1 Lezione del 28-03-25

Riprendiamo il discorso sull'errore inerente dei sistemi lineari.

1.0.1 Condizionamento in δA

Avevamo preso delle perturbazioni sulle matrici A e b (dovute a vari effetti reali, quali errori di arrotondamento, di misura, ecc...) nella forma:

$$(A + \delta A)(x + \delta x) = (b + \delta b)$$

e volevamo capire quanto può essere grande l'errore relativo $\frac{|\delta x|}{|x|}$, da:

$$\frac{\text{sol. perturbata} - \text{sol. reale}}{\text{sol. reale}} = \frac{|x + \delta x - x|}{|x|} = \frac{|\delta x|}{|x|}$$

Nel caso di $\delta A = 0$, abbiamo visto di poter maggiorare tale quantità come:

$$\frac{|\delta x|}{|x|} \leq \mu(A) \cdot \frac{|\delta b|}{|b|}$$

con $\mu(A) = |A| \cdot |A^{-1}|$ *numero di condizionamento* (definizione 9.1).

Riguardo a $\mu(A)$, si ha che è ≥ 1, cioè chiaramente non si può ridurre l'errore oltre la perfezione, e se $\mu(A) \approx 10^k$, k è il numero di cifre significative che si *perdono* nel risultato $x + \delta x$.

Possiamo quindi reintrodurre il termine δA ed enunciare il seguente teorema:

Teorema 1.1: Condizionamento in δA

Se $|\delta A| \cdot |A^{-1}| < 1$ allora si ha:

$$\frac{|\delta x|}{|x|} \leq \frac{\mu(A)}{1 - \mu(A) \cdot \frac{|\delta A|}{|A|}} \cdot \left(\frac{|\delta A|}{|A|} + \frac{|\delta b|}{|b|} \right)$$

dove osserviamo che se $\delta A = 0$ si ottiene la stessa diseguaglianza che abbiamo dimostrato col teorema 9.1.

La dimostrazione del teorema non è immediata. Ci arriviamo in 3 iterazioni successive, restringendo man mano le approssimazioni fatte.

1. Una prima stima si potrebbe avere calcolando direttamente:

   $$(A + \delta A)(x + \delta x) = (b + \delta b) \implies Ax + A\delta x + \delta A x + \delta A \delta x = b + \delta b$$

   $Ax = b$ dalle ipotesi, mentre il termine $\delta A \delta b$, il prodotto di due errori, è trascurabile, quindi:

   $$A\delta x + \delta A x \approx \delta b \implies \delta x \approx A^{-1} (\delta b - \delta A x)$$

   Passando alle norme si ha:

   $$|\delta x| \leq |A^{-1}| \, (|\delta b| + |\delta A||x|) \tag{1}$$

   da cui dividendo per $|x| \geq \frac{|A|}{|b|}$ e moltiplicando e dividendo il termine a destra per $|A|$, si ha:

   $$\frac{|\delta x|}{|x|} \leq |A^{-1}| \left(\frac{|\delta b|}{|b|} |A| + |\delta A| \frac{|A|}{|A|} \right) = |A^{-1}||A| \left(\frac{|\delta b|}{|b|} + \frac{|\delta A|}{|A|} \right) = \mu(A) \left(\frac{|\delta b|}{|b|} + \frac{|\delta A|}{|A|} \right)$$

   Da cui otteniamo essenzialmente la forma del teorema.

