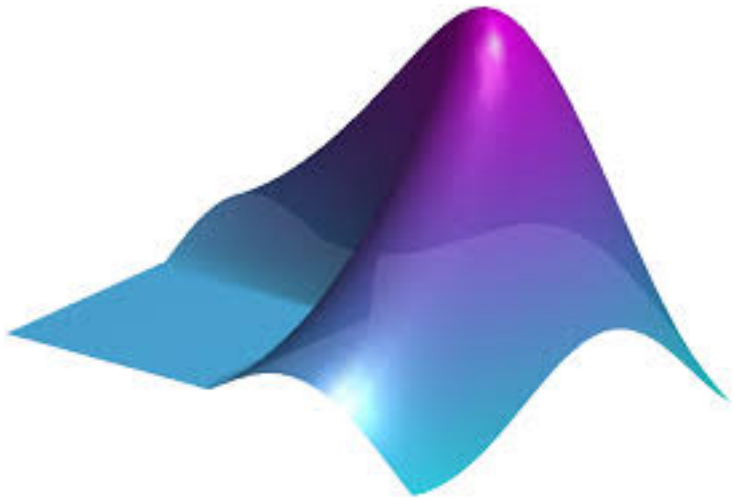
Università degli Studi di Firenze

Elaborato di Calcolo Numerico

Mohamed Salah Jebali - Denny Sbanchi
5968114 - 6336700



Anno Scolastico 2019/20

INDICE

1 ERRORI ED ARITMETICA FINITA 2
2 RADICI DI UN'EQUAZIONE 4
3 SISTEMI LINEARI E NON LINEARI 14
4 APPROSSIMAZIONE DI FUNZIONI 24
5 FORMULE DI QUADRATURA 35

1

ERRORI ED ARITMETICA FINITA

Esercizio 1.1. *Verificare che, per h sufficientemente piccolo,*

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

Soluzione. Supponendo che $f(x)$ sia sufficientemente regolare, con $h > 0$ sufficientemente piccolo, si ha:

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + \mathcal{O}(h^4) \qquad (1)$$

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + \mathcal{O}(h^4) \qquad (2)$$

Sommando membro a membro la (1) e la (2) si ottiene:

$$f(x-h)-2f(x)+f(x+h) = h^2f''(x)+\mathcal{O}(h^4) \qquad (3)$$

Dunque, dividendo la (3) per h^2, ricordando che per ipotesi $h > 0$, si ottiene:

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

Come volevasi dimostrare.

Esercizio 1.2. *Eseguire il seguente script Matlab, e spiegare cosa calcola*

```
u = 1; while 1, if 1+u==1, break, end, u = u/2; end, u
```

Soluzione. Il software MATLAB è costurito adottando lo standard IEEE 754 a doppia precisione con base b = 2, m = 53, e s = 11. Inoltre, la rappresentazione di un numero di macchina è effettuata per arrotondamento, pertanto la precisione di macchina si calcola come:

$$u = \frac{1}{2} b^{1-m}$$

Dunque, si può concludere osservando l'output restituito, che lo script calcola la suddetta precisione di macchina.

Esercizio 1.3. *Eseguire il seguente script Matlab:*

```
a = 1e20; b = 100; a-a+b
a = 1e20; b = 100; a+b-a
```

Spiegare i risultati ottenuti.

Soluzione. Andiamo ad analizzare i 2 casi:

- Nel primo caso il risultato ottenuto è **100**. Questo perché, **a - a** restituisce *zero*, essendo la sottrazione di due numeri identici, nonostante l'approssimazione che si ha per la rappresentazione dei numeri di macchina nello standard **IEEE754**.

- Invece, nel secondo caso il risultato ottenuto è **0**. Questo perché, la rappresentazione del numero di macchina associato ad **a** è soggetta ad errore di *round off* perché il numero **1e20** eccede il numero massimo rappresentabile senza errori nello standard **IEEE754**. Di conseguenza, la somma **a+b**, dato che **b = 100**, è soggetta all'errore di *round off* e la sua rappresentazione di macchina non differisce da quella di **1e20**.
  Per questo motivo, il risultato della somma tra a e b viene rappresentato con lo stesso numero di macchina di a. Dunque, quando vado a sottrarre a dalla somma $a + b$ ottengo *zero*.

Possiamo concludere osservando che, nelle operazioni di macchina, le normali proprietà di operazioni algebriche, come quella *commutativa*, non sono sempre garantite, in particolare quando si opera con numeri che eccedono la rappresentazione massima di macchina.

2

RADICI DI UN'EQUAZIONE

Esercizio 2.1. *Scrivere una function Matlab,* $\mathtt{radn}(x, n)$ *che, avendo in ingresso un numero positivo* x *ed un intero* n*, ne calcoli la radice* $n-$esima *con la massima precisione possibile.*

Soluzione. Di seguito è riportata la *function* Matlab richiesta per il calcolo della radice $n-$esima di un numero positivo x (dove n è un numero intero):

```
function radice = radn(x, n)
%
%   radice = radn(x,n) calcola la radice n-esima di un numero positivo x.
%   Il problema del calcolo della radice n-esima di x si riconduce a cerca
%   la radice della funzione f(x) = x^(1/n) - radice. In questo caso
%   stato adottato il metodo di Newton poich  il pi  efficiente tra quelli
%   studiati.
%
%   La function prende in input:
%   - x: numero positivo di cui si vuole calcolare la radice.
%   - n: numero intero che indica il grado della radice.
%
%   In output viene restituita la radice calcolata.
%

x_start = x;
x0 = x;
fx = f(x0,n,x_start);
derx = der(x0,n);
x = x0 - fx/derx;
while (abs(x-x0)>eps(x))
    x0 = x;
    fx = f(x0,n,x_start);
    derx = der(x0,n);
    x = x0- fx/derx;
end
radice = x;
end
```

```
function y = f(x, n, x_start)
y = x^n - x_start;
end

function y = der(x, n)
y = n * x ^ (n-1);
end
```

Esercizio 2.2. *Scrivere function Matlab distinte che implementino efficientemente i seguenti metodi per la ricerca degli zeri di una funzione:*

- *metodo di bisezione*
- *metodo di Newton*
- *metodo delle secanti*
- *metodo delle corde*

Detta x_i *l'approssimazione al passo i-esimo, utilizzare come criterio di arresto*

$$|x_{i+1} - x_i| \leqslant tol \cdot (1 + |x_i|),$$

essendo `tol` *una opportuna tolleranza specificata in ingresso.*

Soluzione. Di seguito vengono mostrate le implementazioni dei 4 codici richiesti:

- Metodo di Bisezione:

```
function [radice, iterations] = bisezione(func, a, b, tol)
%
%   radice = bisezione(func, a, b, tol)
%   calcola la radice di una funzione con il metodo di bisezione.
%
%   La function prende in input:
%   -func: funzione di cui calcolare la radice.
%   -a, b: estremi intervallo di confidenza iniziale.
%   -tol: tolleranza per criterio di arresto richiesto.
%
%   La function resituisce in output:
%   - [radice, iterations] = [radice, numero di iterazioni per ottenerla]
%

    fa=feval(func,a);
    imax= ceil( log2(b-a) - log2(tol) );
    x2= (a+b)/2;
    for iterations=1:imax
```

```
        x = x2;
        fx = feval(func,x);
        if fa*fx > 0
            a=x;
            fa=fx;
        else
            b=x;
        end
        x2= (a+b)/2;
        if abs(x2 - x) <= (tol * (1 + abs(x)))
            break;
        end
    end
    radice = x2;
    return
end
```

- Metodo di Newton:

```
function [radice, iterations] = newton(func, der, iMax, x0, tol)
%
%   radice = newton(func, der, iMax, x0, tol)
%   calcola la radice di una funzione con il metodo di Newton.
%
%   La function prende in input:
%   -func: funzione di cui calcolare la radice.
%   -der: derivata prima della funzione.
%   -iMax: numero di iterazioni massimo.
%   -x0: approssimazione iniziale.
%   -tol: tolleranza per criterio di arresto richiesto.
%
%   La function resituisce in output:
%   - [radice, iterations] = [radice, numero di iterazioni per ottenerla]
%

    iterations = 0;
    fx0=feval(func,x0);
    derx0=feval(der,x0);
    x=x0-(fx0/derx0);
    while (iterations < iMax) && (abs(x-x0) > (tol * (1 + abs(x0)) ) )
        iterations= iterations+1;
        x0=x;
        fx0= feval(func,x0);
        derx0= feval(der,x0);
        if derx0 == 0
            error('derivata nulla');
        end
        x=x0-(fx0/derx0);
    end
```

```
    if abs(x-x0) > (tol * (1 + abs(x0)))
         disp('Il metodo non converge');
    end
    radice = x;
    return
end
```

