Calcolissimo semplice (manco per il cazzo)

1. MAIN.M - Script Principale

```
clc; clear; close all;
```

- `clc`: pulisce la command window
- `clear`: cancella tutte le variabili dal workspace
- `close all`: chiude tutte le figure aperte

  Motivazione: Setup pulito per l'esecuzione

```
fprintf('=== PROGETTO NODI DI LEJA APPROSSIMATI ===\n\n');
```

Header informativo per identificare chiaramente l'output del progetto.

```
N = 10000;  % Punti della mesh
x = linspace(-1, 1, N)';
d = 10;     % Grado del polinomio di test
```

- `N = 10000`: dimensione della discretizzazione dell'intervallo [-1,1]. **Scelta**: compromesso tra precisione e costo computazionale
- `linspace(-1, 1, N)'`: crea N punti equidistanti in [-1,1] e li trasforma in **vettore colonna** (il transpose è cruciale!)
- `d = 10`: grado di test per il confronto iniziale dei due algoritmi

```
fprintf('Generazione mesh con N=%d punti in [-1,1]\n', N);
fprintf('Test con polinomio di grado d=%d\n\n', d);
```

Output informativo per tracciare i parametri utilizzati.

```
fprintf('--- CONFRONTO ALGORITMI PER d=%d ---\n', d);
```

Sezione di test preliminare per confrontare DLP vs DLP2 su un caso specifico.

```
tic;
nodi_dlp = DLP(x, d);
tempo_dlp = toc;
fprintf('DLP:  tempo = %.6f s\n', tempo_dlp);
```

- `tic/toc`: cronometro MATLAB per misurare tempo di esecuzione
- `DLP(x, d)`: chiama l'algoritmo 1 che implementa la ricerca iterativa
- **6 cifre decimali** per precisione nella misurazione dei tempi

```
tic;
nodi_dlp2 = DLP2(x, d);
tempo_dlp2 = toc;
fprintf('DLP2: tempo = %.6f s\n', tempo_dlp2);
```

Stesso processo per l'algoritmo 2 basato su fattorizzazione LU.

```
L_dlp = leb_con(nodi_dlp, x);
L_dlp2 = leb_con(nodi_dlp2, x);
```

Calcolo delle costanti di Lebesgue per valutare la stabilità numerica di entrambi gli approcci.

```
fprintf('\nCostanti di Lebesgue:\n');
fprintf('DLP:  L = %.4f\n', L_dlp);
fprintf('DLP2: L = %.4f\n\n', L_dlp2);
```

Output comparativo delle costanti di Lebesgue (4 cifre decimali sufficienti).

2. DLP.M - Algoritmo 1

```
function dlp = DLP(x, d)
```

Signature della funzione: input `x` (mesh), `d` (grado), output `dlp` (nodi di Leja).

```
if ~isvector(x) || ~isscalar(d) || d < 0 || round(d) ~= d || length(x) < d+1
    error('Input non valido: x deve essere un vettore, d intero positivo, length(x) >= d+1');
end
```

Validazione robusta degli input:

- `~isvector(x)`: x deve essere vettore (riga o colonna)
- `~isscalar(d)`: d deve essere scalare
- `d < 0`: d deve essere non negativo
- `round(d) ~= d`: d deve essere intero
- `length(x) < d+1`: servono almeno d+1 punti per selezionare d+1 nodi

```
x = x(:);
```

Normalizzazione formato: forza x ad essere vettore colonna per uniformità nelle operazioni matriciali successive.

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

- **Inizializzazione**: vettore riga di d+1 zeri
- **Primo nodo**: per convenzione si sceglie il primo punto della mesh come ξ_0

```
for s = 2:d+1
```

Loop principale: seleziona i nodi ξ_1, ξ_2, ..., ξ_a iterativamente.

```
produttoria = prod(abs(x - dlp(1:s-1)), 2);
```

Cuore dell'algoritmo:

- `x - dlp(1:s-1)`: differenza tra ogni punto della mesh e i nodi già selezionati (broadcasting)
- `abs(...)`: valore assoluto elemento per elemento
- `prod(..., 2)`: prodotto lungo le righe, calcola $\prod_{i=0}$^{s-2} $|x - \xi_i|$ per ogni x

Matematicamente: implementa la funzione obiettivo da massimizzare secondo la definizione dei nodi di Leja.

```
[~, idx_max] = max(produttoria);
dlp(s) = x(idx_max);
```

- `max(produttoria)`: trova il valore massimo e il suo indice
- `~`: ignora il valore massimo, tieni solo l'indice
- **Selezione**: il punto che massimizza la produttoria diventa il nuovo nodo di Leja

3. DLP2.M - Algoritmo 2

```
x = x(:);
```

Stessa normalizzazione del formato vettore colonna.

```
V = cos((0:d)' * acos(x'));
```