2. Per capire il denominatore del numero di condizionamento, che migliora il maggiorante nei casi dove A è quasi singolare, adottiamo un procedimento diverso che evidenzia l'errore fatto sull'inversa $(A+\delta A)^{-1}$ (nel calcolo precedente, starebbe nel termine $\delta A \delta x$ che abbiamo trascurato). Avremo quindi:

$$(A+\delta A)(x+\delta x) = (b+\delta b) \implies (x+\delta x) = (A+\delta A)^{-1}(b+\delta b)$$

$(A+\delta A)^{-1}$ risulta di difficile approssimazione. Riscriviamolo come:

$$(A+\delta A)^{-1} = (I + A^{-1}\delta A)^{-1}A^{-1}$$

e consideriamo la serie di Von Neumann:

$$(I-M)^{-1} = I + M + M^2 + ...$$

troncando al primo termine, si ha che possiamo riscrivere la prima inversa come:

$$(I + A^{-1}\delta A)^{-1} \approx I - A^{-1}\delta A$$

e quindi:

$$(A+\delta A)^{-1} \approx (I - A^{-1}\delta A)A^{-1} = A^{-1} - A^{-1}\delta A A^{-1} \tag{2}$$

Possiamo convincerci che l'approssimazione è valida sostituendo nella formula per $x+\delta x$ e trovando:

$$x+\delta x = (A^{-1} - A^{-1}\delta A A^{-1})(b+\delta b) = A^{-1}b + A^{-1}\delta b - A^{-1}\delta A A^{-1}b - A^{-1}\delta A A^{-1}\delta b$$

$x = A^{-1}$ dalle ipotesi, mentre il termine $A^{-1}\delta A A^{-1}\delta b$ contiene il prodotto di due errori, ed è trascurabile, quindi:

$$\delta x \approx A^{-1}\delta b - A^{-1}\delta A A^{-1}b = A^{-1}\delta b - A^{-1}\delta A x = A^{-1}(\delta b - \delta A x)$$

che è esattamente la forma della (1) che avevamo al primo metodo.

3. Riprendiamo la forma in (2) dell'inversa:

$$(A+\delta A)^{-1} \approx (I - A^{-1}\delta A)A^{-1}$$

per introdurre il denominatore che vediamo nell'enunciato del teorema. Passando alle norme, si avrà:

$$|I - A^{-1}\delta A||A^{-1}| = \frac{|A^{-1}|}{1 + |A^{-1}\delta A|}$$

Prendendo questa come la norma dell'inversa per il passaggio alle norme della (1), si ha:

$$\frac{|\delta x|}{|x|} \leq \frac{|A^{-1}|}{1+|A^{-1}\delta A|}\left(\frac{|\delta b|}{|b|}|A| + |\delta A|\frac{|A|}{|A|}\right) \leq \frac{|A^{-1}||A|}{1+|A^{-1}||\delta A|}\left(\frac{|\delta b|}{|b|} + \frac{|\delta A|}{|A|}\right)$$

dove l'$|A^{-1}||\delta A|$ al denominatore si può scrivere come $\mu(A)\frac{|\delta A|}{|A|}$, in quanto:

$$|A^{-1}||\delta A| = \mu(A)\cdot\frac{|\delta A|}{|A|} = |A||A^{-1|}\cdot\frac{|\delta A|}{|A|}$$

da cui la forma del teorema. □

1.0.2 Stima di mu

In genere è abbastanza costoso calcolare il numero di condizionamento, in quanto bisogna calcolare un inversa e quindi la sua norma. Quello che si può fare è cercarne una stima.

Ad esempio, se A è hermitiana ($A = A^H$) e si considera la norma euclidea $|\cdot|_2$, si ha che:

$$|A|_2 \cdot |A^{-1}|_2 = \rho(A) \cdot \rho(A^{-1}) = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)}$$

cioè prendiamo il rapporto fra l'autovalore più grande e l'autovalore più piccolo di A, per cui si possono usare i metodi per gli autovalori (che vedremo verso la fine del corso).