- Metodo delle Secanti:

```
function [radice, iterations] = secanti(func, iMax, x0, x, tol)
%
%   radice = secanti(func, iMax, x0, x, tol)
%   calcola la radice di una funzione con il metodo delle Secanti.
%
%   La function prende in input:
%   -func: funzione di cui calcolare la radice.
%   -iMax: numero di iterazioni massimo.
%   -x0,x: approssimazioni iniziali.
%   -tol: tolleranza per criterio di arresto richiesto
%
%   La function resituisce in output:
%   - [radice, iterations] = [radice, numero di iterazioni per ottenerla]
%

    fx0=feval(func,x0);
    fx = feval(func,x);
    x1 = (fx*x0 - fx0*x)/(fx - fx0);
    iterations = 1;
    while (iterations < iMax) && (abs(x1-x) > ( tol * (1 + abs(x)) ))
        x0 = x;
        x = x1;
        iterations= iterations+1;
        fx0 = fx;
        fx = feval(func,x);
        x1 = (fx*x0 - fx0*x)/(fx - fx0);
    end
    if abs(x1-x) > (tol * (1 + abs(x)))
         disp('Il metodo non converge');
    end
    radice = x1;
    return
end
```

- Metodo delle Corde:

```
function [radice, iterations] = corde(func, der, iMax, x0, tol)
%
%   radice = corde(func, der, iMax, x0, tol)
%   calcola la radice di una funzione con il metodo delle Corde.
%
%   La function prende in input:
%   -func: funzione di cui calcolare la radice.
%   -der: derivata prima della funzione.
%   -iMax: numero di iterazioni massimo.
%   -x0: approssimazione iniziale.
%   -tol: tolleranza per criterio di arresto richiesto.
%
%   La function resituisce in output:
%   - [radice, iterations] = [radice, numero di iterazioni per ottenerla]
%
    iterations = 0;
    fx0=feval(func,x0);
    derx0=feval(der,x0);
    x=x0-(fx0/derx0);
    while (iterations < iMax) && (abs(x-x0) > ( tol * (1 + abs(x0)) ) )
        iterations= iterations+1;
        x0=x;
        fx0= feval(func,x0);
        x=x0-(fx0/derx0);
    end
    if abs(x-x0) > (tol * (1 + abs(x0)))
         disp('Il metodo non converge');
    end
    radice = x;
    return
end
```

Esercizio 2.3. *Utilizzare le **function** del precedente esercizio per determinare una approssimazione della radice della funzione*

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

per $\mathtt{tol} = 10^{-3}, 10^{-6}, 10^{-9}, 10^{-12}$*, partendo da* $x_0 = 0$*.*
Per il metodo di bisezione, utilizzare [0,1], come intervallo di confidenza iniziale, mentre per il metodo delle secanti utilizzare le approssimazioni iniziali $x_0 = 0$ *e* $x_1 = 1$*.*
Tabulare i risultati, in modo da confrontare le iterazioni richieste da ciascun metodo. Commentare il relativo costo computazionale.

Soluzione. Tabulazione dei risultati ottenuti dall'esecuzione delle function degli esercizi precedenti su tolleranze sempre minori ($\mathtt{tol} = 10^{-3}, 10^{-6}, 10^{-9}, 10^{-12}$).

Bisezione

Con input **a = 0, b = 1, x0 = 0, f(x) = x - cos(x), tol**

Tolleranza	**Radice ottenuta**	**Numero di iterazioni**
$\mathtt{tol} = 10^{-3}$	x = 0.7392578125	9
$\mathtt{tol} = 10^{-6}$	x = 0.7390851974	19
$\mathtt{tol} = 10^{-9}$	x = 0.7390851332	29
$\mathtt{tol} = 10^{-12}$	x = 0.7390851332	39

Newton

Con input

f(x) = x - cos(x), f1(x) = 1 + sin(x), imax = 100, x0 = 0, tol

Tolleranza	**Radice ottenuta**	**Numero di iterazioni**
$\mathtt{tol} = 10^{-3}$	x = 0.7390851334	3
$\mathtt{tol} = 10^{-6}$	x = 0.7390851332	4
$\mathtt{tol} = 10^{-9}$	x = 0.7390851332	4
$\mathtt{tol} = 10^{-12}$	x = 0.7390851332	5

Secanti

Con input **f(x) = x - cos(x), imax = 100, x0 = 0, x = 1, tol**

Tolleranza	**Radice ottenuta**	**Numero di iterazioni**
$\mathtt{tol} = 10^{-3}$	x = 0.7390851121	4
$\mathtt{tol} = 10^{-6}$	x = 0.7390851332	5
$\mathtt{tol} = 10^{-9}$	x = 0.7390851332	6
$\mathtt{tol} = 10^{-12}$	x = 0.7390851332	6

Corde

Con input

f(x) = x - cos(x), f1(x) = 1 + sin(x), imax = 100, x0 = 0, tol

Tolleranza	Radice ottenuta	Numero di iterazioni
tol = 10^{-3}	x = 0.7395672022	16
tol = 10^{-6}	x = 0.7390845496	33
tol = 10^{-9}	x = 0.7390851327	51
tol = 10^{-12}	x = 0.7390851332	68

In termini computazionali è evidente che il metodo di Newton risulti il più performante, seguito subito dopo da una delle sue "varianti", ovvero il metodo delle Secanti. Negli altri due metodi è evidente che il numero di iterazioni richieste per ottenere una stessa approssimazione di una radice cresce molto rapidamente con il diminuire della tolleranza fornita. Per quanto riguarda il numero di operazioni algebriche (**flops**) eseguite abbiamo:
- Metodo di bisezione: $3i + 4$ flops
- Metodo di Newton: $4i + 4$ flops
- Metodo delle secanti: $6i + 7$ flops
- Metodo delle corde: $3i + 4$ flops

con i= numero di iterazioni eseguite.

Esercizio 2.4. *Calcolare la molteplicità della radice nulla della funzione*

$$f(x) = x^2 \cdot \tan(x)$$

Confrontare, quindi, i metodi di Newton, Newton modificato, e di Aitken, per approssimarla per gli stessi valori di tol del precedente esercizio (ed utilizzando il medesimo criterio di arresto), partendo da $x_0 = 1$. *Tabulare e commentare i risultati ottenuti.*

Soluzione. Una radice x^* dell'equazione $f(x) = 0$, con $x \in R$ ha molteplicità esatta $m \geqslant 1$, se:

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

$$f^m(x^*) \neq 0.$$

Nel nostro caso la radice x^* è la radice nulla, ovvero $x^* = 0$, dunque adesso procederemo con i calcoli per vedere fino a quale grado di derivazione la nostra $f(x)$ si annulla, con $x = 0$:

1.
$$f(x) = x^2 \cdot \tan(x)$$
Per $x = 0$ diventa
$$f(0) = 0^2 \cdot \tan(0) = 0$$

2.
$$f'(x) = 2 \cdot x \cdot \tan(x) + x^2 \cdot \frac{1}{\cos^2(x)}$$
Per $x = 0$ diventa
$$f'(0) = 0 \cdot 0 + \frac{0}{1} = 0$$

3.
$$f''(x) = 2 \cdot \tan(x) + \frac{4x}{\cos^2(x)} + 2x^2 \cdot \frac{\operatorname{sen}(x)}{\cos^3(x)}$$
Per $x = 0$ diventa
$$f''(0) = 2 \cdot \tan(0) + \frac{4 \cdot 0}{\cos^2(0)} + 2 \cdot 0^2 \cdot \frac{\operatorname{sen}(0)}{\cos^3(0)} = 0 + 0 + 0 = 0$$

4.
$$f'''(x) = \frac{2}{\cos^2(x)} + \frac{4\cos(x)2\sin(x)}{\cos^3(x)} + \frac{(4 \cdot x \cdot \operatorname{sen}(x) + 2 \cdot x^2 \cdot \cos(x)) \cdot \cos^3(x) + 6 \cdot x^2 \cdot \operatorname{sen}^2(x) \cdot \cos^2(x)}{\cos^6(x)}$$
Per $x = 0$ diventa
$$f'''(0) = \frac{2}{\cos^2(0)} + \frac{4\cos(0)2\sin(0)}{\cos^3(0)} + \frac{(4 \cdot 0 \cdot \operatorname{sen}(0) + 2 \cdot 0^2 \cdot \cos(0)) \cdot \cos^3(0) + 6 \cdot 0^2 \cdot \operatorname{sen}^2(0) \cdot \cos^2(0)}{\cos^6(0)},$$
ovvero
$$f'''(0) = 2 + 4 + 0 = 6 \neq 0.$$
Dunque, dato che
$$f(0) = f'(0) = f''(0) = 0,$$
$$f'''(0) \neq 0,$$
possiamo concludere che $m = 3$, ovvero la molteplicità della radice nulla è 3.