Costruzione matrice di Vandermonde di Chebyshev:

- `acos(x')`: calcola arccos per ogni elemento di x (x deve essere in [-1,1])
- `(0:d)'`: vettore colonna [0, 1, 2, ..., d]
- `(0:d)' * acos(x')`: prodotto esterno, crea matrice (d+1) × N
- `cos(...)`: applica coseno elemento per elemento

Matematicamente: $V(i,j) = T_{i-1}(x_j) = \cos((i-1)\cdot\arccos(x_j))$ dove T_k è il k-esimo polinomio di Chebyshev.

```
[~, ~, P] = lu(V, 'vector');
```

Fattorizzazione LU con pivoting:

- `lu(V, 'vector')`: calcola P·V = L·U dove P è vettore di permutazione
- `~, ~`: ignora le matrici L e U, tieni solo P
- **Proprietà matematica**: la permutazione P ordina automaticamente i punti secondo i criteri di massimizzazione del determinante (equivalente ai nodi di Leja)

```
dlp2 = x(P(1:d+1))';
```

- `P(1:d+1)`: prende i primi d+1 indici della permutazione
- `x(P(1:d+1))`: seleziona i corrispondenti punti dalla mesh
- `(...)'`: trasforma in vettore riga per uniformità con DLP

4. LEB_CON.M - Costante di Lebesgue

```
if isempty(z) || isempty(x)
    error('Input non valido: z e x non possono essere vuoti');
end
```

Validazione base: previene errori con input vuoti.

```
z = z(:)';  % vettore riga
x = x(:);   % vettore colonna
```

Normalizzazione formati:

- `z` (nodi): vettore riga per compatibilità con operazioni successive
- `x` (punti valutazione): vettore colonna per broadcasting corretto

```
n = length(z);
```

```
lebesgue_vals = zeros(size(x));
```

- `n`: numero di nodi di interpolazione
- **Inizializzazione**: vettore per accumulare i valori della funzione di Lebesgue $\lambda_n(x)$

```
for i = 1:n
```

Loop sui polinomi di Lagrange: calcola $\ell_i(x)$ per i = 1, ..., n.

```
altri_nodi = [1:i-1, i+1:n];
```

Indici dei nodi escluso il corrente: per calcolare il polinomio di Lagrange $\ell_i(x)$ servono tutti i nodi tranne ξ_i.

```
lagrange_poly = prod((x - z(altri_nodi)) ./ (z(i) - z(altri_nodi)), 2);
```

Calcolo polinomio di Lagrange:

- `x - z(altri_nodi)`: numeratore del prodotto, broadcasting automatico
- `z(i) - z(altri_nodi)`: denominatore del prodotto
- `./`: divisione elemento per elemento
- `prod(..., 2)`: prodotto lungo le colonne

Matematicamente: $\ell_i(x) = \prod_{j \neq i} (x-\xi_j)/(\xi_i-\xi_j)$

```
lebesgue_vals = lebesgue_vals + abs(lagrange_poly);
```

Accumulo: $\lambda_n(x) = \Sigma_{i=1}^{n} |\ell_i(x)|$

```
L = max(lebesgue_vals);
```

Costante di Lebesgue: L = max_{x∈[-1,1]} $\lambda_n(x)$

5. INTERP_CHEBYSHEV.M - Interpolazione

```
if length(x_nodes) ~= length(f_nodes)
    error('x_nodes e f_nodes devono avere la stessa lunghezza');
end
```

Consistenza dati: nodi e valori funzione devono corrispondere.

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

Matrice del sistema: stessa logica di DLP2 ma con dimensione n×n (sistema quadrato).

```
c = V \ f_nodes;
```

Risoluzione sistema lineare: V·c = f_nodes per trovare i coefficienti c del polinomio nella base di Chebyshev.

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

Valutazione polinomio:

- Costruisce matrice di valutazione nei punti x_eval
- Calcola p(x_eval) = V_eval^T · c

6. SPERIMENTAZIONE.M - Analisi Prestazioni

```
d_max = 50;
gradi = 1:d_max;
```

Range di test: da grado 1 a 50 per analisi completa.

```
tempi_DLP = zeros(size(gradi));
tempi_DLP2 = zeros(size(gradi));
lebesgue_vals = zeros(size(gradi));
```

Pre-allocazione: ottimizzazione memoria, evita crescita dinamica array.

```
if mod(d, 10) == 0
    fprintf('%d ', d);
end
```

Progress indicator: mostra avanzamento ogni 10 iterazioni per feedback utente.

```
if tempi_DLP2(i) < tempi_DLP(i)
    lebesgue_vals(i) = leb_con(nodi_dlp2, x);
else
    lebesgue_vals(i) = leb_con(nodi_dlp, x);
end
```

Selezione algoritmo ottimale: usa l'algoritmo più veloce per il calcolo della costante di Lebesgue (ottimizzazione computazionale).

```
plot(gradi, tempi_DLP, 'r-o', 'LineWidth', 2, 'MarkerSize', 4);
hold on;
plot(gradi, tempi_DLP2, 'b-s', 'LineWidth', 2, 'MarkerSize', 4);
```

Visualizzazione:

- `'r-o'`: linea rossa con cerchi
- `'b-s'`: linea blu con quadrati
- `LineWidth`, `MarkerSize`: estetica professionale
- `hold on`: sovrappone i grafici

```
semilogy(gradi, lebesgue_vals, 'g-o', 'LineWidth', 2, 'MarkerSize', 4);
```

Scala semilogaritmica: evidenzia la crescita esponenziale della costante di Lebesgue.