1.0.3 Stime a posteriori

Supponiamo di aver calcolato $\tilde{x} \in \mathbb{C}^n$ con un qualunque metodo di approssimazione di x, prese A e b per buone. Per valutare se $\tilde{x}$ è una *buona* approssimazione basta guardare al **vettore residuo**, cioè:

$$r = b - A\tilde{x}$$

Potremmo chiederci se, con $|r|$ piccolo, si hanno anche $|x - \tilde{x}|$ piccoli. Sottraiamo allora $Ax = b$ da r:

$$A(x - \tilde{x}) = r \implies x - \tilde{x} = A^{-1}r$$

e quindi vale la diseguaglianza:

$$|x - \tilde{x}| \leq |A^{-1}||r|$$

Usando:

$$|x| \geq \frac{|b|}{|A|}$$

si otterrà allora che:

$$\frac{|x - \tilde{x}|}{|x|} \leq \frac{|A||A^{-1}||r|}{|b|} = \mu(A) \cdot \frac{|r|}{|b|}$$

Allora, in problemi ben condizionati, avremo che $\frac{|x-\tilde{x}|}{|x|}$ errore relativo e $|r|$ sono comparabili, mentre in problemi con condizionamento $\mu(A) >> 1$ si potrebbe avere:

$$\frac{\frac{|x-\tilde{x}|}{|x|}}{|r|} \approx \mu(A)$$

1.0.4 Tecniche per l'approssimazione delle inverse

Come nota conclusiva della sezione sull'errore nei sistemi lineari, notiamo che ogni volta che c'è bisogno di risolvere una forma del tipo:

$$Ax = b \implies x = A^{-1}b$$

il metodo naive sarebbe quello di calcolare A^{-1} e moltiplicare per b. Vediamo se si può fare di meglio.

Preso $n = 1$, la forma sarà quella di una semplice equazione lineare:

$$ax = b. \implies x = \frac{b}{a}$$

cioè basta fare una sola divisione, mentre l'approccio naive risulterebbe nel:

1. Calcolare il reciproco a^{-1};
2. Moltiplicare il reciproco per $b \implies x = a^{-1} \cdot b$.

che chiaramente risulta in più passaggi, e quindi più approssimazioni intermedie e in definitiva maggiore errore.

Nel caso matriciale il metodo di divisione equivale a fare una divisione matrice-vettore (che in MATLAB si effettua come `A \ b`), di complessità $O(\frac{2}{3}n^3)$. Di contro, il metodo naive (che in MATLAB si effettua come `inv(A) * b`) avrà complessità intorno ad $O(\frac{8}{3}n^3)$, in quanto calcola la fattorizzazione LU e risolve 2n sistemi triangolari.

Inoltre, l'approccio naive è anche meno accurato, in quanto se il numero di condizionamento $\mu(A)$ è alto, l'errore di approssimazione nei passaggi intermedi potrebbe accumularsi molto di più che rispetto all'approccio della semplice divisione matrice-vettore (tanto che la stessa documentazione di MATLAB suggerisce di evitarlo).

1.1 Metodi iterativi per i sistemi lineari

Veniamo quindi a trattare i metodi iterativi per la risoluzione dei sistemi lineari.

L'idea è quella di approssimare la soluzione di un sistema lineare $Ax = b$ generando una successione di vettori $\{x^{(k)}\}_{k\in\mathbb{N}}$ tali che:

$$\lim_{k\to+\infty} x^{(k)} = x$$

La motivazione è chiaramente quella di eludere l'alta complessità di $\sim O(n^3)$ che ha la risoluzione con metodi diretti. Altra motivazione potrebbe essere quella di non conoscere direttamente A, ma solo l'applicazione:

$$v \to A \cdot v$$

(si pensi, anche se rappresenta un caso *non* lineare, ai metodi di discesa a gradiente che ottimizzano funzioni non immediatamente calcolabili o anche solo esprimibili).

Abbiamo quindi che un buon metodo iterativo dovrà:

1. Costare meno di $O(n^3)$ ad ogni passaggio (altrimenti sarebbe inutile rispetto ai metodi diretti), e quindi richiedere solo prodotti di matrici con vettori, o risoluzione di sistemi lineari favorevoli (triangolari, diagonali, ecc...);
2. Data una certa **accuratezza** posta come obiettivo, impiegare un numero ragionevole di iterazioni per raggiungerla.