Di seguito vengono riportati i risultati tabulati ottenuti esegendo il calcolo delle radici con i metodi di Newton, Newton modificato e Aitken, utilizzando le tolleranze dell'esercizio precedente:

Newton

Con input

$f(x) = x^2 \cdot \tan(x)$, $f1(x) = 2 \cdot x \cdot \tan(x) + x^2 \cdot \frac{1}{\cos^2(x)}$, $imax = 100$, $x0 = 1$

Tolleranza	**Radice ottenuta**	**Numero di iterazioni**
$tol = 10^{-3}$	x = 0.0020	15
$tol = 10^{-6}$	x = 1.3492e-06	33
$tol = 10^{-9}$	x = 1.3694e-09	50
$tol = 10^{-12}$	x = 1.3899e-12	67

Newton Modificato

Con input

$f(x) = x^2 \cdot \tan(x)$, $f1(x) = 2 \cdot x \cdot \tan(x) + x^2 \cdot \frac{1}{\cos^2(x)}$,$m = 3$, $imax = 100$, $x0 = 1$

Tolleranza	**Radice ottenuta**	**Numero di iterazioni**
$tol = 10^{-3}$	x = 1.3235e-23	3
$tol = 10^{-6}$	x = 1.3235e-23	3
$tol = 10^{-9}$	x = 0	4
$tol = 10^{-12}$	x = 0	4

Aitken

Con input

$f(x) = x^2 \cdot \tan(x)$, $f1(x) = 2 \cdot x \cdot \tan(x) + x^2 \cdot \frac{1}{\cos^2(x)}$, $imax = 100$, $x0 = 1$

Tolleranza	**Radice ottenuta**	**Numero di iterazioni**
$tol = 10^{-3}$	x = 3.7260e-24	4
$tol = 10^{-6}$	x = 3.7260e-24	4
$tol = 10^{-9}$	x = 2.9358e-39	5
$tol = 10^{-12}$	x = 2.9358e-39	5

Come abbiamo visto nella prima parte della risoluzione, stiamo analizzando il caso di una radice multipla di una funzione, la cui molteplicità è nota.
Abbiamo dimostrato che, nel caso di radici multiple, il problema risulta essere mal condizionato e che, in particolare, il metodo di Newton converge solo linearmente, e questo risultato è empiricamente evidente nella prima tabella.

Il metodo di Newton Modificato, nota la molteplicità m della radice, è in grado di ripristinare la convergenza quadratica del metodo di Newton. Infatti, come si può osservare, già ad una tolleranza di 10^{-9}, in sole 4 iterazioni si raggiunge la radice esatta.
Per quanto riguarda il metodo di accelerazione di Aitken, i risultati ottenuti mostrano che nonostante la diminuzione della tolleranza il metodo si avvicina alla radice della funzione senza tuttavia calcolarla perfettamente (la approssima però con scarsissimo errore dopo sole 5 iterazioni).
Possiamo concludere che il metodo di Newton Modificato è quello che calcola la radice in modo più efficiente in questo caso.

3

SISTEMI LINEARI E NON LINEARI

Esercizio 3.1. *Scrivere una function Matlab che, data in ingresso una matrice* ***A****, restituisca una matrice,* ***LU****, che contenga l'informazione sui suoi fattori* ***L*** *ed* ***U****, ed un vettore* ***p*** *contenente la relativa permutazione, della fattorizzazione* ***LU*** *con* ***pivoting parziale di A****:*

function [LU, p] = palu(A)

Curare particolarmente la scrittura e l'efficienza della function.

Soluzione. Di seguito viene proposta un'implementazione della **function [LU, p] = palu(A)**:

```
function [LU,p] = palu(A)
%
%   [LU,p] = palu(A)
%   Calcola la fattorizzazione LU di una matrice quadrata A con a tecnica
%   del pivoting parziale, restituendo il vettore delle permutazioni
%   eseguite p.
%
%   La function prende in input:
%   - A: matrice quadrata non singolare, se la matrice passata non ha
%   questi requisiti viene rifiutata dalla funzione con messaggio di errore
%
%  In output vengono restituiti:
%  - LU: matrice contenente l'informazione sui fattori L ed U in cui
%  stata scomposta la matrice di partenza
%  - p: vettore contenente le permutazioni effettuate sulla matrice durante
%  la fattorizzazione con tecnica del pivoting parziale

    LU = A;
    [n,m] = size(A);
    if  (n~=m)
        error("Matrice non quadrata.");
    end
    for i=1:n-1
        if (LU(i,i) == 0)
```

```
            error("Matrice singolare, quindi non fattorizzabile LU.");
        end
    end


    p = 1:n;
    for i=1: n-1
        [mi,ki] = max(abs(LU(i:n,i)));
        if (mi==0)
            error("Matrice singolare, quindi non fattorizzabile LU.");
        end
        ki = ki + i - 1;
        if (ki>i)
            p([i,ki]) = p([ki,i]);
            LU([i,ki],:) = LU([ki,i],:);
        end
        LU(i+1:n,i) = LU(i+1:n,i)/LU(i,i);
        LU(i+1:n,i+1:n) = LU(i+1:n,i+1:n) - LU(i+1:n,i)*LU(i,i+1:n);
    end
    return
end
```

Esercizio 3.2. *Scrivere una function Matlab che, data in ingresso la matrice* ***LU*** *ed il vettore* ***p*** *creati dalla* ***function*** *del precedente esercizio, ed il termine noto del sistema lineare* ***Ax=b****, ne calcoli la soluzione:*

function x = lusolve(LU, p, b)

Curare particolarmente la scrittura e l'efficienza della function.

Soluzione. Di seguito viene proposta un'implementazione della **function x = lusolve(LU, p, b)**:

```
function x = lusolve(LU, p, b)
%
%   x = lusolve(LU, p, b)
%   Calcola la soluzione x del sistema lineare Ax=b in cui la matrice A
%   stata fattorizzata mediante l'utilizzo della fattorizzazione LU con
%   tecnica di pivoting parziale.
%
%   La function prende in input:
%   - LU: matrice contenente l'informazione sui fattori L ed U in cui
%  stata scomposta la matrice dei coefficienti del sistema lineare Ax=b
%   - p: vettore contenente le permutazioni effettuate sulle righe della
%   matrice fattorizzata
%   - b: termine noto del sistema lineare Ax=b
%
%  In output viene restituita:
```

```
%  - x: vettore contenente la soluzione del sistema lineare Ax=b
%

    [m,n] = size(LU);
    if ((m~=n) || (n~=length(b)) || (n~=length(p)))
        error('Dati inconsistenti.');
    elseif (min(abs(diag(LU)))== 0)
            error('Elemento sulla diagonale = 0.');
    end

    % Permuto gli elementi di b
    x = b(:);
    x = x(p);
    % Risolvo L
    for i=1: n-1
        x(i+1:n) = x(i+1:n)-LU(i+1:n, i) * x(i);
    end
    % Risolvo U
    for i=n: -1: 1
        x(i) = x(i)/LU(i,i);
        x(1:i-1) = x(1:i-1) - LU(1:i-1, i) * x(i);
    end
    return
end
```

Esercizio 3.3. *Scaricare la function cremat dal sito:*
http://web.math.unifi.it/users/brugnano/appoggio/linsis.m

che crea sistemi lineari $n \times n$ *la cui soluzione è il vettore* $x = (1 \dots n)^T$.
Eseguire, quindi, lo script Matlab:

```
n = 10;
xref = (1:10)';
for i = 1:10
    [A,b] = linsis(n,i);
    [LU,p] = palu(A);
    x = lusolve(LU,p,b);
    disp(norm(x-xref))
end
```

Tabulare in modo efficace, e spiegare in modo esauriente, i risultati ottenuti.

Soluzione. Risultati ottenuti

- i=1
  - x = [1.0000 2.0000 3.0000 4.0000 5.0000 6.0000 7.0000 8.0000 9.0000 10.0000]
  - Errore ottenuto = $2.0706e^{-14}$
  - Condizionamento di A = 10.0000
- i=2
  - x = [1.0000 2.0000 3.0000 4.0000 5.0000 6.0000 7.0000 8.0000 9.0000 10.0000]
  - Errore ottenuto = $1.0952e^{-14}$
  - Condizionamento di A = 50.0000
- i=3
  - x = [1.0000 2.0000 3.0000 4.0000 5.0000 6.0000 7.0000 8.0000 9.0000 10.0000]
  - Errore ottenuto = $5.2797e^{-13}$
  - Condizionamento di A = $5.0000e^{03}$
- i=4
  - x = [1.0000 2.0000 3.0000 4.0000 5.0000 6.0000 7.0000 8.0000 9.0000 10.0000]
  - Errore ottenuto = $6.3727e^{-11}$
  - Condizionamento di A = $5.0000e^{05}$

- i=5
  - x = [1.0000 2.0000 3.0000 4.0000 5.0000 6.0000 7.0000 8.0000 9.0000 10.0000]
  - Errore ottenuto = $1.2091e^{-08}$
  - Condizionamento di A = $5.0000e^{07}$
- i=6
  - x = [1.0000 2.0000 3.0000 4.0000 5.0000 6.0000 7.0000 8.0000 9.0000 10.0000]
  - Errore ottenuto = $1.4472e - 06e^{-06}$
  - Condizionamento di A = $5.0000e^{09}$
- i=7
  - x = [1.0000 2.0001 3.0000 4.0000 4.9999 5.9999 6.9999 8.0000 9.0000 10.0000]
  - Errore ottenuto = $1.3678e^{-04}$
  - Condizionamento di A = $5.0000e^{11}$
- i=8
  - x = [0.9936 2.0021 3.0034 4.0008 4.9981 5.9983 6.9978 7.9977 8.9948 10.0011]
  - Errore ottenuto = 0.0101
  - Condizionamento di A = $5.0015e^{13}$

