SLAP
Simple Linear Algebra Package

Andrea Marchi

16 luglio 2023

Indice

1 Data type

Le matrici sono salvate in array monodimensionali *row-major*, ovvero che i dati sulle righe sono sequenziali. Quindi nella indicizzazione degli elementi della matrice con n righe e m colonne ($\mathbb{R}^{n \times m}$) si usa la formula:

```
matrix[i][j] = array[i*m + j]
```

Le righe vanno da 0 a $n-1$, mentre le colonne vanno da 0 a $m-1$.

Alcune librerie usano la notazione *column-major*, ovvero con indicizzazione `array[j*n+i]`. Avere una funzione che organizza i dati della matrice in colonne o in righe è comodo per quanto riguarda l'ottimizzazione della cache di lettura dei dati sequenziali dalla RAM alla CPU.

Il tipo di dati (essendo in C) è una struttura (`struct`) e la definizione cambia al variare del tipo di dati base (il C non permette l'uso di template). La struttura base è:

```
typedef struct _matd{
        unsigned int n_rows;
        unsigned int n_cols;
        double *data; // row-major matrix data array
} matd;
```

dove la lettera finale indica il tipo di variabile usata, in questo caso `double`. I tipi di dati che ha senso utilizzare nella libreria sono:

`d` virgola mobile a doppia precisione (`double`)
`f` virgola mobile (`float`)
`i` intero (`int`)
`b` byte (`unsigned char`)

Per allocare la memoria e liberarla (sempre liberare la memoria dopo averla allocata):

```
matd* new_matd(unsigned int num_rows, unsigned int num_cols)
{
        int i;
        // create a new double matrix
        if(num_rows == 0) { /*SLAP_ERROR(INVALID_ROWS);*/ return 0; } // dovrebbe ritornare
            NULL
        if(num_cols == 0) { /*SLAP_ERROR(INVALID_COLS);*/ return 0; } // dovrebbe ritornare
            NULL

        matd *m = calloc(1, sizeof(*m)); // allocate space for the struct
        // CONTROLLARE LA MATRICE CREATA ( SLAP_CHECK(m) )
        m->n_rows = num_rows;
        m->n_cols = num_cols;
        m->data = calloc(m->n_rows*m->n_cols, sizeof(*m->data));
        // CONTROLLARE I DATI CREATI ( SLAP_CHECK(m->data) )
        for(i=0; i<num_rows*num_cols; i++) m->data[i] = 0; // set to zero

        return m;
}

void free_mat(matd *matrix)
{
        free(matrix->data); // delete the data
        free(matrix); // delete the data structure
}
```

Come setters e getters non potendo usare le operation del C++ e non riuscendo a fare qualcosa di funzionante e decente con le macro[1] uso le funzioni

[1] Usare le macro mi permetterebbe di risparmiare tempo nella allocazione dei parametri delle funzioni. Negli algoritmi potrei usare l'accesso diretto all'array `data`.

```c
double matd_get(matd matrix, unsigned int row, unsigned int col) {
        return matrix.data[row*matrix.n_cols + col]; // row-major
}
void matd_set(matd matrix, unsigned int row, unsigned int col, double val) {
        matrix.data[row*matrix.n_cols + col] = val; // row-major
}
```

Dovrei controllare che l'accesso sia corretto (che non cerchi di scrivere/leggere dati non allocati).

2 Basic operations

Mettere un pannello (anche una tabella) che riassume le operazioni, il nome delle funzioni e come si usano.