1.1.1 Metodi di punto fisso

L'idea è quella di partire da $Ax - b = 0$ e di riscrivere come un'equazione equivalente in forma:

$$x = Hx + c, \quad H \in \mathbb{C}^{n\times m},\ c \in \mathbb{C}^n$$

Facciamo alcuni esempi:

1. Si sceglie una matrice $G \in \mathbb{C}^{n\times n}$ invertibile e si considera:

   $$x = x \cdot G(Ax - b) = (I - GA)x + Gb$$

   cioè:

   $$H = I - GA, \quad c = Gb$$

2. Si scompone A come:
$$A = A_1 + A_2$$
per cui:
$$Ax = b \Leftrightarrow (A_1 + A_2)c = b \Leftrightarrow A_1 x = -A_2 x + b \Leftrightarrow x = -A_1^{-1} A_2 x + A_1^{-1} b$$
cioè ancora:
$$H = -A_1^{-1} A_2, \quad c = A_1^{-1} b$$

Una volta trovata un'equazione di punto fisso $x = Hx + c$, quindi, si considera il seguente metodo iterativo:

$$\begin{cases} x^{(0)} \text{ dato} \\ x^{(k+1)} = Hx^{(k)} + c, \quad k = 1, 2, 3, ... \end{cases}$$

partendo da $x^{(0)}$ come dato a priori (sarà la soluzione che vorremo raffinare).

Vediamo che infatti:

Definizione 1.1: Matrice di iterazione

La matrice H di un equazione di punto fisso $x = Hx + c$ viene detta matrice di iterazione.

Per avere una verifica della validità dei metodi di punto fisso, enunciamo il seguente teorema:

Teorema 1.2: Validità dei metodi di punto fisso

Il metodo di punto fisso dato da:

$$\begin{cases} x^{(0)} \text{ dato} \\ x^{(n+1)} = Hx^{(k)} + c, \quad k = 1, 2, 3, ... \end{cases}$$

converge $\forall x^{(0)} \in \mathbb{C}^n$ se e solo se $\rho(H) < 1$.

Questo si dimostra prendendo l'equazione di punto fisso $x = Hx + c$, soddisfatta da x soluzione esatta, per cui:

$$x^{(k+1)} - x = Hx^{(k)} + c - (Hx + c) = H(x^{(k)} - x)$$

Possiamo chiamare $e^{(k)} = x^{(k)} - x$, cioè l'errore al passo k, e quindi l'errore al passo $k + 1$ sarà:
$$e^{(k+1)} = He^{(k)} = H^k e^{(0)}$$

Basterà allora prendere il limite:

$$\lim_{k \to +\infty} e^{(k+1)} = \lim_{k \to +\infty} H^k e^{(0)}$$

Perchè questo tenda a 0, basterà imporre $\rho(H) < 1$, in quanto in tal caso:

$$\lim_{k \to +\infty} H^k e^{(0)} = 0$$

qualsiasi sia l'errore iniziale $e^{(0)}$ (e quindi la scelta di soluzione iniziale $x^{(0)}$). □

In particolare, possiamo ricordare che se $|H| < 1 \implies \rho(H) < 1$, quindi può risultare verificare questa condizione, che è sì più stretta ma anche più facile da verificare. Ricordiamo che non vale assolutamente il contrario, cioè $|H| > 1 \not\implies \rho(H) > 1$.

Di contro, vale anche la condizione $|\det(H)| \geq 1 \implies \rho(H) > 1$, che come prima non si inverte in $|\det(H)| < 1 \not\implies \rho(H) < 1$.

1.1.2 Velocità di convergenza

Guardando alla dimostrazione del teorema 10.2, possiamo osservare che il raggio spettrale $\rho(H)$ ci dà un informazione anche riguardo alla **velocità di convergenza** del metodo di punto fisso.

