluzione che ottenuta e quella imposta all'inizio, usando l'apposita formula

$$\rho = \frac{\|\underline{x} - \underline{x}^*\|}{\|\underline{x}\|}$$

dove $\underline{x}$ è la soluzione originale e $\underline{x}^*$ è la soluzione perturbata. Si calcola poi il condizionamento della matrice A usando la funzione *cond* di Matlab. Questa operazione viene ripetuta per tutte le matrici generate.

Lo script **test_gauss_palu.m** è uguale al precedente, con l'unica differenza che la fattorizzazione usata è la fattorizzazione $PA = LU$.
La fattorizzazione $PA = LU$ utilizza l'algoritmo di Gauss con pivoting, infatti P è la matrice di permutazione che tiene traccia degli scambi; è una matrice identità con le righe scambiate secondo gli scambi che sono stati effettuati applicando l'algoritmo di Gauss, nello stesso ordine in cui sono stati effettuati. L'unica differenza nella risoluzione del sistema lineare tra la fattorizzazione $A = LU$ e $PA = LU$ è che nella fattorizzazione $PA = LU$ si moltiplica nella prima equazione il vettore b per la matrice di permutazione P in modo che il vettore dei termini noti abbia gli stessi scambi che sono stati effettuati durante la fattorizzazione della matrice A.

Lo script **gs.m** risolve il sistema lineare associato ad $A\underline{x} = b$ applicando l'algoritmo di Gauss-Seidel.
L'algoritmo di Gauss-Seidel è un algoritmo iterativo, ovvero esegue un certo numero di iterazioni k che approssima sempre di più la soluzione vera del sistema.
I metodi iterativi lineari stazionari del primo ordine utilizzano la formula

$$\underline{x}^{(k+1)} = B\underline{x}^{(k)} + \underline{f}$$

per risolvere il sistema, che viene applicata per un certo numero di passi k, e ogni passo risulta in un'approssimazione più accurata della soluzione $\underline{x}$. In questo tipo di metodi iterativi il passo k+1

che viene applicato utilizza come unico parametro che varia la soluzione $\underline{x}^{(k)}$ trovata al passo precedente.

Se l'algoritmo converge, maggiore è il numero di iterazioni, più accurata sarà la soluzione. La definizione di convergenza dell'algoritmo è

$$\lim_{k\to\infty} \left\| \underline{e}^{(k)} \right\| = 0$$

ovvero, più si aumenta il numero di iterazioni più l'errore sulla soluzione tende a zero. Si può verificare se l'algoritmo converge per una matrice A in diversi modi. Nel caso dell'algoritmo di Gauss-Seidel si può controllare che la matrice A sia a predominanza diagonale stretta, oppure che la matrice A sia simmetrica definita positiva. Queste condizioni sono condizioni sufficienti; una condizione invece necessaria e sufficiente, che quindi determina sempre se il metodo converge o no, per verificare la convergenza dell'algoritmo di Gauss-Seidel e anche dell'algoritmo di Jacobi, è che il raggio spettrale della matrice di iterazione B, sia tale che $1 \geq \rho(B) + \varepsilon$.

Più il raggio spettrale è minore di uno maggiore è la convergenza; se il raggio spettrale di B è molto vicino a uno invece l'algoritmo non converge bene e quindi saranno necessarie molte iterazioni per arrivare a una buona approssimazione della soluzione.

L'algoritmo di Gauss-Seidel, così come l'algoritmo di Jacobi, utilizza una fattorizzazione sulla matrice A chiamata *splitting additivo*, dove si scrive A nella sua scomposizione $A = P - N$.

In Gauss-Seidel, si scompongono poi le matrici P ed N come $P = D - E$, e $N = F$, dove:
- La matrice D è la matrice diagonale che contiene gli elementi diagonali di A;
- La matrice E è una matrice triangolare inferiore senza diagonale che contiene gli opposti degli elementi di A nel triangolo inferiore;
- La matrice F è una matrice triangolare superiore senza diagonale che contiene gli opposti degli elementi di A nel triangolo superiore.

Si costruisce poi la matrice di iterazione B

$$B = (D - E)^{-1} F$$

e il vettore di iterazione f

$$\underline{f} = (D - E)^{-1} \underline{b}$$

Avendo calcolato la matrice di iterazione e il vettore di iterazione si possono applicare i passi del metodo iterativo.
I criteri di arresto sono il raggiungimento di un certo numero di iterazioni massimo kmax, oppure arrivare ad una approssimazione della soluzione sotto una certa tolleranza tol; infatti, se un metodo iterativo converge allora soddisfa il criterio di Cauchy, e il criterio di arresto sarà

$$\left\| \underline{x}^{(k)} - \underline{x}^{(k-1)} \right\| \leq \tau \left\| \underline{x}^{(k)} \right\|$$

dove τ è la tolleranza. Quindi continuando ad approssimare la soluzione si arriverà eventualmente ad avere una soluzione con un errore inferiore a una certa tolleranza τ.