2.1 Equality

2.2 Addition and sottration

2.3 Multiplication

2.3.1 Scalar-matrix multiplication

2.3.2 Vector-matrix multiplication

La moltiplicazione matrice-vettore può essere vista come la moltiplicazione tra una matrice e un'altra nella forma di un vettore colonna. Fare un controllo numerico su questa affermazione

2.3.3 Matrix-matrix multiplication

Il prodotto tra due matrici $\mathbf{A} \in \mathbb{R}^{n \times m}$ (matrice con n righe e m colonne) $\mathbf{B} \in \mathbb{R}^{m \times p}$ (m righe, p colonne) è una matrice $\mathbf{C} \in \mathbb{R}^{n \times p}$ le cui componenti sono definite come:

$$C_{ij} = \sum_{k=1}^{m} A_{ik} \, B_{kj} \tag{1}$$

Applicando direttamente la definizione della moltiplicazione tra matrici si ottiene un algoritmo che ha un'efficienza computazionale di $O(n^3)$ (dove n è la dimensione di una matrice quadrata $n \times n$). Migliori limiti asintotici sono stati scoperti dopo l'algoritmo di Strassen negli anni '60 (il limite teorico rimane ancora ignoto). Recentemente è stato annunciato un algoritmo con l'efficienza teorica di $O(n^{2.37188})$, ma si tratta di un *algoritmo galattico*, ovvero che richiede una dimensione talmente grande delle matrici per funzionare con tale efficienza da non essere praticamente realizzabile (in questo caso a causa di una grossa costante).

Simple algorithm: Dalla definizione della moltiplicazione tra matrici si ricava direttamente l'algoritmo 1.

Questo algoritmo richiede un tempo pari a $O(nmp)$ (se le matrici sono quadrate $n \times n$ il tempo diventa $O(n^3)$).

Cache optimization: I tre cicli all'interno della moltiplicazione possono essere scambiati senza modificare il risultato in termini matematici, tuttavia questo può avere un considerevole impatto in termini di implementazione ed in particolare in termini di tempo di esecuzione su elaboratori elettronici. Infatti i computer hanno degli algoritmi ottimizzati per l'accesso dei dati dalla RAM alla CPU quando questi sono sequenziali (uno di fila all'altro in RAM). Quando il dato successivo necessario all'algoritmo non è quello immediatamente successivo in RAM si parla di *cache miss*. La versione ottimale dell'algoritmo semplice per due matrici $\mathbf{A}$ e $\mathbf{B}$ in row-major è la versione tiled, dove le matrici sono implicitamente divise in *tiles* di dimensione $\sqrt{N} \times \sqrt{N}$, è descritta nell'algoritmo 2.

Algorithm 1 Semplice algoritmo per la moltiplicazione tra matrici

Require: matrix **A**, matrix **B**
C ← matrix of the appripriate size
for $i \leftarrow 1$ to n **do** ▷ nell'implementazione parte da 0 e arriva a $n-1$
    for $j \leftarrow 1$ to p **do** ▷ stessa considerazione dell'indice i
        $sum \leftarrow 0$
        for $k \leftarrow 1$ to m **do**
            $sum \leftarrow sum + A_{ik}\, B_{kj}$
        end for
        $C_{ij} \leftarrow sum$
    end for
end for

Algorithm 2 Semplice algoritmo per la moltiplicazione tra matrici con gestione migliorata della cache non mi torna un granche'

Require: matrix **A**, matrix **B**
C ← matrix of the appripriate size
tile size $T \leftarrow \sqrt{N}$
for $I \leftarrow 1$ to n in steps of T **do**
    for $J \leftarrow 1$ to p in steps of T **do**
        for $K \leftarrow 1$ to m in steps of T **do**
            for $i \leftarrow I$ to $\min(I+T, n)$ **do** ▷ Multiply $\mathbf{A}_{I:I+T,K:K+T}$ and $\mathbf{B}_{K:K+T,J:J+T}$ into $\mathbf{C}_{I:I+T,J:J+T}$
                for $j \leftarrow J$ to $\min(J+T, p)$ **do**
                    $sum \leftarrow 0$
                    for $k \leftarrow K$ to $\min(K+T, m)$ **do**
                        $sum \leftarrow sum + A_{ik}\, B_{kj}$
                    end for
                    $C_{ij} \leftarrow C_{ij} + sum$
                end for
            end for
        end for
    end for
end for

Nel modello di cache ideale l'algoritmo incorre in solamente $O(\frac{n^3}{b\sqrt{N}}$. Per le macchine moderne il denominatore $m\sqrt{N}$ ammonta a diversi ordini di grandezza, dunque il tempo necessario è quello effettivo del calcolo, invece di perdere tempo in *cache misses*.