- i=9
  - x = [1.1344 2.2555 2.8471 3.6337 4.8866 5.9212 7.0307 8.1849 8.99 9.7011]
  - Errore ottenuto = 0.6202
  - Condizionamento di A = $5.0998e^{15}$
- i=10
  - x = [0.3473 -0.192 0.320 0.304 -0.032 -0.096 0.448 1.024 -0.384 -0.128]
  - Errore ottenuto = $1.3267e^{03}$
  - Condizionamento di A = $7.1198e^{18}$

Dai risultati ottenuti possiamo notare che, con l'aumentare di **i**, l'errore commesso durante la risoluzione del sistema lineare aumenta, passando dall'ordine di $2e^{-14}$ a ben $1e^{03}$, nell'ultima iterazione.
L'aumento dell'errore relativo commesso sui dati di output **x**, della risoluzione del sistema lineare Ax=b, è giustificato dal malcondizionamento della matrice in input A, il cui numero di condizionamento aumenta all'aumentare di **i**.
Infatti, abbiamo monitorato l'aumento del numero di condizionamento della matrice A (**k(A)**), grazie alla function **cond** di Matlab che si è rivelata molto utile a comprendere le ragioni celate dietro all'aumento dell'errore su **x** all'aumentare di **i**.

Esercizio 3.4. *Scrivere una function Matlab che, data in ingresso una matrice* $A \in \mathbb{R}^{m \times n}$*, con* $m \geqslant n = \mathrm{rank}(A)$*, restituisca una matrice QR, che contenga l'informazione sui fattori Q ed R della fattorizzazione QR di A:*

```
function QR = myqr(A)
```

Curare particolarmente la scrittura e l'efficienza della function.

Soluzione. La function Matlab richiesta è:

```
function QR = myqr(A)
%
%   QR = myqr(A)
%   Data in ingresso una matrice A di numeri reali, la cui dimensione risulta
%   n x m con m >= n = rank(A), restituisca una matrice QR che contenga
%   l'informazione sui fattori Q ed R della fattorizzazione QR di A.
%

    [m,n] = size(A);

    % Controlli di integrit
    if n > m
        error('Dimensioni matrice non corrette');
    end

    QR = A;

    for i=1 : n
        alpha = norm(QR(i:m, i));

        if (alpha == 0)
            error('La matrice non    di rango massimo')
        end

        if (QR(i,i) >= 0)
            alpha = -alpha;
        end

        v1 = QR(i,i) - alpha;
        QR(i,i) = alpha;
        QR(i+1:m, i) = QR(i+1:m, i) / v1;
        beta = -v1/alpha;
        v = [1; QR(i+1:m, i)];
        QR(i:m,i+1:n) = QR(i:m, i+1:n) - (beta*v)*(v'*QR(i:m,i+1:n));
    end
    return
end
```

Esercizio 3.5. *Scrivere una function Matlab che, data in ingresso la matrice QR creata dalla function del precedente esercizio, ed il termine noto del sistema lineare* $Ax = b$*, ne calcoli la soluzione nel senso dei minimi quadrati:*

```
function x = qrsolve(QR,b)
```

Curare particolarmente la scrittura e l'efficienza della function.

Soluzione. La function Matlab richiesta è:

```
function x = qrsolve(QR, b)
%
%   x = qrsolve(QR, b)
%   Data in ingresso la matrice fattorizzata QR e il
%   termine noto b, risolve il sistema Ax = b nel senso dei minimi quadrati.
%

    [m,n] = size(QR);
    k = length(b);

    if k ~= m || n > m
        error('Dati Inconsistenti');
    end

    x = b(:);

    for i= 1 : n
        v = [1; QR(i+1:m, i)];
        beta = 2/(v' * v);
        x(i:m) = x(i:m) - (beta * (v' * x(i:m)))*v;
    end

    x = x(1:n);

    for i=n:-1:1
        if (QR(i,i) == 0)
            error('Sistema irrisolvibile');
        end

        x(i) = x(i)/QR(i,i);
        if i>1
            x(1:i-1) = x(1:i-1) - x(i) * QR(1:i-1, i);
        end
    end
    return
end
```

Esercizio 3.6. *Utilizzare le **function** scritte negli esercizi 11 e 12 per risolvere, nel senso dei minimi quadrati, il sistema lineare sovradeterminato definito dai seguenti dati:*
$A =$

$$\begin{matrix} 1 & 2 & 3 \\ 1 & 2 & 4 \\ 3 & 4 & 5 \\ 3 & 4 & 6 \\ 5 & 6 & 7 \end{matrix}$$

$b =$

$$\begin{matrix} 14 \\ 17 \\ 26 \\ 29 \\ 38 \end{matrix}$$

Soluzione. Utilizzando le **function** degli esercizi precedenti, prima **myqr** su A e dopo **qrsolve** sulla fattorizzazione QR di A e sul termine noto b, si ottiene il vettore soluzione:
x = [1.0000 2.0000 3.0000].

Esercizio 3.7. *Le seguenti istruzioni,*

```
A = rot90(vander(1:10)); A = A(:,1:8); x = (1:8)'; b = A*x;
```

creano una matrice $A \in R^{10x8}$ *di rango massimo ed un vettore* $b \in R^{10}$*, che definisce un sistema lineare la cui soluzione è data dal vettore*

$$x = (1, 2, 3, 4, 5, 6, 7, 8)^T.$$

Spiegare quindi qual è il significato delle espressioni Matlab:

```
A\b,          (A'*A)\(A'*b)
```

Spiegare i risultati ottenuti

Soluzione. In Matlab il comando $A\backslash b$ risolve il sistema lineare $b = A * x$. Osservando che la $A \in R^{10x8}$, con $m > n = rank(A)$ ha rango massimo per ipotesi e che $b = R^{10}$ e $x = R^8$, stiamo trattando il caso di sistemi lineari sovradeterminati.
Dunque, possiamo affermare che nel primo caso, $A\backslash b$ risolve il sistema lineare $b = A * x$ nel senso dei minimi quadrati.

Nel secondo caso, invece, il sistema lineare viene risolto con il metodo delle **equazioni normali**, il quale prevede di moltiplicare entrambi i membri dell'equazione del sistema lineare per la trasposta della matrice A e dividere. Questo metodo è stato dimostrato essere **mal condizionato**, a lezione. Infatti, Matlab, dopo l'esecuzione, lancia un segnale di "warning" che avvisa del fatto che la matrice è mal condizionata o vicino all'essere singolare.

4

APPROSSIMAZIONE DI FUNZIONI

Esercizio 4.1. *Approssimare la funzione* $f(x) = \cos(\pi\frac{x^2}{2})$ *con i polinomi interpolanti rispettivamente costruiti con n+1 ascisse equidistanti e con n+1 ascisse di Chebyshev sull'intervallo [-1,1]. Graficare (in formato* ***semilogy****) il massimo errore di interpolazione,* $n = 1, \ldots, 40$*. Commentare i risultati ottenuti.*

Soluzione. Di seguito sono riportati i grafici del massimo errore di interpolazione commesso utilizzando il polinomio interpolante costruito su $n+1$ ascisse equidistani Fig. 1 e su $n+1$ ascisse di Chebyshev sull'intervallo [-1,1] Fig. 2.
Abbiamo scelto di usare il polinomio nella forma di Lagrange come polinomio di interpolazione della funzione.

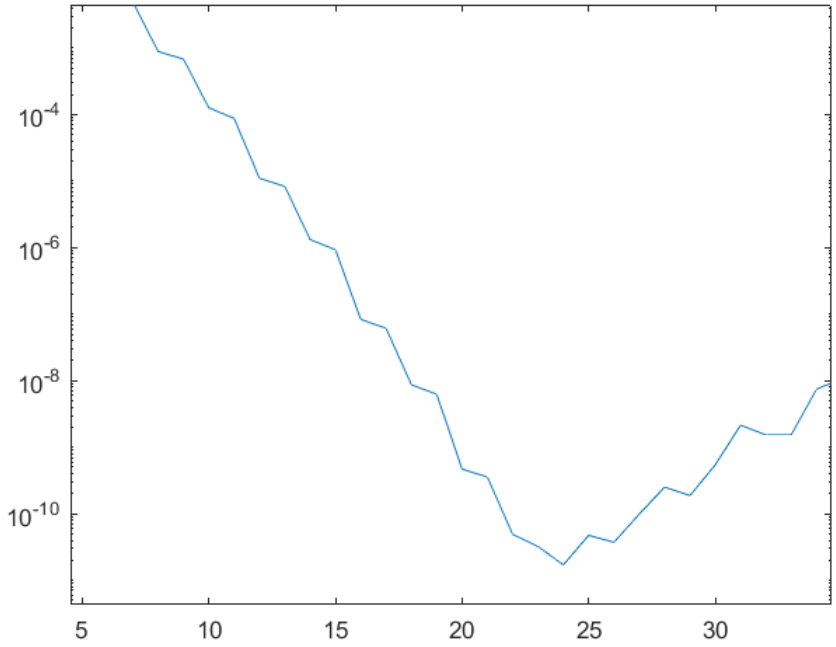


Figura 1: Grafico massimo errore su ascisse equidistanti. Asse x: n. Asse y: Massimo errore di interpolazione.