Lo script **test_metodi_iter.m** effettua i test sulla risoluzione di sistemi lineari con l'algoritmo di Jacobi e Gauss-Seidel. Il test crea 10 matrici con dimensione crescente tra 100 e 1000 con passo 100 in modo che siano strettamente diagonalmente dominanti. Poi, come negli script precedenti, si impone una soluzione x che è un vettore con tutti gli elementi uguali a 1, e si calcola il vettore dei termini noti b.
Facendo finta di non conoscere la soluzione x si applicano gli algoritmi di Jacobi e Gauss Seidel, entrambi sulla stessa matrice, e dato che le matrici generate sono tutte strettamente diagonalmente dominanti gli algoritmi di Jacobi e Gauss-Seidel convergeranno sempre. Il parametro k è il fattore di predominanza, che indica quanto è diagonalmente dominante la matrice A. Più k è alto più la matrice è diagonalmente dominante, e quindi i metodi iterativi convergeranno più in fretta. Quando si imposta k=1, la matrice A è poco diagonalmente dominante, e infatti la convergenza è molto lenta.
Eseguendo lo script con k=2, tol=1e-08, kmax=200, si ottengono i seguenti risultati:

Metodo di Jacobi

dim	n.iter	errore	r.spettrale(B)
100	29	1.8626e-09	0.5
200	29	1.8626e-09	0.5
300	29	1.8626e-09	0.5
400	29	1.8626e-09	0.5
500	29	1.8626e-09	0.5
600	29	1.8626e-09	0.5
700	29	1.8626e-09	0.5
800	29	1.8626e-09	0.5
900	29	1.8626e-09	0.5
1000	29	1.8626e-09	0.5

Metodo di Gauss-Seidel

dim	n.iter	errore	r.spettrale(B)
100	10	1.2138e-09	0.10679
200	10	1.1803e-09	0.10773
300	10	1.1695e-09	0.1067
400	10	1.1516e-09	0.10698
500	10	1.197e-09	0.10725
600	10	1.1895e-09	0.10723
700	10	1.1604e-09	0.10698
800	10	1.133e-09	0.10678
900	10	1.1541e-09	0.10711
1000	10	1.1841e-09	0.10699

Alcune osservazioni che si possono fare sui risultati è che come ci si aspetta il metodo di Gauss-Seidel converge più velocemente rispetto al metodo di Jacobi. Infatti con una fattore di predominanza k=2, in media il metodo di Jacobi richiede 29 iterazioni, mentre il metodo di Gauss-Seidel circa 10. Se si osserva il raggio spettrale della matrice B del metodo di Jacobi, questo è sempre circa 0.5, mentre in Gauss-Seidel il raggio spettrale è circa 0.1. Il fatto che il raggio spettrale nel metodo di Gauss-Seidel tenda a essere più piccolo spiega come mai Gauss-Seidel converga più velocemente.
Inoltre, l'errore relativo del metodo di Gauss Seidel tende anche a essere leggermente inferiore rispetto al metodo di Jacobi. Nonostante queste osservazioni siano a favore del metodo di Gauss-Seidel, il metodo di Jacobi ha il grande vantaggio di essere parallelizzabile, a differenza del metodo di Gauss-Seidel, quindi bisogna valutare quale si vuole usare in base alle proprie esigenze.

Esercitazione 5

Lo script **corde.m** implementa il metodo delle corde per la risoluzione di un'equazione non lineare. Prende in input un *function handle* f, un coefficiente angolare m, un punto iniziale x0, una tolleranza sulla soluzione tol, e un numero massimo di iterazioni kmax, e restituisce l'approssimazione sulla soluzione x e il numero di iterazioni effettuate k.

Il metodo di Newton è un metodo iterativo per la risoluzione di un'equazione non lineare. Approssima la soluzione calcolando la $f(x)$ in un punto iniziale x_0, e poi calcola la retta tangente a $f(x_0)$ e trova il punto x_1 di intersezione tra la retta tangente e l'asse delle ascisse. Questa sarà la nuova approssimazione sulla soluzione. Ripeterà poi il procedimento, valutando $f(x_1)$ e trovando la retta tangente a $f(x_1)$ e la sua intersezione con l'asse delle ascisse, e così via fino a che non si raggiunge uno dei criteri di arresto. La sua formula (geometrica) è

$$x_{k+1} = x_k - \frac{f(x)}{f'(x)}$$

Il metodo delle corde invece è uno dei metodi "quasi-Newton", che consistono nel considerare l'iterazione

$$x_{k+1} = x_k - \frac{f(x)}{m_k}$$

la formula è dunque simile al metodo di Newton, ma la derivata $f'(x)$ viene sostituita con un'approssimazione m_k. Questo è utile quando $f'(x)$ al denominatore si azzera, o quando $f(x)$ non è nota in modo analitico, o semplicemente quando si vuole risparmiare potenza di calcolo; infatti, il metodo di Newton ha un costo computazionale più alto rispetto ai metodi quasi-Newton perché richiede il calcolo della derivata.
Nel metodo delle corde, il coefficiente angolare m_k è costante, e si può scegliere valutando il grafico di $f(x)$ oppure calcolando $f'(x_0)$ e riutilizzandolo in tutti i passi. L'ordine di convergenza è lineare (p=1) e ha un basso costo computazionale, tuttavia le prestazioni non sono molto soddisfacenti.