Divide-and-conquer algorithm: Una alternativa è l'algoritmo *dividi e conquista* per la moltiplicazione delle matrici che fa affidamento nella partizione delle matrici in blocchi

$$\begin{bmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{12} & \mathbf{C}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{12} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{12} & \mathbf{B}_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11}\,\mathbf{B}_{11} + \mathbf{A}_{12}\,\mathbf{B}_{21} & \mathbf{A}_{11}\,\mathbf{B}_{12} + \mathbf{A}_{12}\,\mathbf{B}_{22} \\ \mathbf{A}_{21}\,\mathbf{B}_{11} + \mathbf{A}_{22}\,\mathbf{B}_{21} & \mathbf{A}_{21}\,\mathbf{B}_{12} + \mathbf{A}_{22}\,\mathbf{B}_{22} \end{bmatrix} \tag{2}$$

che si applica a tutte le matrici quadrate che hanno dimensione pari a una potenza di due ($2^n \times 2^n$). L'algoritmo dividi e conquista calcola le moltiplicazioni più piccole ricorsivamente come un prodotto scalare $C_{11} = A_{11}B_{11}$ come caso base.

Una variante che funziona per matrici non-quadrate si basa sul dividere le matrici in due invece che in quattro. In questo caso si tratta di dividere le matrici in due parti uguali, o comunque il più vicino possibile a due parti uguali nel caso di dimensioni dispari.

Algorithm 3 Moltiplicazione tra metrici tramite *divide-and-conquer*

Require: matrix $\mathbf{A} \in \mathbb{R}^{n\times m}$, matrix $\mathbf{B} \in \mathbb{R}^{m\times p}$

1: if $\max(n, m, p)$ is below some threshold, use an unrolled version of the simple algorithm
2: **if** $\max(n, m, p) = n$ **then**
3: $\quad \mathbf{C} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \end{bmatrix} \mathbf{B} = \begin{bmatrix} \mathbf{A}_1\mathbf{B} \\ \mathbf{A}_2\mathbf{B} \end{bmatrix}$ ▷ split **A** horizontally
4: **else if** $\max(n, m, p) = p$
5: $\quad \mathbf{C} = \mathbf{A} \begin{bmatrix} \mathbf{B}_1 & \mathbf{B}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{A}\mathbf{B}_1 \\ \mathbf{A}\mathbf{B}_2 \end{bmatrix}$ ▷ split **B** vertically
6: **else** ▷ $(\max(n, m, p) = m)$
7: $\quad \mathbf{C} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{A}_2 \end{bmatrix} \begin{bmatrix} \mathbf{B}_1 \\ \mathbf{B}_2 \end{bmatrix} = \mathbf{A}_1\mathbf{B}_1 + \mathbf{A}_2\mathbf{B}_2$ ▷ split **A** vertically and **B** horizontally
8: **end if**

3 Gauss elimination

L'eliminazione di Gauss consiste nell'eseguire operazioni sulla matrice che non cambiano il risultato del sistema di equazioni associato, come ad esempio *scambiare delle righe, moltiplicare righe per scalari e sommare una riga all'altra*. Queste operazioni sono eseguite per semplificare il problema.

L'eliminazione di Gauss viene usata per portare la matrice ad una forma più facile da gestire, chiamata *Row Echelon Form* (REF). In questa forma la matrice ha le seguenti proprietà:

- il primo[2] elemento non nullo di una riga (*pivot*) è esattamente 1.0
- righe che hanno tutti elementi nulli sono sotto le righe che hanno almeno un elemento non nullo
- ogni pivot di una riga si trova ad una colonna alla destra dei pivot delle righe superiori

L'algoritmo che trasforma la matrice nella *Row Echelon Form* è il seguente:

1. Trova il pivot (primo elemento non nullo della riga). Se la colonna ha elementi tutti nulli passa alla colonna successiva.
2. Scambia le righe posizionando la riga del pivot come prima.
3. Moltiplica ogni elemento sulla riga per $\frac{1}{\text{pivot}}$ in maniera tale che il pivot è pari a 1.0.
4. Tramite la somma tra righe (premoltiplicando per il pivot dell'altra riga), fa in maniera tale che tutti gli elementi sulla colonna del pivot (elementi che non sono il pivot) sono pari a zero.
5. Continua il processo finché non si è giunti alla *Row Echelon Form*.