Infatti, si avrà che vale:

$$\frac{|e^{(k)}|}{|e^{(0)}|} \leq |H^k|$$

Dall'algebra lineare, si ha che quando k è abbastanza grande, $H^k \approx \rho(H)^k$, almeno per norme indotte, in quanto:

$$\lim_{k \to +\infty} \sqrt[k]{|H^k|} = \rho(H)$$

Questo ci dice che se si hanno 2 metodi di punto fisso con matrici di iterazione H_1 e H_2 tali che:

$$\rho(H_1) < \rho(H_2) < 1$$

allora il primo metodo converge più velocemente del secondo, è dall'ulteriore ipotesi < 1, entrambi convergono.

Inoltre, si può stimare il numero di iterazioni k necessarie a raggiungere un certo valore dell'errore:

$$\frac{|e^{(k)}|}{|e^{(0)}|} = \frac{|x^{(k)} - x|}{|e^{(0)}|} \leq \delta$$

infatti, in tal caso basterà imporre:

$$\rho(H)^k \leq \delta \implies k \geq \frac{\log(\delta)}{\log(\rho(H))}$$

1.1.3 Criteri di stop

Molto spesso non è pratico decidere un errore e fare le k iterazioni che dovrebbero portare a tale errore (in quanto non è scontato conoscere $\rho(H)$), ma bensì si preferisce definire un **criterio di stop** per la terminazione dell'algoritmo raggiunte determinate condizioni.

Queste condizioni possono essere:

1. Si può restringere il residuo:
$$\frac{|r^{(k)}|}{|b|} < \delta$$
2. Si può restringere direttamente la variazione di errore:
$$|x^{(k+1)} - x^{(k)}| < \delta$$

1.2 Metodi di Jacobi e Gauss-Seidel

Vediamo i metodi più famosi di questo tipo, detti metodi di **Jacobi** e **Gauss-Seidel**.

L'idea è di scomporre la matrice A come:

$$A = D - E - F =$$

$$\begin{pmatrix} d_1 & 0 & ... & 0 \\ 0 & d_2 & ... & 0 \\ 0 & ... & d_{n-1} & 0 \\ 0 & ... & 0 & d_n \end{pmatrix} - \begin{pmatrix} 0 & 0 & ... & 0 \\ -e_{21} & 0 & ... & 0 \\ -e_{n-1,1} & ... & 0 & 0 \\ -e_{n1} & ... & -e_{n,n-1} & 0 \end{pmatrix} - \begin{pmatrix} 0 & -f_{12} & ... & -f_{1n} \\ 0 & 0 & ... & -f_{2n} \\ 0 & ... & 0 & -f_{n-1,n} \\ 0 & ... & 0 & 0 \end{pmatrix}$$

con D diagonale, E l'opposta della triangolare inferiore a diagonale nulla e F l'opposta della triangolare superiore a diagonale nulla. Varrà quindi:

$$Ax = b \Leftrightarrow (D - E - F)x = b$$

1.2.1 Metodo di Jacobi

Il metodo di Jacobi consiste nel riscrivere quanto trovato come:

$$Dx = (E + F)x + b \implies x = D^{-1}(E + F)x + D^{-1}b$$

cioè trovare un equazione di punto fisso con:

$$H = D^{-1}(E + F), \quad c = D^{-1}b$$

e quindi applicare l'algoritmo:

$$\begin{cases} x^{(0)} \text{ dato} \\ x^{(k+1)} = D^{-1}(E + F)x^{(k)} + D^{-1}b = D^{-1}((E + F)x^{(k)} + b) \end{cases}$$