Lo script **secanti.m** implementa il metodo delle secanti per la risoluzione di un'equazione non lineare. Prende in input un *function handle* f, due punti iniziali x0 e x1, una tolleranza sulla soluzione tol, e un numero massimo di iterazioni kmax, e restituisce l'approssimazione sulla soluzione x e il numero di iterazioni effettuate k.

Anche il metodo delle secanti è uno dei metodi quasi-Newton, e utilizza la stessa formula, ma utilizza il coefficiente angolare

$$m_k = \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$$

che è la retta secante la funzione in $f(x_k)$ e $f(x_{k-1})$.
Dato che questa formula dipende dalla valutazione di f su due punti precedenti, deve essere inizializzata con due punti x_0 e x_1. L'ordine di convergenza del metodo delle secanti è di $p = \frac{1+\sqrt{5}}{2}$, ovvero è il rapporto aureo, e a parità di costo computazionale produce una riduzione dell'errore maggiore, oltre a non richiedere il calcolo della derivata.

Lo script **test_nonlin.m** effettua dei test sulla risoluzione di un'equazione non lineare usando i dati presenti nelle tabelle 7.1, 7.2, 7.3 e 7.4 del libro di testo con i metodi di bisezione, Newton, corde, e secanti.

Tutti i metodi effettuano i test sulle stesse 4 funzioni per due volte a funzione con diversi punti iniziali. La tolleranza sulla soluzione è sempre 1e-08.
I criteri di arresto dei metodi implementati sono:

- la funzione approssima lo 0;
- si è raggiunto kmax iterazioni;
- l'errore tra la soluzione approssimata e la soluzione reale è sotto la tolleranza tol.

Lo script memorizza le funzioni f da analizzare, le vere soluzioni alfa, le derivate d delle funzioni, e i dati delle tabelle nelle opportune strutture dati. Poi itera attraverso i dati applicando i quattro metodi, salvando le soluzioni e il numero di iterazioni. Nel caso in cui vengano lanciate eccezioni le stampa a video. Infine raccoglie tutti i dati in tabelle e le stampa.

Metodo di bisezione

	alfa	x	[a	b]	| x - alfa |	n.iter
f(x) = (x.^2)-2	"sqrt(2)"	1.4142	0	2	1.8738e-09	29
	"sqrt(2)"	1.4142	0	200	1.1153e-11	32
f(x) = exp(x)-2	"log2"	0.69315	0	2	1.8206e-09	29
	"log2"	0.69315	0	200	2.1699e-09	36
f(x) = (1/x)-3	"1/3"	0.33333	0	2	1.2418e-09	29
	"1/3"	0.33333	0	200	6.5969e-10	36
f(x) = (x-3).^3	"3"	3.0003	1.3333	3.3333	0.00032552	11
	"3"	2.9996	1.3333	201.33	0.0004069	15

Il primo metodo implementato è il metodo di bisezione, ed è l'unico metodo che performa circa allo stesso modo per tutte le funzioni e in tutti i punti iniziali, perché non varia in nessun modo a seconda della funzione che sta risolvendo. Infatti il metodo di bisezione prende un intervallo iniziale $[a, b]$ all'interno del quale è contenuta la soluzione (ovvero il punto in cui $f(x) = 0$), e valuta $f(a)$ e $f(b)$ per scoprire il segno. Poi calcola il punto medio c dell'intervallo e valuta $f(c)$. Dopodiché sceglie un nuovo intervallo con c in uno degli estremi, e l'altro sarà a o b in modo che il segno con c sia discorde, che indica che la funzione si azzera nell'intervallo e quindi contiene la soluzione. Infine ripete il passo, dimezzando sempre di più l'intervallo. Per via della natura del processo, che ricorda quello della ricerca binaria, utilizza circa lo stesso numero di iterazioni per qualsiasi funzione. Anche partendo da un intervallo molto ampio il metodo eventualmente converge (con ordine di convergenza p=1), a differenza di altri metodi che potrebbero fallire se si sceglie un punto iniziale sbagliato. Per esempio nella prima funzione presa in esame quando si inizia con l'intervallo $[0, 2]$ sono necessarie 29 iterazioni, mentre quando si inizia con l'intervallo $[0, 200]$ sono necessarie 32 iterazioni, quindi solo 3 iterazioni in più, nonostante il secondo intervallo fosse considerevolmente più grande.

