Interpolazione Polinomiale con Nodi di Leja Approssimati

Spiegazione e Implementazione in MATLAB

Marco Cola

1 Introduzione

L'obiettivo del progetto è costruire un'interpolazione polinomiale utilizzando nodi di **Leja approssimati**, selezionati in modo iterativo per massimizzare la stabilità numerica. Il progetto prevede:

- Il calcolo dei **punti di Leja approssimati**;
- La determinazione della **costante di Lebesgue**;
- La costruzione e il confronto di interpolanti polinomiali;
- L'analisi delle prestazioni computazionali.

2 Introduzione ai Nodi di Leja

I **nodi di Leja** sono una sequenza di punti selezionati iterativamente in modo da massimizzare la separazione tra i nodi e garantire una migliore stabilità numerica nei problemi di interpolazione polinomiale.

2.1 Costruzione dei Nodi di Leja

Dati un insieme di punti candidati $\{x_1, x_2, ..., x_N\}$ in un intervallo $[a, b]$, i nodi di Leja $\{z_0, z_1, ..., z_d\}$ vengono costruiti come segue:

1. **Scelta del primo nodo**: si seleziona un punto iniziale, tipicamente l'estremo sinistro dell'intervallo o il punto più vicino all'origine.

2. **Selezione iterativa**: ogni nodo successivo viene scelto come il punto x_j che massimizza il prodotto delle distanze rispetto ai nodi già selezionati:

$$z_k = \arg\max_{x \in \mathcal{X}} \prod_{i=0}^{k-1} |x - z_i| \tag{1}$$

Questo criterio garantisce una distribuzione spaziale ottimale dei nodi, riducendo l'instabilità numerica.

2.2 Importanza dei Nodi di Leja

- **Stabilità numerica**: riducono la crescita della costante di Lebesgue, migliorando il condizionamento dell'interpolazione polinomiale.
- **Migliore approssimazione**: rispetto ai nodi equispaziati, riducono l'errore nei problemi di interpolazione.
- **Utilizzo nei metodi di interpolazione**: sono usati in approssimazione numerica, interpolazione di Chebyshev e quadratura numerica.

3 Calcolo dei Punti di Leja Approssimati

3.1 Algoritmo 1: DLP

La funzione `DLP` seleziona iterativamente i nodi di Leja massimizzando la produttoria delle distanze dai nodi già scelti.

- **Input:**
  - `x` vettore di punti nell'intervallo I da cui estrarre i punti di Leja approssimati
  - `d` scalare intero positivo che corrisponde al grado relativo al polinomio interpolante di cui estraiamo i `d+1` nodi;
- **Output:**
  - `dlp` vettore di `d+1` punti corrispondenti ai punti di Leja approssimati.

3.2 Implementazione in MATLAB

```
function dlp = DLP(x, d)
    dlp = zeros(1, d+1);
    dlp(1) = x(1);

    for s = 2:d+1
        [~, idx] = max(prod(abs(x - dlp(1:s-1)), 2));
        dlp(s) = x(idx);
    end
end
```

3.3 Spiegazione del Codice

- **Inizializzazione:** Si crea un vettore riga `dlp` di lunghezza `d+1` inizializzato a zero che conterrà i nodi di Leja.
- **Scelta del primo nodo:** Si assegna il primo elemento della mesh `x` a `dlp(1)`.
- **Ciclo iterativo:** Per ogni `s` da 2 a `d+1`:
  - Si calcola la distanza assoluta tra ogni punto di `x` e i nodi già selezionati.
  - Si calcola il **prodotto delle distanze** nella seconda dimensione, cioè per ogni riga della matrice, per ogni punto candidato. Il risultato è un vettore colonna in cui ogni elemento è il prodotto delle distanze di x_i dai nodi selezionati.
  - Trova il valore massimo di questo vettore, ovvero si sceglie il punto che **massimizza** il prodotto delle distanze.
  - `[ ,idx]` Restituisce l'indice del punto che massimizza il prodotto, `x(idx)`, che viene poi assegnato a dlp(s).