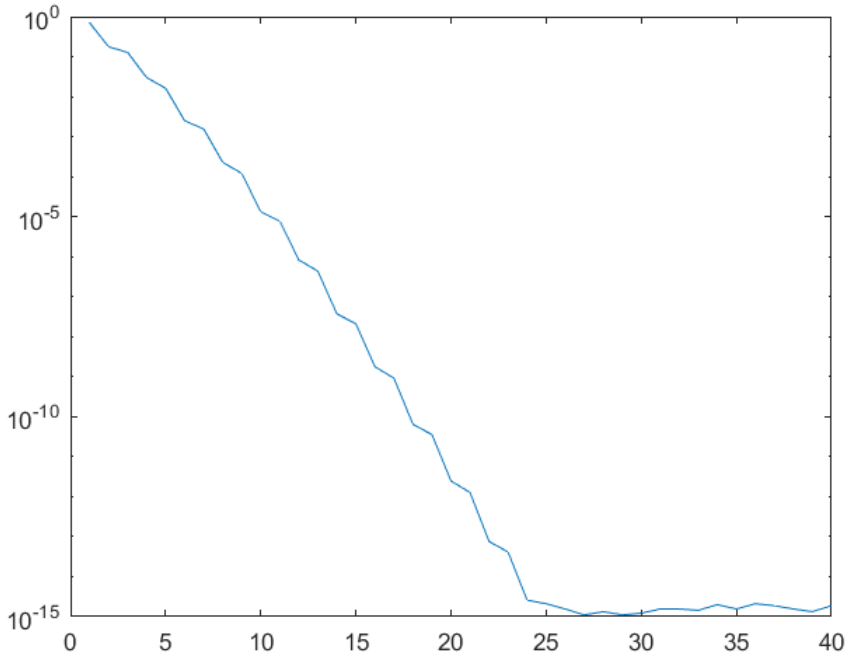


Figura 2: Grafico massimo errore su ascisse di Chebyshev. Asse x: n. Asse y: Massimo errore di interpolazione.

Osserviamo, innanzitutto, che l'errore di interpolazione mostrato nei grafici in alto dipende dal numero di condizionamento del problema dell'interpolazione polinomiale, il quale è strettamente connesso alla costante di Lebesgue che, per ascisse equidistanti, tende esponenzialmente verso ∞ al crescere di n.
Preliminarmente, possiamo osservare che in entrambi i grafici si ha una decrescita del massimo errore commesso durante l'interpolazione. Questa decrescita iniziale è dovuta all'aumentare del numero di ascisse di interpolazione, che comporta, inizialmente, una maggiore precisione nell'approssimazione della funzione da parte del polinomio.
Tuttavia, si osserva subito che, da un certo numero di ascisse in poi, circa $n = 24$, nel primo caso Fig. 1, l'errore inizia a crescere esponenzialmente, proprio per il motivo che abbiamo introdotto inizialmente, ovvero: al crescere del numero di ascisse di interpolazione, nel caso di ascisse equidistanti, cresce la costante di Lebesgue in modo esponenziale, e con essa l'errore.
Invece, com'è lecito aspettarsi, nel caso di ascisse di Chebyshev Fig. 2, la crescita dell'errore commesso è lentissima, per non dire assente, perché nel caso in cui vengano utilizzate le ascisse di Chebyshev, la costante di Lebesgue ha una crescita ottimale, di ordine logaritmico.

Esercizio 4.2. *Approssimare la funzione* $f(x) = \cos(\pi\frac{x^2}{2})$ *con i polinomi interpolanti di Hermite rispettivamente costruiti con n+1 ascisse equidistanti e con n+1 ascisse di Chebyshev sull'intervallo [-1,1]. Graficare (in formato* ***semilogy****) il massimo errore di interpolazione,* $n = 1, \ldots, 20$. *Commentare i risultati ottenuti.*

Soluzione. Di seguito sono riportati i grafici dell'errore di interpolazione commesso utilizzando il polinomio interpolante di Hermite di grado 2n+1.
Nella Fig. 3 viene rappresentato il grafico del massimo errore ottenuto su n+1 ascisse di interpolazione equidistanti, mentre in Fig. 4 viene rappresentato il grafico del massimo errore ottenuto su n+1 ascisse di Chebyshev appartenenti a [-1, 1].

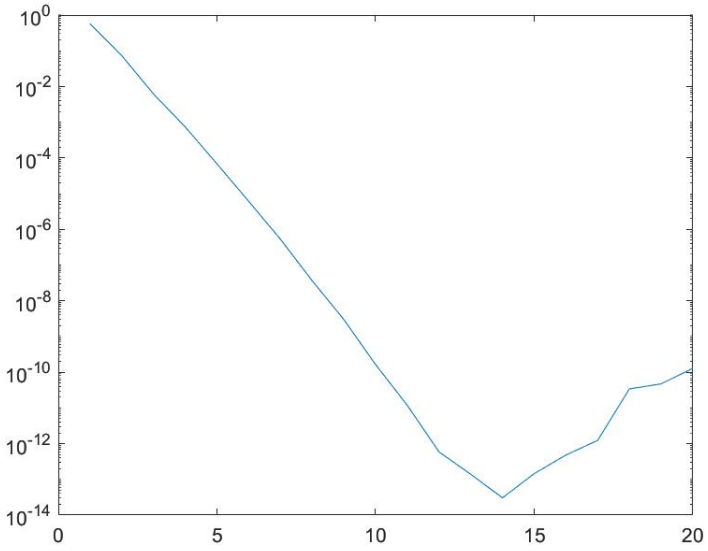


Figura 3: Grafico massimo errore su ascisse equidistanti. Asse x: n. Asse y: Massimo errore di interpolazione.

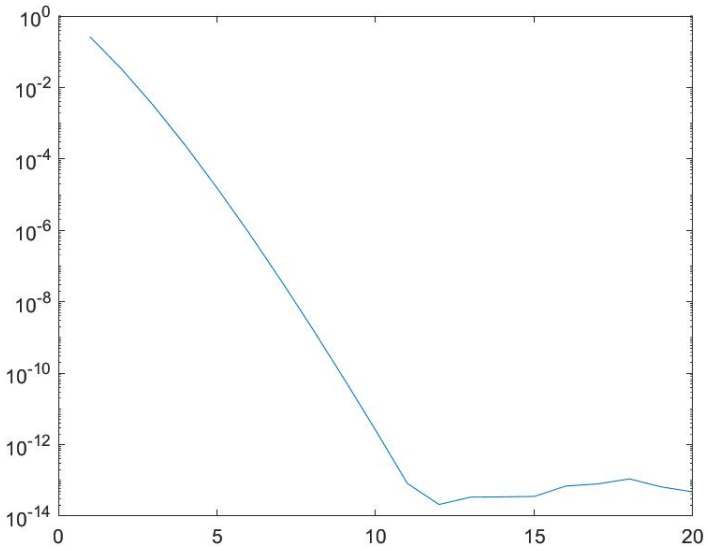


Figura 4: Grafico massimo errore su ascisse di Chebyshev. Asse x: n. Asse y: Massimo errore di interpolazione.

L'errore di interpolazione che possiamo osservare nei grafici in alto dipende dal numero di condizionamento del problema dell'interpolazione polinomiale, il quale è strettamente connesso alla costante di Lebesgue che, per ascisse equidistanti, tende esponenzialmente verso ∞ al crescere di n. Possiamo osservare, in entrambi i grafici, che fino ad un certo valore di n l'errore massimo decresce, a causa del crescere del numero di ascisse di interpolazione che consegue una precisione maggiore da parte

del polinomio interpolante nell'approssimare la funzione.
Tuttavia, da un certo valore di n in poi, circa n = 12, possiamo notare un'inversione del trend del grafico: nel caso di ascisse equidistanti l'errore inizia a crescere esponenzialmente Fig. 3, come ci aspettavamo, mentre nel caso di ascisse di Chebyshev, la crescita è logaritmica Fig. 4.