Osserviamo che ad ogni iterazione si calcola un prodotto matrice-vettore e si risolve un sistema diagonale ($O(n)$), in quanto la D si inverte facilmente (è diagonale):

$$H_J = D^{-1}(E + F) = \begin{pmatrix} \frac{1}{d_1} & 0 & ... & 0 \\ 0 & \frac{1}{d_2} & ... & 0 \\ 0 & ... & \frac{1}{d_{n-1}} & 0 \\ 0 & ... & 0 & \frac{1}{d_n} \end{pmatrix} \begin{pmatrix} 0 & -f_{12} & ... & -f_{1n} \\ -e_{21} & 0 & ... & -f_{2n} \\ -e_{n-1,1} & ... & 0 & -f_{n-1,n} \\ -e_{n1} & ... & -e_{n,n-1} & 0 \end{pmatrix}$$

per cui:

$$(H_J)_{ij} = \begin{cases} -\frac{a_{ij}}{a_{ii}}, & i \neq j \\ 0 & i = j \end{cases}$$

Le matrici che descriveranno il passo di Jacobi saranno allora:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{i \neq j}^{n} a_{ij} x_j^{(k)} \right)$$

osservando che per calcolare $x_i^{(k+1)}$ serviranno *tutte* le componenti di $x^{(k)}$, cioè in codice bisogna mantenere due vettori, in quanto non si può sovrascrivere $x^{(k)}$ prima di aver finito di calcolare $x^{(k+1)}$.

1.2.2 Implementazione MATLAB del metodo di Jacobi

In MATLAB, una possibile implementazione che sfrutta le funzionalità di calcolo matriciale può essere la seguente:

```
function x_old = jacobi(A, b, k, x_old)
    n = height(A);

    if nargin < 4
        % stima iniziale
        x_old = zeros(n, 1);
    end

    % imposta le matrici D E F
    D = diag(A);
    EF = -A + diag(D); % D e' un vettore

    % imposta c
    c = b ./ D;

    for j = 1:k
        x_new = (EF * x_old) ./ D + c;
        x_old = x_new;

        % stampa informazioni
        fprintf("it = %d, res = %.3f\n", j, norm(b - A * x_old));
        disp(x_old);
    end
end
```

1.2.3 Implementazione C++ del metodo di Jacobi

Possiamo sfruttare la forma scalare del metodo di Jacobi per realizzarne un'implementazione direttamente in C++. Definiamo inizialmente alcune funzioni helper in un header, `mat.h`, che useremo anche nell'esempio successivo a questo:

```
#ifndef MAT_H
#define MAT_H

#include <iostream>

inline double** get_matrix(unsigned n) {
  double** A = new double*[n];

  for(int r = 0; r < n; r++) {
    A[r] = new double[n];

    for(int c = 0; c < n; c++) {
      double e;
      std::cout << "Inserisci l'entrata " << r + 1 << ", " << c + 1 << "
    di A" << std::endl;
      std::cin >> e;

      A[r][c] = e;
    }
  }

  return A;
}

```

```
inline double* get_vector(unsigned n) {
  double* b = new double[n];

  for(int r = 0; r < n; r++) {
    double e;
    std::cout << "Inserisci l'entrata " << r + 1 << " di b" << std::endl;
    std::cin >> e;

    b[r] = e;
  }

  return b;
}

inline void print_matrix(double** A, unsigned n) {
  for(int r = 0; r < n; r++) {
    for(int c = 0; c < n; c++) {
      std::cout << A[r][c] << "\t";
    }
    std::cout << std::endl;
  }
}

inline void print_vector(double* b, unsigned n) {
  for(int r = 0; r < n; r++) {
    std::cout << b[r] << std::endl;
  }
}

inline void init_env(unsigned &n, unsigned &k, double** &A, double* &b) {
  // prendi dati
  std::cout << "Inserisci la dimensione n" << std::endl;
  std::cin >> n;
  std::cout << std::endl;

  std::cout << "Inserisci il numero di iterazioni k" << std::endl;
  std::cin >> k;
  std::cout << std::endl;

  std::cout << "Inserzione della matrice A" << std::endl;
  A = get_matrix(n);
  std::cout << std::endl;

  std::cout << "Inserzione del vettore b" << std::endl;
  b = get_vector(n);
  std::cout << std::endl;

  // conferma dati
  std::cout << "La matrice A e':" << std::endl;
  print_matrix(A, n);
  std::cout << std::endl;

  std::cout << "Il vettore b e':" << std::endl;
  print_vector(b, n);
  std::cout << std::endl;
}

inline void clean_env(unsigned n, double** A, double* b) {
  // ripulisci
  for(int r = 0; r < n; r++) {
```

```
    delete[] A[r];
  }
  delete[] A;

  delete[] b;
}

inline void vec_copy(unsigned n, double* v1, double* v2) {
  for(int r = 0; r < n; r++) {
    v2[r] = v1[r];
  }
}

#endif
```