Metodo di Newton

	alfa	x	x0	| x - alfa |	n.iter
f(x) = (x.^2)-2	"sqrt(2)"	1.4142	2	1.5947e-12	4
	"sqrt(2)"	1.4142	200	7.4607e-13	11
f(x) = exp(x)-2	"log2"	0.69315	2	0	6
	"log2"	Inf	200	Inf	200
f(x) = (1/x)-3	"1/3"	NaN	2	NaN	0
	"1/3"	0.33333	0.1	5.5511e-17	7
f(x) = (x-3).^3	"3"	2.9995	2	0.00045109	19
	"3"	2.9997	2.9	0.00034255	14

Il secondo metodo esaminato è il metodo di Newton. Questo metodo è spesso più veloce rispetto al metodo di bisezione, infatti nel caso della prima funzione esaminata sono state necessarie 4 iterazioni con $x_0 = 2$, e 11 iterazioni con $x_0 = 200$, e ha restituito un errore significativamente più piccolo rispetto al metodo di bisezione. Tuttavia in alcuni casi, come per esempio nella seconda funzione con $x_0 = 200$, e nella terza funzione con $x_0 = 2$ non è riuscito a trovare la soluzione, perché la derivata si annulla al denominatore in alcuni punti. Scegliendo un punto iniziale diverso si è riusciti ad arrivare alla soluzione, però questo dimostra che il metodo di Newton non è sempre applicabile.

Metodo delle corde

	alfa	x	x0	| x - alfa |	n.iter
f(x) = (x.^2)-2	"sqrt(2)"	1.4142	2	2.003e-09	16
	"sqrt(2)"	1.4142	200	1.9786e-06	1749
f(x) = exp(x)-2	"log2"	0.69315	2	2.5455e-08	51
	"log2"	1	200	0.30685	1
f(x) = (1/x)-3	"1/3"	0	2	0.33333	0
	"1/3"	0.33333	0.1	9.4508e-08	190
f(x) = (x-3).^3	"3"	3.0273	2	0.027336	2000
	"3"	-Inf	2.9	Inf	6

Il metodo delle corde è quello con la performance peggiore di tutti, infatti in tutti i casi richiede più iterazioni del metodo di Newton e con un errore significativamente più alto, e riesce a battere il metodo di bisezione solo in un singolo caso. In tre casi non è riuscito a calcolare la soluzione, probabilmente perché il coefficiente angolare (che in questo script è la derivata della funzione calcolata in x_0) si annulla al denominatore. In due casi raggiunge o si avvicina al numero massimo di iterazioni.

Metodo delle secanti

	alfa	x	x0	| x - alfa |	n.iter
f(x) = (x.^2)-2	"sqrt(2)"	1.4142	1	2	0	6
	"sqrt(2)"	1.4142	199	200	2.2204e-16	16
f(x) = exp(x)-2	"log2"	0.69315	2	3	4.3407e-11	8
	"log2"	0.69315	199	200	1.0658e-14	293
f(x) = (1/x)-3	"1/3"	0	2	3	0.33333	0
	"1/3"	0.33333	0.1	0.11	4.996e-16	9
f(x) = (x-3).^3	"3"	2.9996	1	2	0.00041608	28
	"3"	2.9996	2.5	2.9	0.00042725	20

Il metodo delle secanti restituisce una performance piuttosto buona in quasi tutti i casi. In uno dei casi non è riuscito a determinare la soluzione, probabilmente anche in questo caso il coefficiente angolare si annulla al denominatore. In un solo caso ha raggiunto 293 iterazioni, pur trovando la soluzione, mentre in tutte le altre è riuscito a trovare la soluzione in relativamente poche iterazioni, in media meno del metodo di bisezione, ma più del metodo di Newton.
Nella maggior parte dei casi l'errore è anche piuttosto piccolo o addirittura nullo, come nel primo.

Esercitazione 6

Lo script **test_interp.m** effettua dei test sui metodi di interpolazione di funzioni, ovvero data una serie di punti con coordinate (x_i, y_1) si trova una funzione che passa per tutti quei punti, cercando di approssimare la vera funzione $f(x)$. Più punti si hanno a disposizione più la funzione interpolante sarà accurata. Il test utilizza l'interpolazione polinomiale, ovvero la funzione approssimata sarà sempre una funzione polinomiale, che dal teorema di Weierstrass sappiamo può sempre approssimare qualsiasi funzione continua, e usa due metodi di interpolazione: l'interpolazione attraverso la forma canonica e l'interpolazione con il polinomio di Lagrange.
La forma canonica utilizza la definizione

$$P_n(x) = \sum_{j=0}^{n} a_j x^i$$

per trovare i coefficienti a_j è necessario risolvere un sistema $X\underline{a} = \underline{y}$ con la matrice dei coefficienti X detta matrice di Vandermonde della forma

$$X = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ 1 & x_3 & x_3^2 & \dots & x_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \dots & x_m^{n-1} \end{bmatrix}$$

e con vettore di termini noti $\underline{y}$, composto dalle ascisse dei punti di interpolazione. Questo metodo ha tre svantaggi:

- la complessità computazionale per risolvere il sistema $X\underline{a} = \underline{y}$ è dell'ordine di $O(n^3)$, quindi piuttosto alta
- se alcuni nodi x_i sono molto vicini tra di loro la matrice X è molto mal condizionata
- piccole perturbazioni nei coefficienti a_j portano a grandi variazioni di nel polinomio trovato