Esercizio 4.3. *Utilizzando la function* `spline0` *vista in esercitazione, costruitre una function Matlab,* ***splinenat****, avente la stessa sintassi della function* ***spline****, che calcoli la spline cubica naturale interpolante una funzione.*

Soluzione. Di seguito l'implementazione della function **splinenat** che fa uso della function `spline0`, mentre per il calcolo dei valori assunti dalla spline nelle ascisse richieste, si basa sullo studio teorico condotto a lezione sul calcolo efficiente di una spline cubica.

```
function spline = splinenat(xi, fi, xx)
%
%   spline = splinenat(xi, fi, xx)
%   Calcola la spline cubica naturale interpolante di una certa funzione
%
%   La function prende in input:
%   - xi: ascisse di interpolazione
%   - fi: valori assunti dalla funzione interpolata in corrispondenza delle
%   ascisse di interpolazione
%   - xx: ascisse in cui avviene il calcolo della spline
%
%    La function restituisce in output:
%    - spline: valori assunti dalla spline sulle ascisse xx
%

    mi= spline0(xi,fi);
    n= length(xi)-1;
    spline = zeros(size(xx));
    k= 1;

    for i=2:n+1
        hi = xi(i) - xi(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));

        while xx(k)<xi(i)
            spline(k) = (((xx(k) - xi(i-1))^3 * mi(i) + ...
                + (xi(i) - xx(k))^3 * mi(i-1))/(6 * hi) + ...
                + qi * (xx(k) - xi(i-1)) + ri);
            k = k+1;
        end
    end
    return
end
```

Esercizio 4.4. *Utilizzare la function* ***splinenat*** *su definita per approssimare la funzione* $f(x) = \cos(\pi\frac{x^2}{2})$ *rispettivamente su n+1 ascisse equidistanti e con n+1 ascisse di Chebyshev sull'intervallo [-1,1]. Graficare (in formato* ***semilogy****) il massimo errore di interpolazione,* $n = 4, \ldots, 100$.

Soluzione. Di seguito sono riportati i grafici del massimo errore calcolato in corrispondenza di un vettore di ascisse equidistanti tra [-1, 1]. Come si può facilmente vedere la decrescita dell'errore è molto più accentuata utilizzando ascisse di Chebyshev in Fig. 6 anzichè ascisse equidistanti in Fig. 5 per la costruzione delle spline.

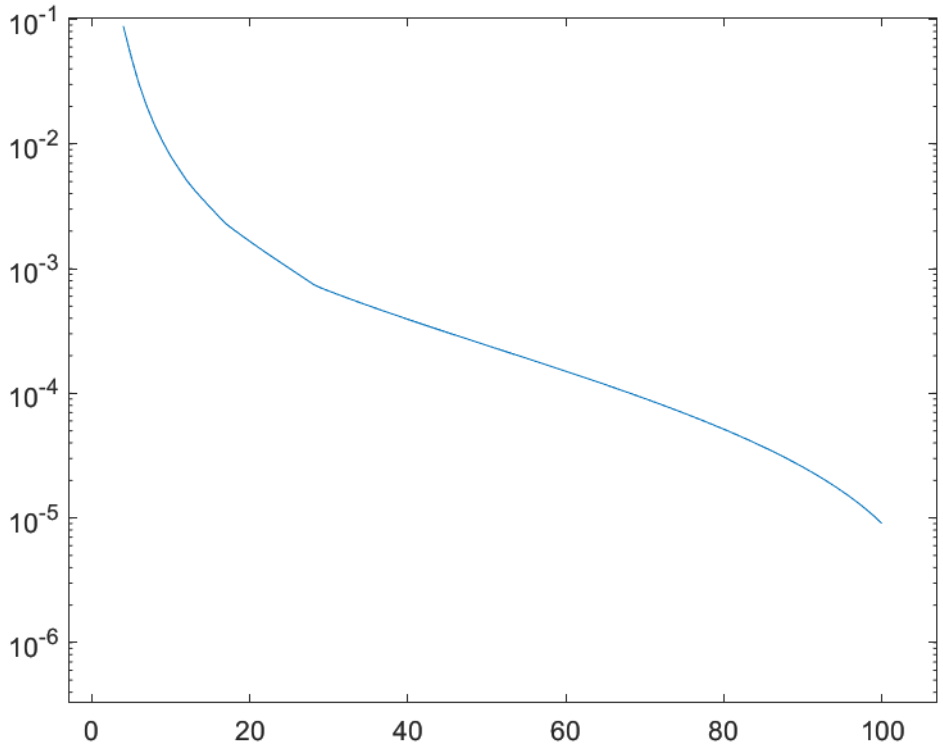


Figura 5: Grafico massimo errore su ascisse equidistanti. Asse x: n. Asse y: Massimo errore di interpolazione.

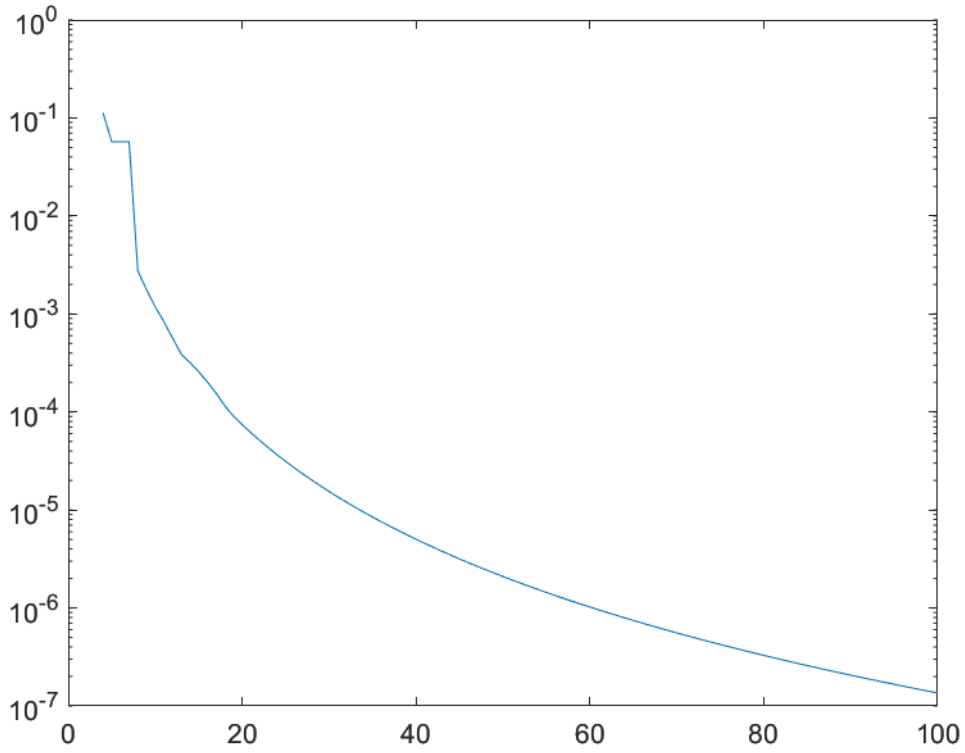


Figura 6: Grafico massimo errore su ascisse di Chebyshev. Asse x: n. Asse y: Massimo errore di interpolazione.

Esercizio 4.5. *Utitilizzare la function **spline** di Matlab per approssimare la funzione* $f(x) = \cos(\pi\frac{x^2}{2})$ *rispettivamente su* $n+1$ *ascisse equidistanti e su* $n+1$ *ascisse di Chebyshev sull'intervallo* $[-1, 1]$. *Graficare (in formato **semilogy**) il massimo errore di interpolazione, per* $n = 4, \ldots, 100$. *Compararli con quelli dell'esercizio precedente.*

Soluzione. Di seguito sono riportati i grafici dell'errorre massimo di interpolazione ottenuto con la function **spline** di Matlab, prima su $n+1$ ascisse di interpolazione equidistanti in Fig. 7 e poi su $n+1$ ascisse di Chebyshev in Fig. 8:

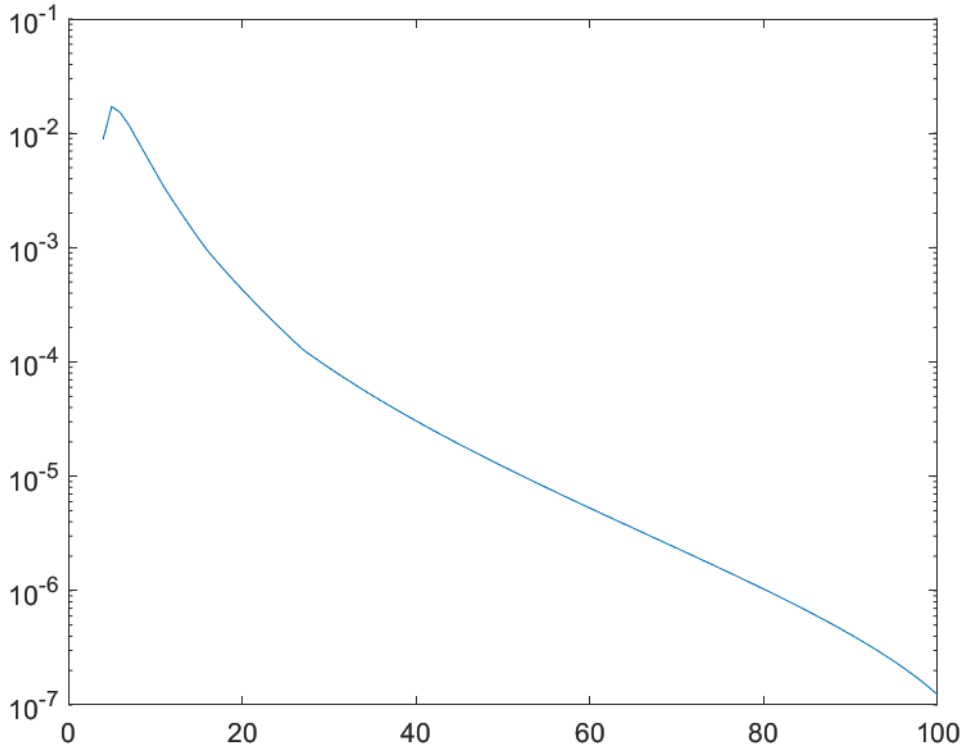


Figura 7: Grafico massimo errore su ascisse equidistanti. Asse x: n. Asse y: Massimo errore di interpolazione.