E definiamo quindi l'algoritmo come segue:

```
double* jacobi(unsigned n, unsigned k, double** A, double* b) {
  // inizializza vettori
  double* v1, *v2;
  v1 = new double[n];
  v2 = new double[n];

  for(int r = 0; r < n; r++) {
    v1[r] = 0;
  }

  for(int i = 0; i < k; i++) {
    std::cout << "Iterazione " << i + 1 << std::endl;

    std::cout << "Il vettore vecchio e':" << std::endl;
    print_vector(v1, n);

    for(int r = 0; r < n; r++) {
      v2[r] = b[r];

      for(int c = 0; c < n; c++) {
        if(r == c) continue;

        v2[r] -= A[r][c] * v1[c];
      }

      v2[r] /= A[r][r];
    }

    vec_copy(n, v2, v1);

    std::cout << "Il vettore nuovo e':" << std::endl;
    print_vector(v1, n);
    std::cout << std::endl;
  }

  // ripulisci
  delete v2;

  return v1;
}
```

A questo punto un programma dovrà semplicemente usare le funzioni per il caricamento dell'ambiente di `mat.h` e chiamare la funzione `jacobi()`:

```
int main() {
```

```
  // inizializza ambiente
  unsigned n, k; double** A; double* b;
  init_env(n, k, A, b);

  double* x = jacobi(n, k, A, b);

  std::cout << "Il risultato finale e':" << std::endl;
  print_vector(x, n);
  std::cout << std::endl;

  delete[] x;

  // ripulisci
  clean_env(n, A, b);
  return 0;
}
```

1.2.4 Metodo di Gauss-Seidel

Il metodo di Gauss-Seidel consiste nel riscrivere quanto trovato come:

$$(D-E)x = Fx + b \implies x = (D-E)^{-1}Fx + (D-E)^{-1}b$$

cioè trovare un equazione di punto fisso con:

$$H = (D-E)^{-1}F, \quad c = (D-E)^{-1}b$$

e quindi applicare l'algoritmo:

$$\begin{cases} x^{(0)} \text{ dato} \\ x^{(k+1)} = (D-E)^{-1}Fx^{(k)} + (D-E)^{-1}b \end{cases}$$

Avremo che la matrice di iterazione è:

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

di cui osserviamo che la prima colonna è il vettore di zeri in quanto F ha anch'essa la prima colonna al vettore di zeri.

In ogni caso, notiamo che non è necessario esplicitare H_{GS}, ma ad ogni iterazione basta calcolare il prodotto matrice-vettore:

$$x^{(k)} \to Fx^{(k)}$$

e risolvere col metodo di sostituzione all'indietro il sistema lineare:

$$(D-E)y = Fx^{(k)}$$

Stesso discorso per c, che si trova sempre risolvendo una sola volta, col metodo di sostituzione all'indietro, il sistema lineare:

$$(D-E)c = b$$

1.2.5 Implementazione MATLAB del metodo di Gauss-Seidel

In MATLAB, una possibile implementazione che sfrutta le funzionalità di calcolo matriciale (in particolare la funzione per la sostituzione all'indietro `bck_subst()` che abbiamo già implementato) può essere la seguente:

```
function x_old = seidel(A, b, k, x)
    n = height(A);

    if nargin < 4
        % stima iniziale
        x_old = zeros(n, 1);
    end

    % imposta le matrici D E F
    DE = triu(A);
    F = -A + DE;

    % imposta c
    c = bck_subst(DE, b);

    for j = 1:k
        x_old = bck_subst(DE, F * x_old) + c;
        x_new = x_old;

        % stampa informazioni
        fprintf("it = %d, res = %.3f\n", j, norm(b - A * x_old));
        disp(x_old);
    end
end
```