3.4 Algoritmo 2 per i punti discreti di Leja: DLP2

La funzione `DLP2` è una variante di `DLP` che calcola i punti di Leja approssimati utilizzando la **fattorizzazione LU** della matrice di Vandermonde con la **base di Chebyshev**. Questo metodo seleziona iterativamente i nodi di Leja in base all'ordinamento determinato dal pivoting della fattorizzazione LU, migliorando l'efficienza computazionale e la stabilità numerica rispetto al metodo basato sulla massimizzazione della produttoria delle distanze.

- **Input:**
  - `x` vettore di punti nell'intervallo I da cui estrarre i punti di Leja approssimati;
  - `d` scalare intero positivo che corrisponde al grado relativo al polinomio interpolante di cui estraiamo i `d+1` nodi
- **Output:**
  - `dlp` vettore di `d+1` punti corrispondenti ai punti di Leja approssimati.

3.5 Implementazione in MATLAB

```
function dlp = DLP2(x, d)
    V = cos((0:d)' * acos(x'));
    [~, ~, P] = lu(V, 'vector');
    dlp = x(P(1:d+1));
end
```

3.6 Spiegazione del Codice

- **Costruzione della matrice di Vandermonde:** La matrice di Vandermonde viene costruita usando la base di Chebyshev:

$$V_{i,j} = \cos((j-1)\arccos(x_i)), \quad \text{con } i = 1, \ldots, M, \quad j = 1, \ldots, d+1. \tag{2}$$

  - `(0:d)` genera un vettore riga [0,1,...,d] che rappresenta gli esponenti della base di Chebyshev
  - `acos(x')` trasforma i punti x nei loro corrispondenti valori nell'intervallo di Chebyshev.
  - `cos((0:d)' * acos(x'))` costruisce la matrice di Vandermonde usando la base di Chebyshev invece della base monomiale classica.
- **Fattorizzazione LU:** Viene applicata la fattorizzazione LU con pivoting tramite la funzione `lu(V, 'vector')` alla matrice di Vandermonde per ottenere una matrice di permutazione P, che rappresenta l'ordine di selezione dei punti per la stabilità numerica.
  - Il pivoting aiuta a ridurre l'errore numerico, selezionando i punti in modo che la matrice rimanga ben condizionata.

- **Selezione dei punti di Leja:** I primi `d+1` elementi della mesh `x` ordinati secondo `P` vengono selezionati come nodi di Leja approssimati.

4 Calcolo della Costante di Lebesgue

La function dovrà valutare il valore assoluto dei polinomi di Lagrange sul vettore x, sommandoli e andando poi a selezionare il massimo tra questi valori. Per valutare la stabilità dell'interpolazione, calcoliamo la **costante di Lebesgue**, definita come:

$$\lambda_n(x) = \sum_{i=0}^{n} |\ell_i(x)|, \tag{3}$$

dove $\ell_i(x)$ sono i polinomi di Lagrange.

- **Input:**
  - `z` vettore riga dei nodi dell'interpolante su cui costruire la funzione di Lebesgue;
  - `x` vettore colonna di punti nell'intervallo I su cui valutare la funzione di Lebesgue;
- **Output:**
  - `L` scalare reale, che è l'approssimazione della costante di Lebesgue

4.1 Implementazione in MATLAB

```
function L = leb_con(z, x)
    n = length(z);
    L_vals = zeros(size(x));

    for i = 1:n
        Li = prod((x - z([1:i-1, i+1:n])) ./ (z(i) - z
           ([1:i-1, i+1:n])), 2);
        L_vals = L_vals + abs(Li);
    end

    L = max(L_vals);
end
```