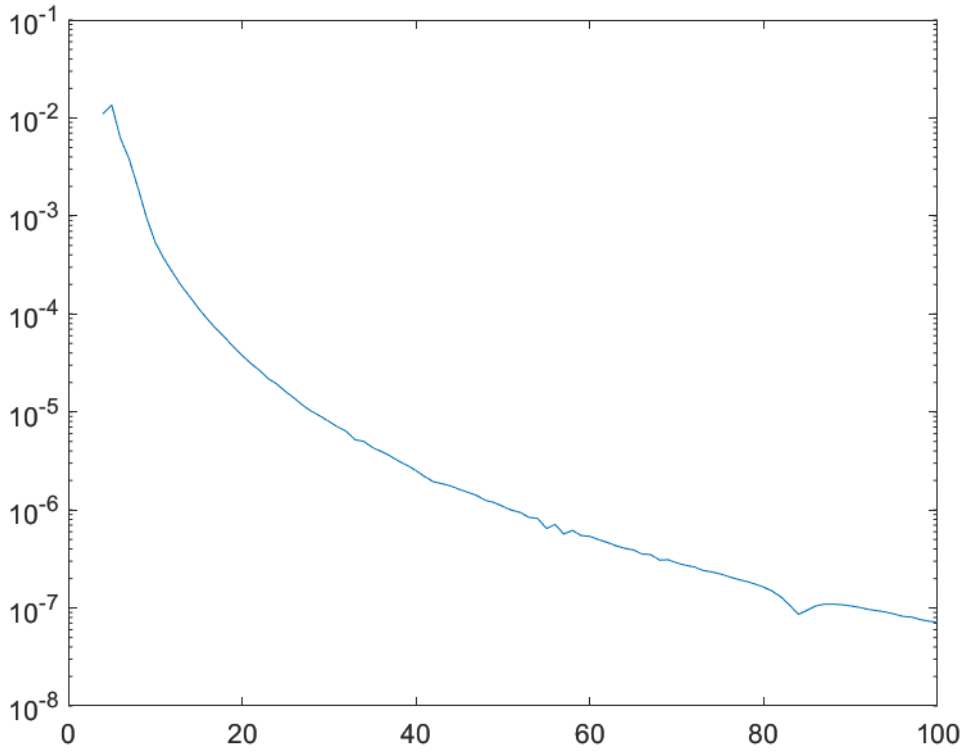


Figura 8: Grafico massimo errore su ascisse di Chebyshev. Asse x: n. Asse y: Massimo errore di interpolazione.

Osservando i grafici in Fig. 7 e in Fig. 8, del massimo errore ottenuto usando la function **spline** di Matlab, possiamo osservare che, oltre ad avere un andamento molto simile, entrambi decrescono al crescere del numero di ascisse di interpolazione. Questo perché, in caso di spline cubiche, l'utilizzo di ascisse equidistanti, piuttosto che quelle di Chebyshev, dà origine ad un problema sempre ben condizionato. Aggiungiamo che questa evidenza è in linea con l' **osservazione 4.3** del libro di testo, ricordando che la function **spline** di Matlab corrisponde alla spline cubica not-a-knot.
Confrontando questi risultati con quelli dell'esercizio precedente (es: 18/4.5) possiamo evincere che l'andamento ottenuto in Fig. 5 e in Fig. 6 è molto simile a quello generato dalla function **spline**, e questo ci fa concludere che l'implementazione della spline cubica naturale effettuata dalla function **splinenat**, oltre ad essere in linea con l' **osservazione 4.3** del libro di testo, è efficiente se confrontata con la function **spline** nativa di Matlab.

Esercizio 4.6. *Sia assegnata la seguente perturbazione della funzione* $f(x) = \cos(\pi \cdot \frac{x^2}{2})$:

$$\widetilde{f}(x) = f(x) + 10^{-3}\mathtt{rand}(\mathtt{size}(x)),$$

in cui ***rand*** *è la function built-in di Matlab. Calcolare polinomio di approssimazione ai minimi quadrati di grado* m*,* $p(x)$*, sui dati* $(x_i, \widetilde{f}(x_i))$*,* $i = 0, \ldots, n$*, con*

$$x_i = -1 + 2i/n, \qquad n = 10^4.$$

Graficare (in formato ***semilogy****) l'errore di approssimazione* $\|f - p\|$*, relativo all'intervallo [-1,1], rispetto ad m, per* $m = 1, \ldots, 20$*. Commentare i risultati ottenuti.*

Soluzione. Di seguito, in Fig. 9 , viene riportato l'errore di approssimazione $\|f - p\|$, relativo all'intervallo [-1,1] rispetto ad m.

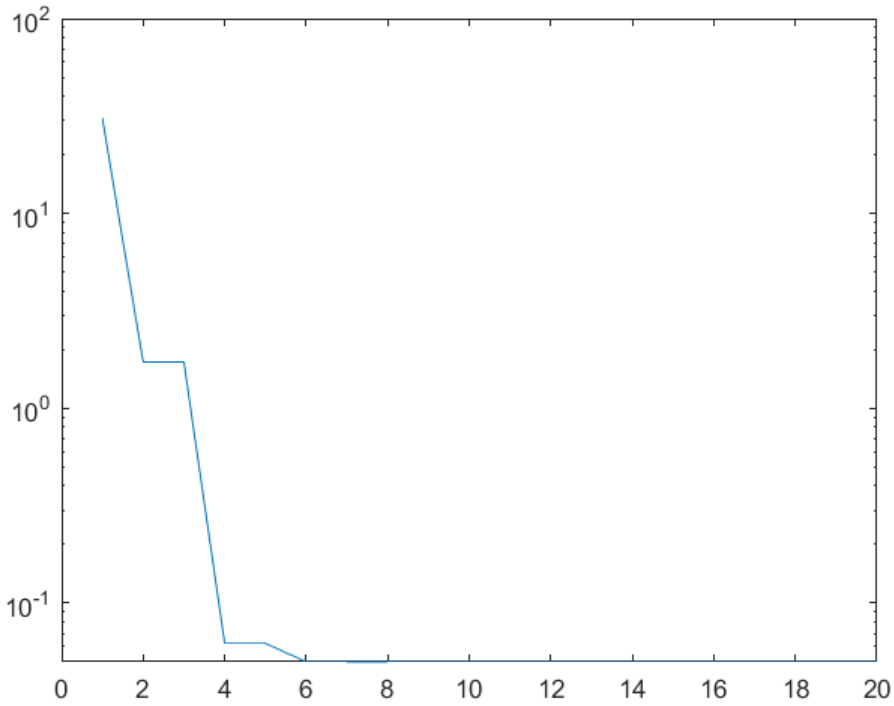


Figura 9: Grafico errore di approssimazione. Asse y: norma euclidea errore. Asse x: grado polinomio.

Si osserva dalla tabella in Fig. 9 che al crescere del grado m, ovvero da m = 6 in poi, il polinomio di approssimazione ai minimi quadrati diventa sempre più preciso, poiché la norma dell'errore tende a zero.

5

FORMULE DI QUADRATURA

Esercizio 5.1. *Costruire una function Matlab che, dato in input* n*, restituisca i pesi della quadratura della formula di Newton-Cotes di grado* n*.*
Tabulare quindi i pesi delle formule di grado $1, 2, \ldots 7$ *(come numeri razionali).*

Soluzione. Di seguito viene rappresentata la function Matlab richiesta per il calcolo dei pesi della quadratura della formula di Newton-Cotes di grado n:

```
function c = calcolaPesi(n)
%
%   c = calcolaPesi(n) calcola i pesi della quadratura
%   della formula di Newton-Cotes di grado n.
%   La function prende in input:
%   - n = grado della formula di quadratura Newton-Cotes.
%
%   La function restituisce in output:
%   - c = vettore dei pesi.
%
    c = zeros(1,n+1);
    for i=1:n+1
        c(i) = integral(@calcolaLin, 0, n);
    end

    function Lin = calcolaLin(t)
        indexi = i-1;
        ind = [0:indexi-1 indexi+1:n];
        Lin = 1/prod( indexi-ind );
        for j = 1:n
            Lin = Lin.*( t - ind(j));
        end
    end
    return
end
```

Tabella dei pesi ottenuti

n	c_{1n}	c_{2n}	c_{3n}	c_{4n}	c_{5n}	c_{6n}	c_{7n}	c_{8n}
1	0.5000	0.5000						
2	0.3333	1.3333	0.3333					
3	0.3750	1.1250	1.1250	0.3750				
4	0.3111	1.4222	0.5333	1.4222	0.3111			
5	0.3299	1.3021	0.8681	0.8681	1.3021	0.3299		
6	0.2929	1.5429	0.1929	1.9429	0.1929	1.5429	0.2929	
7	0.3042	1.4490	0.5359	1.2108	1.2108	0.5359	1.4490	0.3042

Esercizio 5.2. *Utilizzare la function del precedente esercizio per graficare, in formato* ***semilogy****, il rapporto* $\frac{k_n}{k}$ *rispetto ad* n*, essendo* k *il numero di condizionamento di un integrale definito, e* k_n *quello della formula di Newton-Cotes utilizzata di grado* n *per approssimarlo. Riportare i risultati per* $n = 1, \ldots, 50$.