1.2.6 Ottimizzazione del metodo di Gauss-Seidel

L'implementazione vista adesso del metodo di Gauss-Seidel non è la più efficiente. Vediamo infatti che, preso:

$$x^{(k+1)} = (D-E)^{-1}Fx^{(k)} + (D-E)^{-1}b$$

e moltiplicando per $D^{-1}(D-E)$ si ha:

$$x^{(k+1)} = D^{-1}Ex^{(k+1)} + D^{-1}Fx^{(k)} + D^{-1}b$$

Esplicitare una dipendenza di $x^{(k+1)}$ da sé stessa potrebbe sembrare poco intuitivo, ma è invece conveniente in quanto ogni elemento di $x^{(k+1)}$ dipenderà dai soli elementi precedenti, cioè si avrà:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right), \quad i = 1, ..., n$$

In questo modo si hanno due vantaggi:

1. Non occorre risolvere sistemi triangolari;
2. Per calcolare $x_i^{(k+1)}$ non si ha bisogno di x_h^k per $h < i$, e quindi si possono sovrascrivere le entrate di $x^{(k)}$, e mantenere un unico vettore.

1.2.7 Implementazione C++ del metodo di Gauss-Seidel

Vediamo come si può usare quest'ultima versione (scalare) del metodo di Gauss-Seidel per realizzarne un'implementazione C++, sfruttando la stessa libreria `mat.h` dell'esempio 10.2.3:

```
double* jacobi(unsigned n, unsigned k, double** A, double* b) {
  // inizializza vettori
  double* v1;
  v1 = new double[n];

  for(int r = 0; r < n; r++) {
    v1[r] = 0;
  }

  for(int i = 0; i < k; i++) {
    std::cout << "Iterazione " << i + 1 << std::endl;

    std::cout << "Il vettore vecchio e':" << std::endl;
    print_vector(v1, n);

    for(int r = 0; r < n; r++) {
      v1[r] = b[r];

      for(int c = 0; c < n; c++) {
        if(r == c) continue;

        v1[r] -= A[r][c] * v1[c];
      }

      v1[r] /= A[r][r];
    }

    std::cout << "Il vettore nuovo e':" << std::endl;
    print_vector(v1, n);
    std::cout << std::endl;
  }

  return v1;
}
```

Come vediamo, la proprietà dimostrata comporta modifiche minime al codice (la semplice rimozione del vettore `v2`, con prodotti scalari che vengono fatti tutti su `v1`).

La realizzazione di un programma che esegue questo algoritmo è ancora analoga all'esempio 10.2.3 (implementazioni complete si trovano in `../matlab/code`).

1.2.8 Jacobi/Gauss-Seidel e matrici di permutazione

Un'ultima osservazione è che sia Jacobi che Gauss-Seidel hanno come condizione che $a_{ii} \neq 0, \forall i$, in quanto altrimenti D diagonale non sarebbe invertibile. Se questa condizione non è soddisfatta, si possono fare delle trasformazioni "indolori" sul sistema per ritrovare la diagonale non nulla (applicare matrici di permutazione e applicare il metodo sul sistema permutato, per trovare poi una soluzione che al limite andrà *de*-permutata).

Ad esempio, si può pensare di applicare Jacobi a:

$$\Pi A x = \Pi b$$

per trovare una qualche permutazione Πx di x.