La forma *Reduced Row Echelon Form* (RREF) consiste nell'avere solamente un valore diverso da zero per ogni colonna. Si nota che una matrice può avere molte REF, ma solamente una RREF.

[2]Primo elemento non nullo della riga partendo da sinistra verso destra.

3.1 Basic operations

Le operazioni base dell'eliminazione di Gauss sono:

- Scambio di due righe (row swapping: `matd_row_swap_r`).
- Moltiplica ogni elemento della riga per uno scalare diverso da zero (scalar multiplication of rows: `matd_row_smul_r`).
- Moltiplica una riga per in termine diverso da zero e aggiunge il risultato a un'altra riga (row addition: `matd_row_addrow_r`).

3.1.1 Row swapping

```
1  int matd_row_swap_r(matd *m, unsigned int row1, unsigned int row2)
2  {
3          // swap two rows of matrix m
4          int i;
5          double tmp;
6          if(row1 >= m->n_rows || row2 >= m->n_rows) return 0; // cannot swap rows
7          for(i=0; i<m->n_cols; i++){
8                  tmp = m->data[row2*m->n_cols+i];
9                  m->data[row2*m->n_cols+i] = m->data[row1*m->n_cols+i];
10                 m->data[row1*m->n_cols+i] = tmp;
11         }
12         return 1;
13 }
```

Esempio:

$$\texttt{matd\_row\_swap\_r} = \left(\begin{bmatrix}1 & 2 & 3\\ 0 & 2 & 4\\ 2 & 1 & 9\end{bmatrix}, 0, 1\right) = \begin{bmatrix}0 & 2 & 4\\ 1 & 2 & 3\\ 2 & 1 & 9\end{bmatrix}$$

3.1.2 Scalar multiplication of rows

```
1  int matd_row_smul_r(matd *m, unsigned int row, double num)
2  {
3          int i;
4          if(row >= m->n_rows) return 0; // cannot row multiply
5          for(i=0; i<m->n_cols; i++) m->data[row*m->n_cols+i] *= num;
6          return 1;
7  }
```

Esempio:

$$\texttt{matd\_row\_swap\_r} = \left(\begin{bmatrix}1 & 2 & 3\\ 0 & 2 & 4\\ 2 & 1 & 9\end{bmatrix}, 1, 2.0\right) = \begin{bmatrix}1 & 2 & 3\\ 0.0 & 4.0 & 6.0\\ 2 & 1 & 9\end{bmatrix}$$

3.1.3 Row addition

```
1  int matd_row_addrow_r(matd *m, unsigned int where, unsigned int row, double multiplier)
2  {
3          int i = 0;
4          if(where >= m->n_rows || row >= m->n_rows) return 0; // cannot add rows
5          for(i=0; i<m->n_cols; i++) m->data[where*m->n_cols+i] += multiplier * m->data[row*m->
               n_cols+i];
6          return 1;
7  }
```

Esempio:

$$\texttt{matd\_row\_addrow\_r} = \left(\begin{bmatrix}1 & 2 & 3\\ 0 & 2 & 4\\ 2 & 1 & 9\end{bmatrix}, 0, 1, 0.5\right) = \begin{bmatrix}1+0\cdot 0.5 & 2+2\cdot 0.5 & 3+4\cdot 0.5\\ 0 & 2 & 4\\ 4 & 1 & 9\end{bmatrix} = \begin{bmatrix}1 & 3 & 5\\ 0 & 2 & 4\\ 2 & 1 & 9\end{bmatrix}$$

4 Matrix factorizations

La fattorizzazione, o la decomposizione, di una matrice consiste nel *dividere la matrice nel prodotto di matrici più semplici*. In letteratura sono presenti varie decomposizioni applicabili a diverse classi di problemi:

Decomposizione LU: La matrice $\mathbf{A} = \mathbf{LU}$ è decomposta in una matrice triangolare inferiore $\mathbf{L}$ e una triangolare superiore $\mathbf{U}$. Applicabile a matrici quadrate.