Il polinomio interpolante di Lagrange usa un'altra strategia dove lo sforzo computazionale è incentrato nel trovare i polinomi di Lagrange L_j.
Il polinomio di Lagrange è definito

$$P_n(x) = \sum_{j=0}^{n} a_j L_j(x)$$

ed il singolo polinomio L_j è del tipo

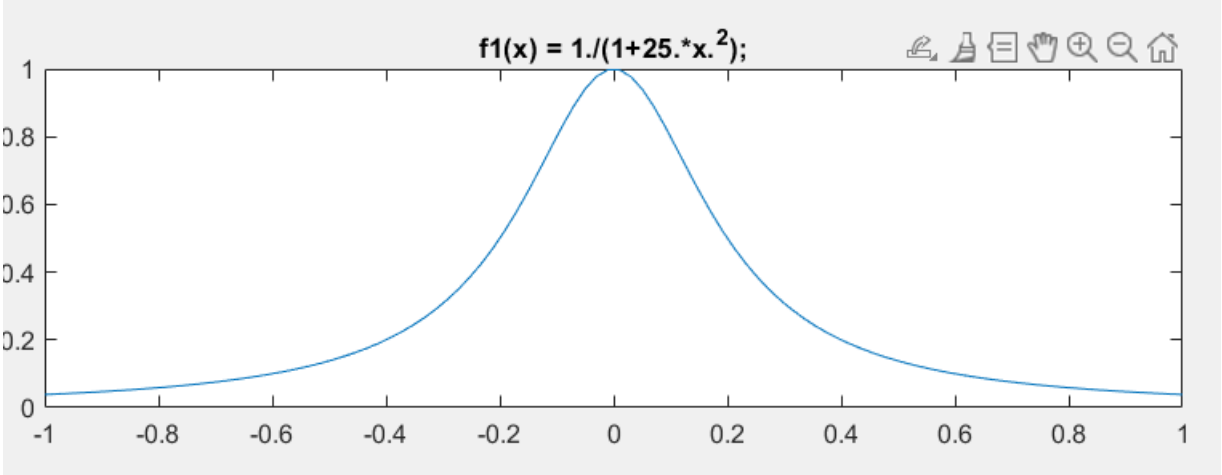
$$L_j(x) = \prod_{\substack{0 \le m \le k \\ m \neq j}} \frac{x - x_m}{x_j - x_m}$$

mentre i coefficienti sono semplicemente le ascisse dei punti già disponibili dai dati.

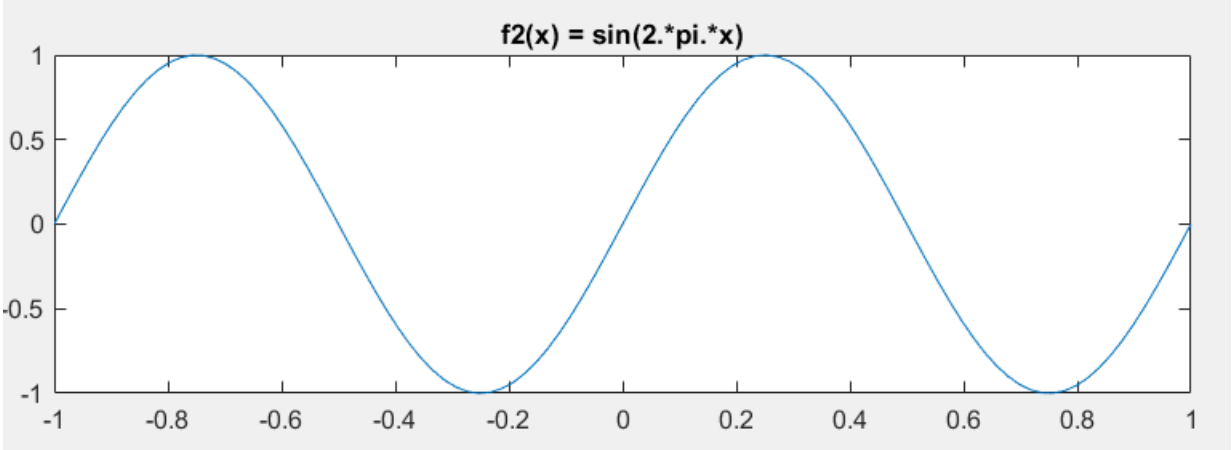
Il polinomio di Lagrange hanno complessità $O(n^2)$, quindi un ordine di grandezza inferiore rispetto alla forma canonica.