Soluzione. Di seguito vengono riportati il codice Matlab per calcolare e graficare il rapporto $\frac{k_n}{k}$ e il grafico ottenuto in Fig. 10.

```
function rapporto_k = rapporto_condizionamento(a,b)
%
%   rapporto_k = rapporto_condizionamento(a,b) calcola
%   il rapporto kn/k, ovvero il rapporto tra il numero
%   di condizionamento della formula di quadratura di
%   Newton-Cotes e quello del calcolo dell'integrale
%   definito tra "a" e "b".
%
%   La function prende in input:
%   - a,b = estremi di integrazione.
%
%   La function restituisce in output:
%   - rapporto_k = vettore contenente valori di kn/k.
%
k = b-a;

for n = 1:50
    c  = calcolaPesi(n);
    kn = (b-a)*(sum(abs(c))/n);
    rapporto_k(n) = kn/k;
end

    return
end
```

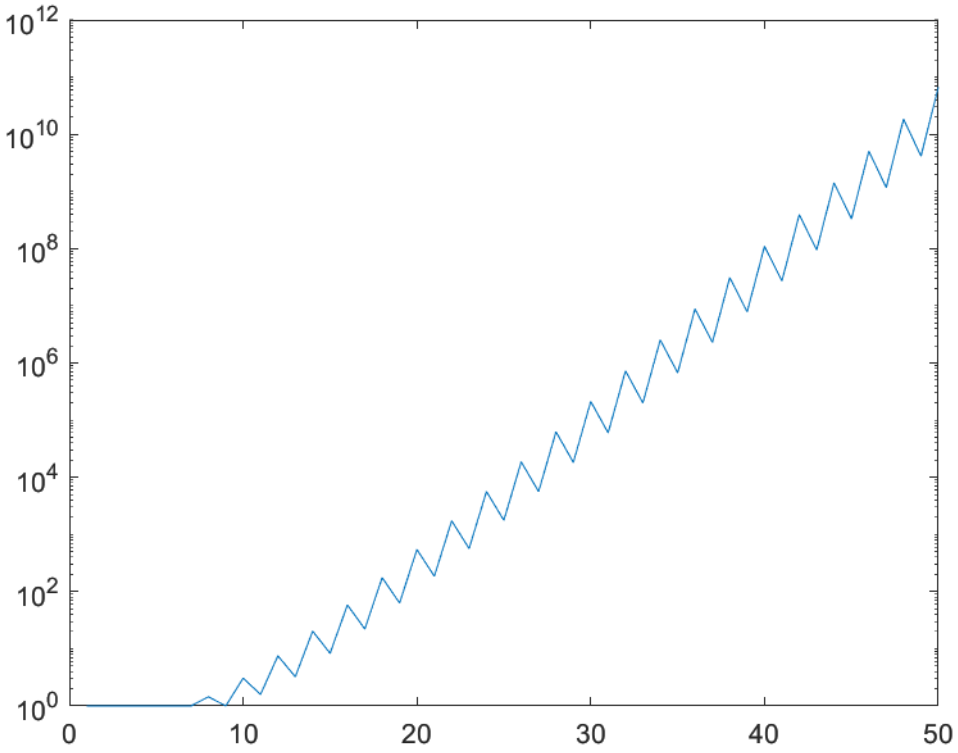


Figura 10: Grafico rapporto $\frac{k_n}{k}$. Asse y: $\frac{k_n}{k}$, Asse x: n

Come possiamo osservare per $n = 1, 2...7$ ed $n = 9$ il rapporto tra i due indicatori k e k_n è uguale ad 1, infatti per i valori di n elencati, i pesi delle formule di quadratura corrispondenti risultano tutti positivi ed il numero di condizionamento del problema coincide con quello del semplice calcolo di un integrale definito, ovvero $kn = k = (b - a)$ con a, b estremi dell'intervallo di integrazione.
Per $n = 8$ ed $n = 10....\infty$ i pesi non sono tutti positivi ed il numero di condizionamento k_n subisce una crescita esponenziale rispetto al numero di condizionamento k.

Esercizio 5.3. *Tabulare le approssimazioni dell'integrale*

$$I(f) = \int_{-1}^{1.1} \tan(x)dx \equiv \log(\frac{\cos(1)}{\cos(1.1)}),$$

ottenute mediante le formule di Newton-Cotes di grado n, $n = 1, \ldots, 9$. *Tabulare anche il relativo errore (in notazione scientifica con 3 cifre significative).*

Soluzione. Di seguito è riportata la tabulazione dell'approssimazione dell'integrale e del relativo errore.

Tabella dei valori ottenuti e degli errori

n	$I_{NC}(f)$	errore
1	$4.277e^{-01}$	$2.528e^{-01}$
2	$2.126e^{-01}$	$3.771e^{-02}$
3	$1.963e^{-01}$	$2.136e^{-02}$
4	$1.804e^{-01}$	$5.508e^{-03}$
5	$1.785e^{-01}$	$3.606e^{-03}$
6	$1.760e^{-01}$	$1.070e^{-03}$
7	$1.757e^{-01}$	$7.438e^{-04}$
8	$1.752e^{-01}$	$2.352e^{-04}$
9	$1.751e^{-01}$	$1.689e^{-04}$

Esercizio 5.4. *Confrontare le formule composite dei trapezi e di Simpson per approssimare l'integrale del precedente esercizio, per valori crescenti del numero dei sottointervalli dell'intervallo di integrazione. Commentare i risultati ottenuti, in termini di costo computazionale.*

Soluzione. Di seguito viene rappresentata la tabella che confronta i valori ottenuti dal metodo dei trapezi e dal metodo di Simpson, al crescere degli n sottointervalli.

Tabella dei valori ottenuti

n	Trapezi	Simpson
1	0.4277	//////
2	0.2664	0.2126
3	0.2220	//////
4	0.2034	0.1824
5	0.1940	//////
6	0.1885	0.1773
7	0.1851	//////
8	0.1828	0.1759
9	0.1812	//////

Confrontando i risultati in tabella con il valore ottenuto dal calcolo dell'integrale definito con la function **integral** di Matlab, che è 0.1749, possiamo osservare, che, a parità di cicli di iterazione, il metodo più preciso è quello di Simpson. Confrontando il numero di flops delle due funzioni, per il metodo dei trapezi abbiamo **n + 6** flops, mentre per il metodo di Simpson abbiamo **n + 5** flops, dove n indica il numero dei sottointervalli.

Esercizio 5.5. *Confrontare le formule adattive dei trapezi e di Simpson, con tolleranza* $\texttt{tol} = 10^{-2}, 10^{-3}, \ldots, 10^{-6}$*, per approssimare l'integrale definito:*

$$I(f) = \int_{-1}^{1} \frac{1}{1 + 10^2 x^2} \, dx.$$

Commentare i risultati ottenuti, in termini di costo computazionale.

Soluzione. Di seguito sono riportate due tabelle:

- Nella prima sono riportati i valori ottenuti con la function **adaptrap** (che implementa la formula adattiva dei trapezi) ed il numero delle chiamate ricorsive eseguite per ottenere il risultato con diversi valori di tolleranza.
- Nella seconda sono riportati i valori ottenuti con la function **adapsim** (che implementa la formula adattiva di Simpson) ed il numero delle chiamate ricorsive eseguite per ottenere il risultato con diversi valori di tolleranza.

Integrale calcolato con formula adattiva dei trapezi

Tolleranza	Integrale approssimato	Numero di chiamate ricorsive
10^{-2}	0.2956	19
10^{-3}	0.2946	91
10^{-4}	0.2943	275
10^{-5}	0.2942	791
10^{-6}	0.2942	2691

Integrale calcolato con formula adattiva di Simpson

Tolleranza	Integrale approssimato	Numero di chiamate ricorsive
10^{-2}	0.2813	7
10^{-3}	0.2813	8
10^{-4}	0.2943	19
10^{-5}	0.2942	39
10^{-6}	0.2942	71

In termini di costo computazionale possiamo vedere che il numero di chiamate ricorsive effettuate al diminuire della tolleranza concessa, aumenta molto rapidamente con la **formula adattiva dei trapezi**, mentre ha una crescita molto più lenta nel caso della **formula adattiva di Simpson**.
In termini di costo computazionale (flops) abbiamo:

- Formula adattiva dei trapezi: $10 + 16 \cdot$ **numero di chiamate ricorsive** flops.
- Formula adattiva di Simpson: $15 + 34 \cdot$ **numero di chiamate ricorsive** flops.