Decomposizione QR: La matrice $\mathbf{A} = \mathbf{QR}$ é divisa in una matrice $\mathbf{Q}$ ortogonale e una $\mathbf{R}$ triangolare superiore di dimensione $n \times m$. Applicabile a matrici di dimensione qualsiasi $n \times m$.

Decomposizione di Cholesky: La matrice $\mathbf{A} = \mathbf{U}^T\mathbf{U}$ é scomposta tramite una matrice $\mathbf{U}$ triangolare superiore con elementi sulla diagonale positivi. Si applica a matrici quadrate, simmetriche, definite positive.

Decomposizione a valori singolari: La matrice $\mathbf{A} = \mathbf{UDV}^H$ é scomposta in una matrice $\mathbf{D}$ diagonale non-negativa e due matrici unitarie $\mathbf{U}$ e $\mathbf{V}$ ($\mathbf{V}^H$ denota la trasposta coniugata di $\mathbf{V}$). Applicabile a matrici di dimensione $n \times m$.

Decomposizione spettrale: La matrice $\mathbf{A} = \mathbf{VDV}^{-1}$ viene decomposta tramite la matrice $\mathbf{V}$ le cui colonne sono gli autovettori di $\mathbf{A}$ e la matrice $\mathbf{D}$ é diagonale con i corrispondenti autovalori come diagonale. Applicabile a matrici quadrate con autovettori distinti[3].

Decomposizione di Schur:

Decomposizione in componenti principali:

Decomposizione non-negativa: La matrice $\mathbf{A} = \mathbf{WH}$ è decomposta in due matrici che possiedono elementi positivi. É una soluzione, in genere di minimo locale, della funzione obiettivo $f(\mathbf{W}, \mathbf{H}) = \frac{1}{2}\|\mathbf{A} - \mathbf{WH}\|_F^2$ con i vincoli $W_{ij} \geq 0$ e $H_{hk} \geq 0$. Applicabile a matrici $n \times m$ non-negative.

4.1 LU(P) decomposition

La decomposizione LU, chiamata anche la fattorizzazione LU, si riferisce alla scomposizione di una matrice $\mathbf{A}$ in due matrici $\mathbf{L}$ e $\mathbf{U}$:

$$\mathbf{A} = \mathbf{L}\,\mathbf{U} \quad \rightarrow \quad \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix} \tag{3}$$

In pratica questa fattorizzazione viene ricavata da operazioni elementari sulle righe (come lo scambio delle righe). In tal caso bisogna introdurre nell'equazione una matrice di permutazione $\mathbf{P}$ che tiene conto del cambio delle righe:

$$\mathbf{P}\,\mathbf{A} = \mathbf{L}\,\mathbf{U} \tag{4}$$

dove $\mathbf{P}$ è una matrice ricavata dalla permutazione delle righe di una matrice identità $\mathbf{I}$, ed è calcolata dalla procedura che trova $\mathbf{L}$ e $\mathbf{U}$. Se una matrice $\mathbf{A}$ é quadrata (di dimensione $n \times n$) può sempre essere scomposta come $\mathbf{P}\,\mathbf{A} = \mathbf{L}\,\mathbf{U}$. Per trovare la decomposizione LU si usa versione modificata dell'algoritmo per l'eliminazione di Gauss. Questa implementazione richiede circa $\frac{2}{3}n^3$ operazioni a virgola mobile. Altri algoritmi richiedono funzioni ricorsive o randomizzazione.

Trovare la decomposizione (4) permette di calcolare il determinante della matrice $\mathbf{A}$, la sua inversa e, quindi, anche risolvere i sistemi di equazioni lineari.