Lo script **test_interp.m** utilizza due funzioni di esempio

$$f_1(x) = \frac{1}{1+25x^2}$$



$$f_2(x) = sin(2\pi x)$$

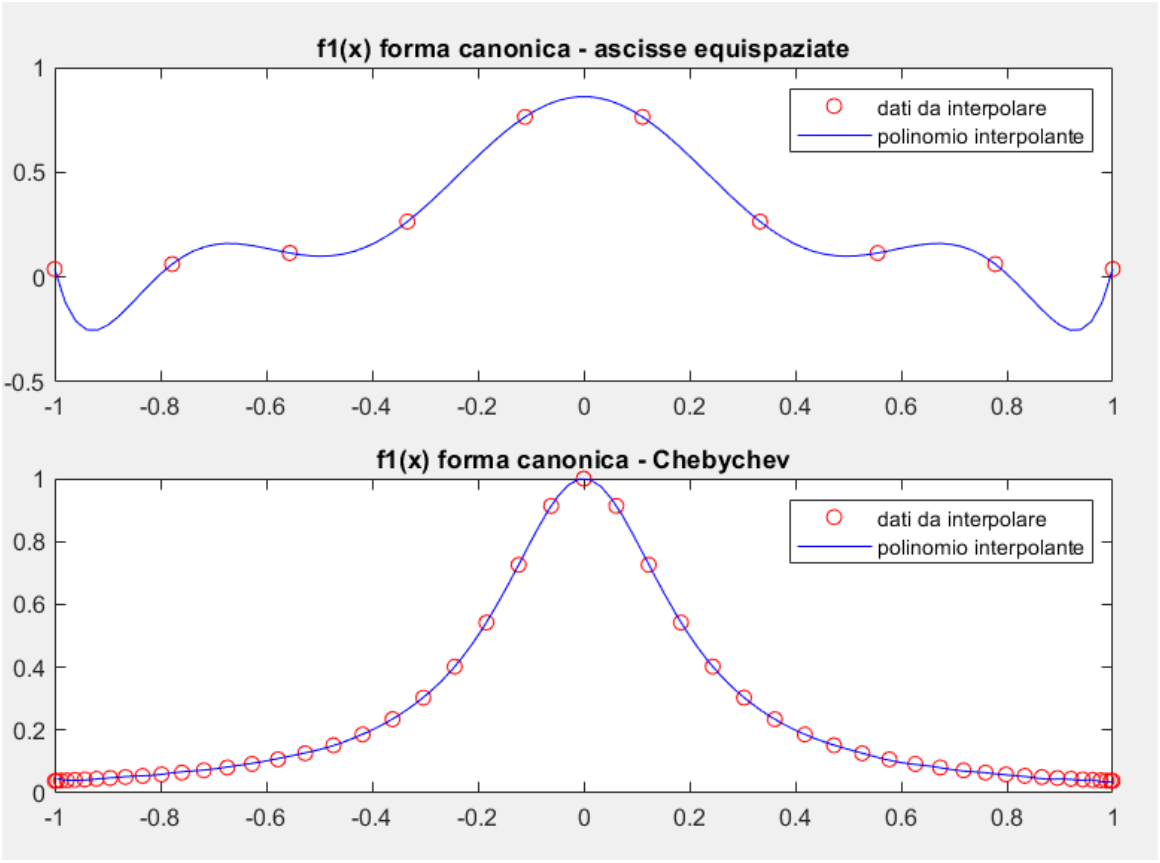


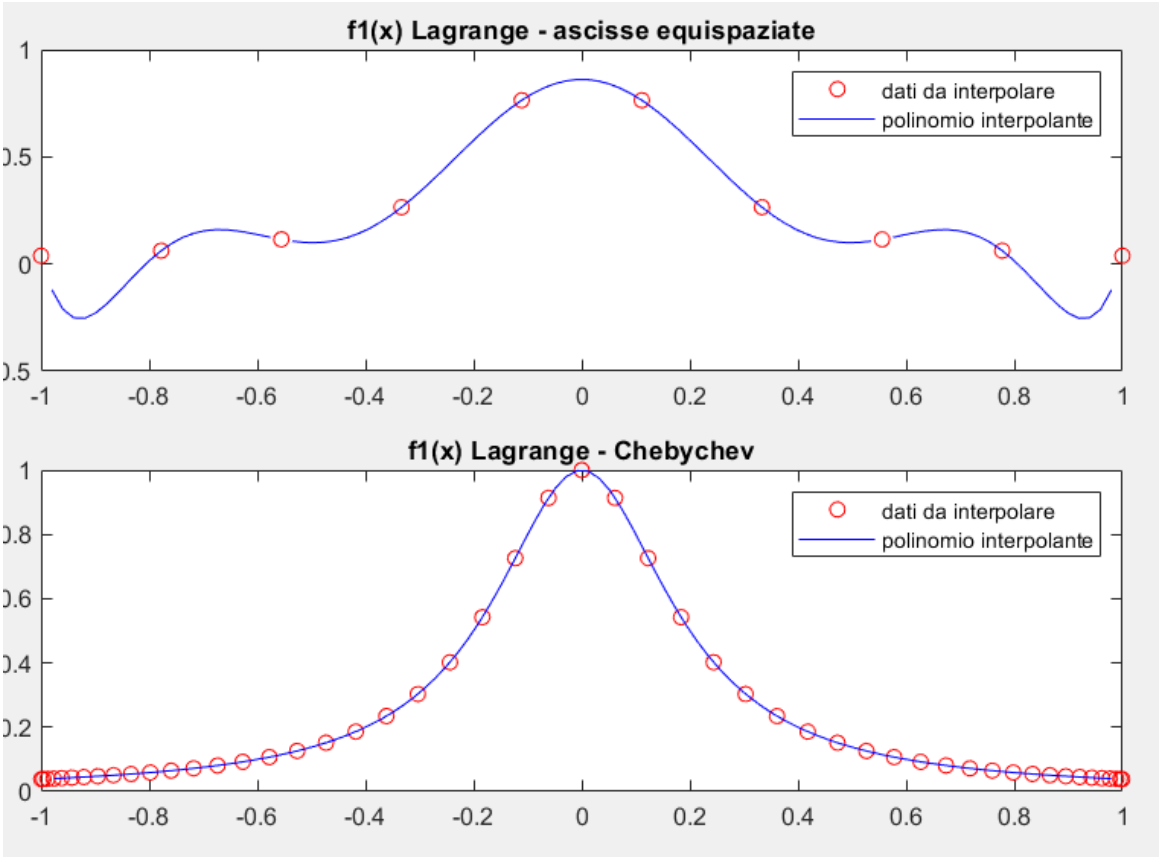
Genera dei punti di ascissa, e calcola le y valutando $f(x)$ in quei punti. Poi cerca una funzione che interpoli questi punti per testare quanto la funzione generata si avvicina alla funzione reale. Per testare anche la performance sull'errore si usano due distribuzioni di punti di ascisse per il campionamento della funzione.
L'errore dell'interpolazione è definito