4.2 Spiegazione del Codice

- **Determinazione della lunghezza di z:** `n` è il numero di nodi di interpolazione (ovvero il grado del polinomio interpolante più uno).
- **Inizializzazione di `L_vals`:** `L_vals` ha la stessa dimensione di `x` e verrà usato per accumulare i valori della funzione di Lebesgue.
- **Ciclo sui nodi di interpolazione:** Itero su ciascun nodo di interpolazione `z(i)`, e per ogni nodo calcolo il polinomio fondamentale di Lagrange $l_i(x)$.
  - Il polinomio fondamentale di Lagrange viene costruito in modo vettoriale utilizzando il prodotto:

$$L_i(x) = \prod_{j \neq i} \frac{x - z_j}{z_i - z_j}. \tag{4}$$

  - In dettaglio:
    1. **‘z([1:i-1, i+1:n])‘:** Si selezionano tutti gli elementi di `z` tranne `z(i)`. Questo è necessario per calcolare il polinomio di Lagrange $l_i(x)$, che dipende dalla relazione tra i punti x e tutti gli altri nodi `z`, tranne il nodo corrente `z(i)`. Si usa `1:i-1` per selezionare i nodi prima del nodo `i` e `i+1:n` per quelli dopo.
    2. **‘x - z([1:i-1, i+1:n])‘:** Si calcola la differenza tra ogni punto `x` e i nodi esclusi, ottenendo il numeratore del prodotto.
    3. **‘z(i) - z([1:i-1, i+1:n])‘:** Si calcola la differenza tra il nodo corrente `z(i)` e tutti gli altri nodi `z`. Questo rappresenta il denominatore di ciascuna frazione.
    4. **‘(x - z([1:i-1, i+1:n])) ./ (z(i) - z([1:i-1, i+1:n]))‘:** Si calcola il rapporto tra il numeratore `x - z(...)` e il denominatore `z(i) - z(...)`, che rappresenta ogni termine del polinomio di Lagrange per ciascun nodo `i` su ogni punto `x`.
    5. **‘prod(..., 2)‘:** La funzione `prod` calcola il prodotto di tutti i termini lungo la dimensione 2 (ovvero lungo le righe), ottenendo il valore del polinomio di Lagrange $l_i(x)$ per ciascun punto `x`.
  - Sommo il valore assoluto di $l_i(x)$ a `L_vals` per costruire la **funzione di Lebesgue**.

– Poiché la costante di Lebesgue è definita come il massimo valore della funzione di Lebesgue, **max(L_vals)** restituisce il valore massimo della funzione nei punti `x`.

5 Interpolazione con Base di Chebyshev

Questa funzione calcola l'interpolazione polinomiale utilizzando la base di **Chebyshev** invece della classica base monomiale. Il metodo si basa sulla matrice di **Vandermonde** con polinomi di Chebyshev, che garantisce maggiore stabilità numerica rispetto alla base standard:

```
function p_eval = interp_chebyshev(x_nodes, f_nodes,
   x_eval)
    n = length(x_nodes);
    V = cos((0:n-1)' .* acos(x_nodes(:).'));
    c = V \ f_nodes(:);
    V_eval = cos((0:n-1)' .* acos(x_eval(:).'));
    p_eval = (V_eval' * c);
    p_eval = p_eval(:);
end
```

5.1 Firma della funzione

```
function p_eval = interp_chebyshev(x_nodes, f_nodes,
   x_eval)
```

- `x_nodes`: Nodi di interpolazione (punti noti).
- `f_nodes`: Valori della funzione nei nodi di interpolazione.
- `x_eval`: Punti in cui vogliamo valutare l'interpolante.
- `p_eval`: Risultato dell'interpolazione nei punti `x_eval`.

5.2 Determinazione della dimensione del problema

```
n = length(x_nodes);
```

Questo determina il numero di nodi di interpolazione, che stabilisce il grado del polinomio interpolante $n - 1$.

5.3 Costruzione della matrice di Vandermonde con la base di Chebyshev

```
V = cos((0:n-1)' .* acos(x_nodes(:).'));
```

- `(0:n-1)'` genera un vettore colonna $[0, 1, \ldots, n-1]$ che rappresenta gli esponenti della base di Chebyshev.
- `acos(x_nodes(:).')` trasforma i nodi in un vettore riga applicando la funzione arco-coseno.
- `cos(...)` costruisce la matrice di Vandermonde basata sulla base di Chebyshev.