Esiste anche un'altra fattorizzazione dove non solo le righe, ma anche le colonne vengono considerate e si chiama *LU Factorization with full pivoting*.

Descrizione dell'algoritmo e codice per la soluzione tramite decomposizione LU

Aggiungere quello che dicono su Wikipedia

[3] non necessariamente anche distinti autovalori.

4.2 QR decomposition

Qualsiasi matrice simmetrica $\mathbf{A}$ può essere decomposta come:

$$\mathbf{A} = \mathbf{Q}\,\mathbf{R} \tag{5}$$

dove $\mathbf{Q}$ è una matrice ortogonale[4] e $\mathbf{R}$ una matrice triangolare superiore. Per trovare la decomposizione (5) si usa l'algoritmo di Gram-Schmidt sicuro?.

Si può dimostrare che tutte le matrici quadrate ammettono una decomposizione QR, anche se essa non è unica. Nel caso in cui la matrice $\mathbf{A}$ sia a coefficienti complessi, allora $\mathbf{Q}$ è una matrice unitaria.

La fattorizzazione QR di una matrice richiede un tempo proporzionale a $O(n^3)$ (tramite trasformazioni di Householder o quelle di Givens). Nonostante la complessità computazionale è simile a quella della decomposizione LU (o algoritmo di Gauss) questo risulta avere una migliore stabilità numerica. Inoltre la fattorizzazione QR può essere utilizzata per il calcolo di basi ortonormali e per la risoluzione di un sistema ai minimi quadrati. La decomposizione QR viene usata anche nel *metodo QR* utilizzato per calcolare gli autovalori e autovettori di una matrice.

Al contrario della fattorizzazione LU(P) che si concentra sulle operazioni sulle righe, la decomposizione QR utilizza operazioni sulle colonne. Considerando le matrici (in questo caso di dimensione 3×3):

$$\mathbf{A} = \begin{bmatrix} | & | & | \\ \mathbf{a}_1 & \mathbf{a}_2 & \mathbf{a}_3 \\ | & | & | \end{bmatrix} \qquad \mathbf{Q} = \begin{bmatrix} | & | & | \\ \mathbf{q}_1 & \mathbf{q}_2 & \mathbf{q}_3 \\ | & | & | \end{bmatrix} \tag{7}$$

dove $\mathbf{a}_i$ sono i vettori colonna che formano la matrice $\mathbf{A}$ e $\mathbf{q}_i$ i vettori colonna di $\mathbf{B}$. La struttura della matrice $\mathbf{R}$ deriva dal fatto che $\mathbf{Q}$, essendo una matrice ortogonale, ha per colonne dei vettori unitari, ovvero di norma euclidea unitaria. Questo porta alle seguenti formule:

$$\begin{cases} \mathbf{q}_1 = \dfrac{\mathbf{a}_1}{\|\mathbf{a}_1\|} \\ \mathbf{q}_2 = \dfrac{\mathbf{a}_2^{\perp}}{\|\mathbf{a}_2^{\perp}\|} \\ \mathbf{q}_2 = \dfrac{\mathbf{a}_3^{\perp}}{\|\mathbf{a}_3^{\perp}\|} \end{cases} \qquad \text{dove} \qquad \begin{aligned} \mathbf{a}_2^{\perp} &= \mathbf{a}_2 - \langle \mathbf{a}_2, \mathbf{q}_1 \rangle \, \mathbf{q}_1 \\ \mathbf{a}_3^{\perp} &= \mathbf{a}_3 - \langle \mathbf{a}_3, \mathbf{q}_1 \rangle \, \mathbf{q}_1 - \langle \mathbf{a}_3, \mathbf{q}_2 \rangle \, \mathbf{q}_2 \end{aligned} \tag{8}$$

dove $\langle \mathbf{u}, \mathbf{v} \rangle$ è il prodotto scalare tra i due vettori $\mathbf{u}$ e $\mathbf{v}$, mentre $\|\mathbf{u}\|$ è la norma-2 euclidea. Quindi la decomposizione QR di questo esempio per matrici 3×3 è pari a:

$$\mathbf{A} = \begin{bmatrix} | & | & | \\ \mathbf{a}_1 & \mathbf{a}_2 & \mathbf{a}_3 \\ | & | & | \end{bmatrix} = \begin{bmatrix} | & | & | \\ \frac{\mathbf{a}_1}{\|\mathbf{a}_1\|} & \frac{\mathbf{a}_2^{\perp}}{\|\mathbf{a}_2^{\perp}\|} & \frac{\mathbf{a}_3^{\perp}}{\|\mathbf{a}_3^{\perp}\|} \\ | & | & | \end{bmatrix} \begin{bmatrix} \|\mathbf{a}_1\| & \langle \mathbf{a}_2, \mathbf{q}_1 \rangle & \langle \mathbf{a}_3, \mathbf{q}_1 \rangle \\ 0 & \|\mathbf{a}_2^{\perp}\| & \langle \mathbf{a}_3, \mathbf{q}_2 \rangle \\ 0 & 0 & \|\mathbf{a}_3^{\perp}\| \end{bmatrix} \tag{9}$$

Questa formula può essere generalizzata al caso $n \times n$.

[4] Una matrice si dice *ortogonale* se

$$\mathbf{Q}\,\mathbf{Q}^T = \mathbf{Q}^T\mathbf{Q} = \mathbf{I} \tag{6}$$

Questo equivale a dire che la trasposta della matrice è uguale all'inversa ($\mathbf{Q}^T = \mathbf{Q}^{-1}$).

4.3 Cholesky decomposition

4.4 Singular value decomposition (SVD)

4.5 Spectral decomposition

4.6 Schur decomposition

4.7 Principal component analysis (PCA)

4.8 Non-negative factorization

5 Solve linear systems

Risolvere il sistema di equazioni lineari:

$$\mathbf{A}\,\mathbf{x} = \mathbf{b} \tag{10}$$

vuol dire trovare il vettore colonna incognito $\mathbf{x}$ sapendo la matrice $\mathbf{A}$ e il vettore dei termini noti $\mathbf{b}$. La soluzione è trovata in maniera esatta tramite la matrice inversa di $\mathbf{A}$:

$$\mathbf{x} = \mathbf{A}^{-1}\,\mathbf{b} \tag{11}$$

Dal punto di vista dell'implementazione il principale problema risiede nel fatto che calcolare la matrice inversa$\mathbf{A}^{-1}$ attraverso il metodo di Come si chiamava? (quello lentissimo esatto) è un'operazione molto dispendiosa. Per questo esistono molti metodi per calcolare velocemente il vettore ignoto $\mathbf{x}$ in maniera computazionalmente efficiente.

5.1 Gauss-Jordan elimination

5.1.1 Backward substitution

La sostituzione all'indietro è la tecnica per la quale si calcola la soluzione del sistema lineare $\mathbf{U}\mathbf{x} = \mathbf{b}$ dove $\mathbf{U}$ è una matrice triangolare superiore:

$$\mathbf{U}\,\mathbf{x} = \mathbf{b} \quad \rightarrow \quad \begin{bmatrix} U_{11} & U_{12} & \cdots & U_{1n} \\ 0 & U_{22} & \cdots & U_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & U_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{12}$$

Si noti che l'ultimo termine del vettore $\mathbf{x}$ è facilmente calcolabile dall'ultima equazione (quella corrispondente alla n-esima riga). Questa equazione permette di trovare x_n e poi si procede *all'indietro* per trovare le altre componenti del vettore delle incognite $\mathbf{x}$. Per questo motivo è chiamato *metodo della sostituzione all'indietro*.