$$E_n(x) = \frac{f^{n+1}(\xi_x)}{(n+1)!}\omega_n$$

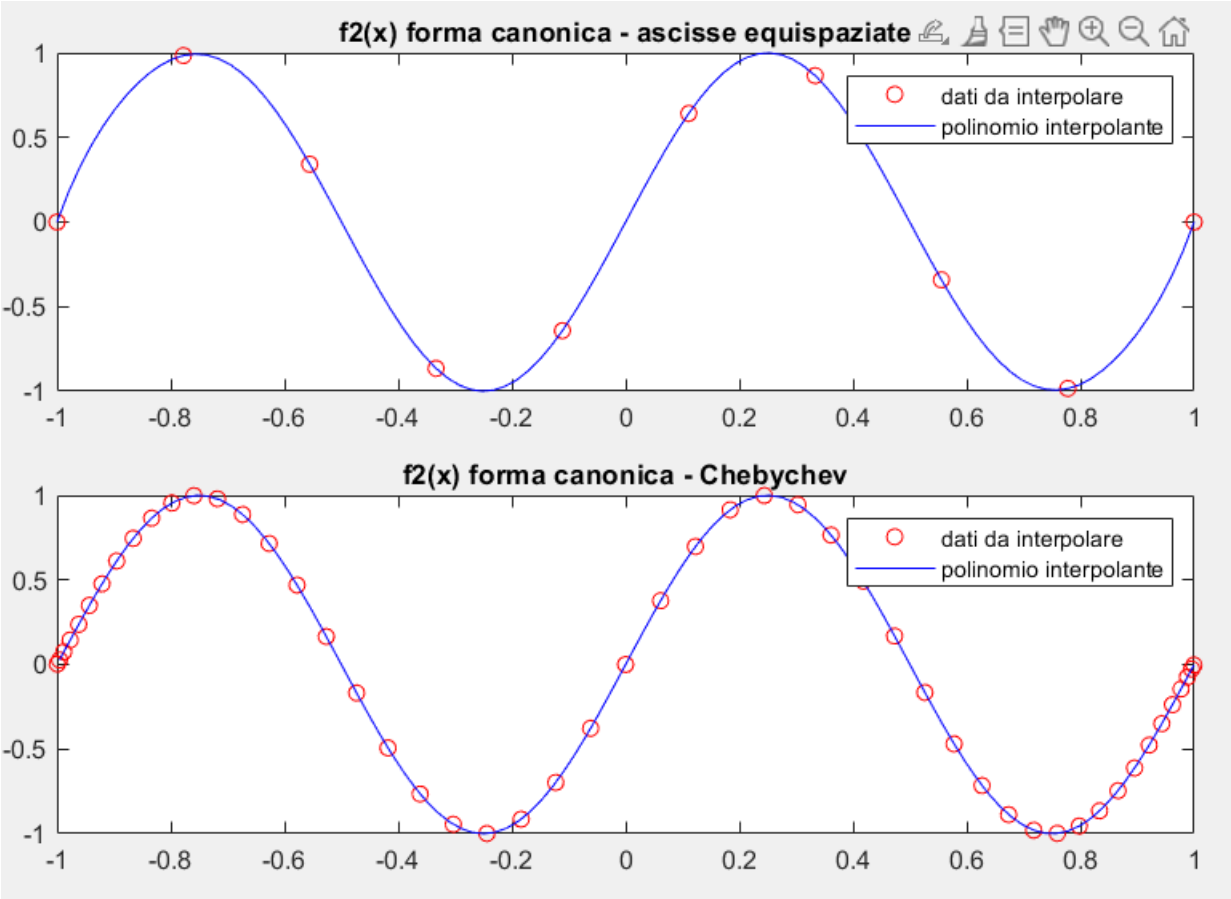
Osservando la formula si può vedere che se il numero di punti campione n è sufficientemente grande, il rapporto può essere reso piccolo a piacere, ma rimane la quantità ω_n che non dipende da n; dipende invece da come sono disposte le ascisse dei punti da interpolare.
Non esiste una regola universale per sceglierli, ma esiste il teorema di Bernstein, che dimostra che se $f(x)$ è continua in $[a, b]$, e le accise di interpolazione x_i sono gli zeri del polinomio di Chebychev di grado n+1, allora l'errore di interpolazione tende a 0 per $n \to Inf$.