5.4 Calcolo dei coefficienti dell'interpolante

```
c = V \ f_nodes(:);
```

Risoluzione del sistema lineare $V \cdot c = f$ per ottenere i coefficienti c. MATLAB utilizza un metodo ottimizzato backslash per evitare problemi numerici nella soluzione del sistema lineare.

5.5 Valutazione del polinomio interpolante nei punti desiderati

```
V_eval = cos((0:n-1)' .* acos(x_eval(:).'));
```

Simile alla costruzione di `V`, ma applicata ai punti di valutazione `x_eval`.

5.6 Calcolo del valore dell'interpolante

```
p_eval = (V_eval' * c);
```

Moltiplicazione della matrice `V_eval'` con il vettore dei coefficienti `c` per ottenere i valori interpolati.

5.7 Conversione del risultato in un vettore colonna

```
p_eval = p_eval(:);
```

Questa operazione assicura che il risultato sia un vettore colonna.

5.8 Riassunto del funzionamento

1. Costruzione della matrice di Vandermonde con la base di Chebyshev.
2. Risoluzione del sistema lineare $V \cdot c = f$ per trovare i coefficienti dell'interpolante.
3. Costruzione della matrice di Vandermonde per i punti `x_eval`.
4. Calcolo dell'interpolazione moltiplicando `V_eval'` per `c`.
5. Conversione del risultato in formato colonna.

5.9 Importanza nel progetto

- Utilizza la base di Chebyshev, riducendo gli errori numerici rispetto alla base monomiale.
- È più stabile rispetto all'interpolazione classica con `polyfit`.
- Fornisce un metodo numericamente robusto per l'interpolazione con nodi di Leja.

6 Script di Analisi e Visualizzazione

Questo script MATLAB esegue l'analisi delle prestazioni computazionali degli algoritmi di selezione dei nodi di Leja, calcola la costante di Lebesgue e confronta l'errore di interpolazione con nodi di Leja e nodi equispaziati.

6.1 Definizione dello Script

```
clc; clear; close all;

% Creazione della mesh
N = 1e4; % 10^4
x = linspace(-1, 1, N)';

% Range dei gradi del polinomio
degr = 1:50;

% Inizializzazione vettori per i tempi computazionali
times_DLP = zeros(size(degr));
times_DLP2 = zeros(size(degr));
```

```
% Inizializzazione vettori per la costante di Lebesgue
L_values = zeros(size(degr));

% Inizializzazione vettori per gli errori
errors_leja = zeros(size(degr));
errors_equi = zeros(size(degr));

% Funzione da interpolare
f = @(x) 1 ./ (x - 1.3);
```

6.2 Ciclo Principale di Calcolo

```
for d = degr
    % Misurazione tempi computazionali
    tic;
    nodes_leja = DLP(x, d);
    times_DLP(d) = toc;

    tic;
    nodes_leja2 = DLP2(x, d);
    times_DLP2(d) = toc;

    % Calcolo della costante di Lebesgue
    L_values(d) = max(leb_con(nodes_leja, x), [],'all');

    % Interpolazione con nodi di Leja
    f_leja = f(nodes_leja);
    p_leja = interp_chebyshev(nodes_leja, f_leja, x);

    % Interpolazione con nodi equispaziati
    nodes_equi = linspace(-1, 1, d+1)';
    f_equi = f(nodes_equi);
    p_equi = interp_chebyshev(nodes_equi, f_equi, x);

    % Calcolo errore massimo
    errors_leja(d) = max(abs(p_leja - f(x)));
    errors_equi(d) = max(abs(p_equi - f(x)));
end
```

6.3 Visualizzazione dei Risultati

- **Grafico dei Tempi Computazionali**: confronta le prestazioni degli algoritmi `DLP` e `DLP2`.
- **Grafico della Costante di Lebesgue**: mostra la crescita della costante di Lebesgue in scala semilogaritmica.
- **Confronto degli Errori**: confronta l'errore massimo di interpolazione tra nodi di Leja e nodi equispaziati.