7. TEST_INTERPOLAZIONE.M - Confronto Accuratezza

```
f = @(x) 1 ./ (x - 1.3);
```

Funzione test: singolarità in 1.3 (fuori da [-1,1]) per testare stabilità numerica.

```
tic; DLP(x, 10); t1 = toc;
tic; DLP2(x, 10); t2 = toc;
usa_dlp2 = (t2 < t1);
```

Benchmark preliminare: determina quale algoritmo è più efficiente per questa mesh.

```
nodi_equi = linspace(-1, 1, d+1)';
```

Nodi equispaziati: baseline di confronto, notoriamente instabili per gradi elevati.

```
errori_leja(i) = max(abs(p_leja - f_vals));
errori_equi(i) = max(abs(p_equi - f_vals));
```

Errore massimo: norma infinito della differenza tra polinomio interpolante e funzione esatta.

Il resto del codice segue logiche analoghe per analisi statistiche e visualizzazione.

Questa implementazione è **matematicamente rigorosa** e **computazionalmente efficiente**, rispettando fedelmente la teoria dei nodi di Leja e fornendo strumenti completi per l'analisi comparativa.

No, non tutto al 100%. Ho spiegato in dettaglio assoluto le parti **matematicamente e algoritmicamente cruciali**, ma mi sono concentrato sui blocchi principali tralasciando alcune righe di:

Parti Non Dettagliate

In `sperimentazione.m`:

```
tempo_medio_DLP = mean(tempi_DLP);
tempo_medio_DLP2 = mean(tempi_DLP2);

if tempo_medio_DLP2 < tempo_medio_DLP
   fprintf('DLP2 è mediamente più veloce (%.6f s vs %.6f s)\n', ...
          tempo_medio_DLP2, tempo_medio_DLP);
else
   fprintf('DLP è mediamente più veloce (%.6f s vs %.6f s)\n', ...
          tempo_medio_DLP, tempo_medio_DLP2);
end
```

Logica: calcolo statistiche comparative e output formattato per determinare quale algoritmo è globalmente superiore.

```
L_min = min(lebesgue_vals);
L_max = max(lebesgue_vals);
L_finale = lebesgue_vals(end);

fprintf('Costante di Lebesgue:\n');
fprintf('  Minima: %.4f (d=%d)\n', L_min, find(lebesgue_vals == L_min, 1));
fprintf('  Massima: %.4f (d=%d)\n', L_max, find(lebesgue_vals == L_max, 1));
fprintf('  Finale (d=%d): %.4f\n', d_max, L_finale);
```

Analisi statistica: estrae min/max/finale delle costanti di Lebesgue con `find()` per localizzare gli indici corrispondenti.

In `test_interpolazione.m`:

```
leja_migliori = errori_leja < errori_equi;
perc_leja_migliori = 100 * sum(leja_migliori) / length(gradi);
```

Confronto percentuale: `errori_leja < errori_equi` crea array booleano, `sum()` conta i `true`, divisione per lunghezza dà percentuale.

```
if length(errori_leja) > 10
    crescita_leja = errori_leja(end) / errori_leja(end-9);
    crescita_equi = errori_equi(end) / errori_equi(end-9);
```

Analisi stabilità: confronta errori finali vs errori 10 posizioni prima per misurare il tasso di crescita dell'instabilità.

In `interp_chebyshev.m`:

```
if length(x_nodes) ~= length(f_nodes)
    error('x_nodes e f_nodes devono avere la stessa lunghezza');
end

% Assicura formato colonna
x_nodes = x_nodes(:);
f_nodes = f_nodes(:);
x_eval = x_eval(:);
```

Validazione e normalizzazione complete per robustezza.

E varie righe di formattazione grafici:

```
xlabel('Grado del polinomio d');
ylabel('Tempo di esecuzione (s)');
title(sprintf('Confronto Tempi Computazionali (N = %d)', N));
legend('DLP (Produttoria)', 'DLP2 (LU)', 'Location', 'northwest');
grid on;
```

Vuoi che entri nel dettaglio anche di queste parti "secondarie"? O sei soddisfatto della spiegazione dei **core algorithms**?