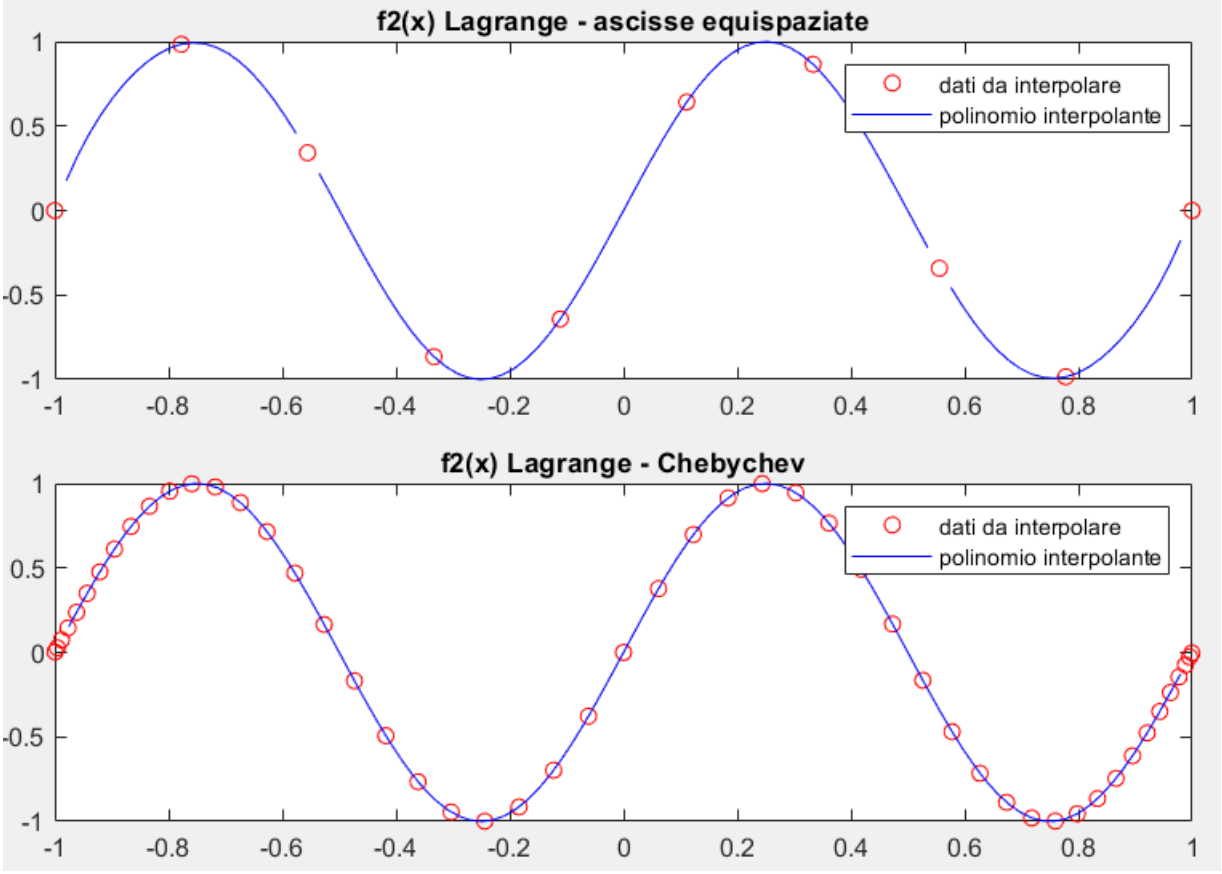
Lo script testa i metodi di interpolazione utilizzando sia le ascisse scelte in modo che siano equispaziate che le ascisse scelte a seconda del polinomio di Chebychev, ottenendo i seguenti risultati per $f_1(x)$





E questi risultati per $f_2(x)$





Come si può vedere, specialmente nella prima funzione, la versione con Chebychev approssima meglio la funzione, tanto che è praticamente identica alla funzione di partenza, mentre le ascisse equispaziate possono dare un risultato non accurato, e infatti Matlab lancia anche un warning sul fatto che la matrice che si utilizza con *canint.m* nella $f_1(x)$ sia molto mal condizionata. Nella seconda funzione invece si ottengono buoni risultati con tutti i metodi avvicinandosi parecchio alla funzione iniziale anche utilizzando le ascisse equispaziate.