6.3.1 Grafico dei Tempi Computazionali

```
figure;
plot(degr, times_DLP, 'r-o', degr, times_DLP2, 'b-o', '
   LineWidth', 1.5);
legend('DLP', 'DLP2');
xlabel('Grado del polinomio');
ylabel('Tempo di esecuzione (s)');
title('Confronto dei Tempi Computazionali');
grid on;
```

6.3.2 Confronto dei Tempi Computazionali

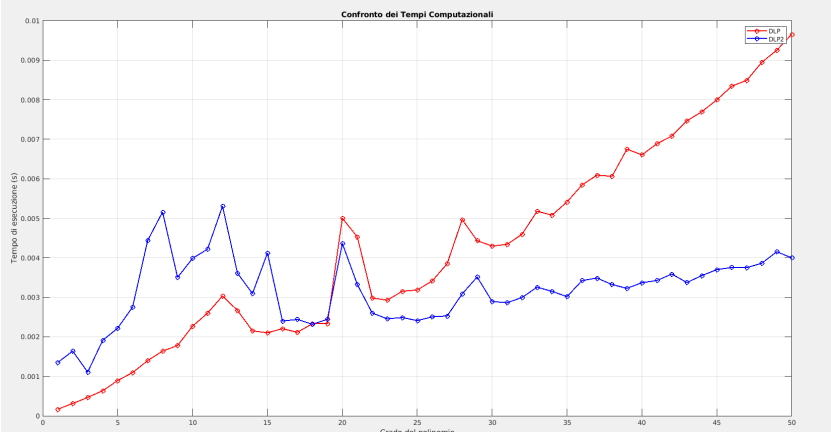

Figure 1: Confronto dei tempi computazionali tra diverse strategie di interpolazione.

Il grafico mostra il tempo computazionale richiesto dai vari metodi di interpolazione. È evidente che metodi con minore stabilità numerica possono richiedere tempi di calcolo più elevati a causa dell'accumulo di errori e

della necessità di una maggiore precisione numerica. Metodi basati su nodi ben distribuiti non solo riducono l'errore, ma migliorano anche l'efficienza computazionale, rendendoli preferibili in applicazioni pratiche.

6.3.3 Grafico della Costante di Lebesgue

```
figure;
semilogy(degr, L_values, 'b-o', 'LineWidth', 1.5);
xlabel('Grado del polinomio');
ylabel('Costante di Lebesgue');
title('Andamento della Costante di Lebesgue');
grid on;
```

6.3.4 Andamento della Costante di Lebesgue

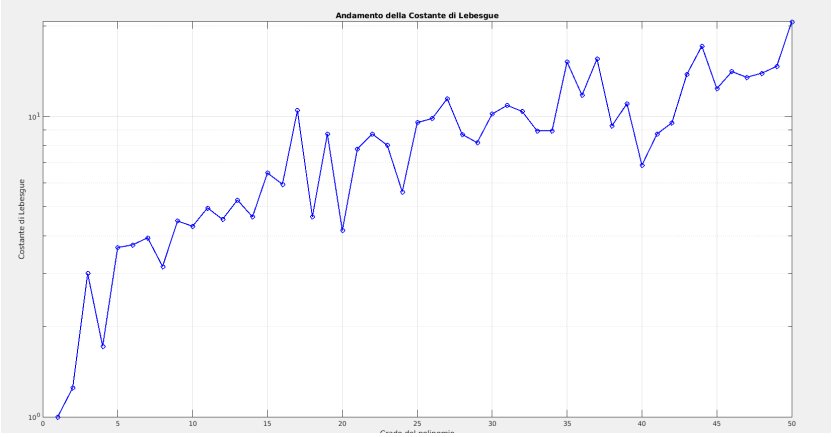

Figure 2: Andamento della costante di Lebesgue rispetto al numero di nodi.

La costante di Lebesgue misura la crescita della norma dell'operatore di interpolazione, influenzando la stabilità dell'interpolazione polinomiale. Dal grafico si osserva che la costante cresce significativamente con l'aumentare del numero di nodi equidistanti, confermando il noto fenomeno di Runge, che dimostra come l'interpolazione polinomiale con nodi equidistanti può fallire, e per migliorare la qualità dell'approssimazione è necessario adottare strategie più avanzate come i nodi di Chebyshev. Questo implica che l'interpolazione polinomiale con nodi equidistanti diventa numericamente instabile per un alto numero di nodi.

6.3.5 Grafico del Confronto degli Errori

```
figure;
semilogy(degr, errors_leja, 'b-o', degr, errors_equi, 'r
    -o', 'LineWidth', 1.5);
legend('Nodi di Leja', 'Nodi equispaziati');
xlabel('Grado del polinomio');
ylabel('Errore massimo');
title('Confronto Errori di Interpolazione');
grid on;
```

6.3.6 Confronto degli Errori di Interpolazione

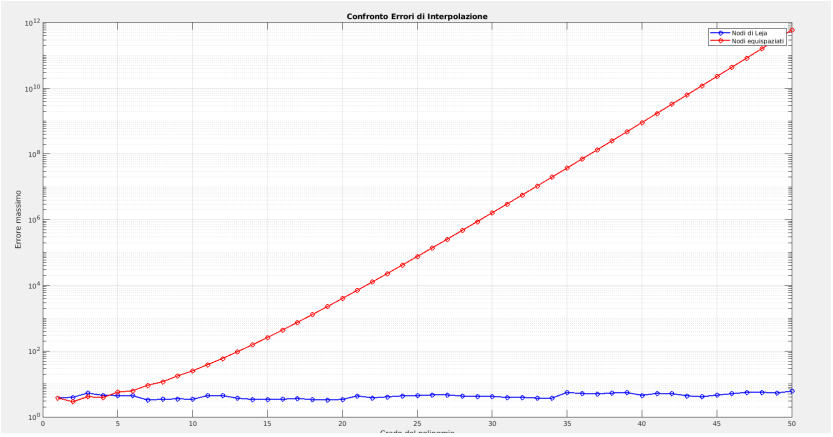

Figure 3: Confronto degli errori di interpolazione tra diversi metodi.

Questo grafico confronta gli errori di interpolazione ottenuti con differenti strategie di scelta dei nodi. Si nota che l'interpolazione con nodi equidistanti presenta errori maggiori rispetto a metodi basati su nodi di Chebyshev. La distribuzione ottimale dei nodi riduce il massimo errore, migliorando la convergenza dell'interpolante alla funzione esatta. La scelta dei nodi risulta quindi determinante per garantire un'accuratezza ottimale.

7 Esperimenti e Conclusioni

L'algoritmo viene testato sulla funzione:

$$f(x) = \frac{1}{x - 1.3}. \tag{5}$$

I risultati mostrano che l'interpolazione con nodi di Leja è **più stabile** rispetto a quella con nodi equispaziati.

7.1 Conclusioni

Dall'analisi dei grafici emergono le seguenti considerazioni principali:

- La costante di Lebesgue cresce rapidamente per nodi equidistanti, suggerendo instabilità per alti gradi polinomiali.
- L'errore di interpolazione è significativamente minore con nodi di Chebyshev rispetto ai nodi equidistanti.
- L'efficienza computazionale migliora con strategie di interpolazione più stabili, rendendo alcuni metodi preferibili per applicazioni numeriche.

Questi risultati confermano l'importanza della scelta dei nodi nell'interpolazione polinomiale e suggeriscono che metodi basati su nodi ottimali siano preferibili in termini di accuratezza e stabilità numerica. Implementando un'**interpolazione polinomiale stabile** con i nodi di Leja approssimati si ottiene una riduzione dell'errore rispetto ai nodi equispaziati.