$$x_n = \frac{b_n}{U_{nn}} \qquad x_{n-1} = \frac{b_{n-1} - U_{(n-1)n}x_n}{U_{(n-1)(n-1)}} \qquad x_i = \frac{b_i - \sum_{k=i}^{n-1} U_{ik}x_k}{U_{ii}} \tag{13}$$

```
1  matd *lu_solvebck(matd *U, matd *b)
2  {
3          matd *x = new_matd(U->n_cols, 1);
4          int i = U->n_cols, j;
5          double tmp;
6          while(i-- > 0){
7                  tmp = b->data[i*b->n_cols];
8                  for(j=i; j<U->n_cols; j++) tmp -= U->data[i*U->n_cols+j] * x->data[j*x->n_cols
                       ];
9                  x->data[i*x->n_cols] = tmp / U->data[i*U->n_cols+i];
10         }
11         return x;
12 }
```

5.1.2 Forward substitution

Questo è il caso opposto alla sostituzione all'indietro, ovvero un metodo per trovare la soluzione del sistema lineare $\mathbf{Lx} = \mathbf{b}$ dove $\mathbf{L}$ è, al contrario del caso precedente, una matrice triangolare inferiore.

$$\mathbf{L}\,\mathbf{x} = \mathbf{b} \qquad \rightarrow \qquad \begin{bmatrix} L_{11} & 0 & \cdots & 0 \\ L_{21} & L_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ L_{n1} & L_{n2} & \cdots & L_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{14}$$

La sostituzione in avanti risulta essere, quindi, pari a:

$$x_1 = \frac{b_1}{L_{11}} \qquad x_2 = \frac{b_2 - L_{21}x_1}{L_{22}} \qquad x_i = \frac{b_i - \sum_{k=1}^{i-1} L_{ik}x_k}{L_{ii}} \tag{15}$$

```
matd *lu_solvefwd(matd *L, matd *b)
{
        matd *x = new_matd(L->n_cols, 1);
        int i,j;
        double tmp;
        for(i=0; i<L->n_cols; i++){
                tmp = b->data[i*b->n_cols];
                for(j=0; j<i; j++){
                        tmp -= L->data[i*L->n_cols+j] * x->data[j*x->n_cols];
                }
                x->data[i*x->n_cols] = tmp / L->data[i*L->n_cols+i];
        }
        return x;
}
```

5.2 LU(P) decomposition

Usando la decomposizione (4) il sistema di equazioni (10) diventa:

$$\mathbf{A}\,\mathbf{x} = \mathbf{b} \qquad \rightarrow \qquad \mathbf{P}\,\mathbf{A}\,\mathbf{x} = \mathbf{P}\,\mathbf{b} \qquad \rightarrow \qquad \mathbf{L}\;\mathbf{U}\,\mathbf{x} = \mathbf{P}\,\mathbf{b} \tag{16}$$

Introducendo il sistema di equazioni ausiliario $\mathbf{y} = \mathbf{U}\,\mathbf{x}$ troviamo la soluzione $\mathbf{x}$ del sistema lineare (10) risolvendo i due sistemi:

$$\mathbf{y} = \mathbf{U}\,\mathbf{x} \tag{17}$$

$$\mathbf{L}\,\mathbf{y} = \mathbf{P}\,\mathbf{b} \tag{18}$$

Per risolvere questi due sistemi di equazioni lineari si procede a trovare il vettore temporaneo $\mathbf{y}$ tramite sostituzione in avanti (*forward substitution*) del sistema $\mathbf{L}\,\mathbf{y} = \mathbf{P}\,\mathbf{b}$, dato che $\mathbf{L}$ è una matrice triangolare inferiore. Successivamente si trova la soluzione $\mathbf{x}$ tramite il sistema $\mathbf{U}\;\mathbf{x} = \mathbf{y}$ tramite sostituzione all'indietro[5] (*backward substitution*), dato che la matrice $\mathbf{U}$ è triangolare superiore.

```
matd *lu_solve(matd_lup *lu, matd* b)
{
        matd *Pb, *x, *y;
        if(lu->U->n_rows != b->n_rows || b->n_cols != 1) return NULL;
        Pb = matd_mul(lu->P, b);

        y = lu_solvefwd(lu->L, Pb); // Solve L*y = P*b using forward substition
        x = lu_solvebck(lu->U, y); // Solve U*x=y using backward substitution

        free_mat(y); // free memory space of the temporary matrices
        free_mat(Pb);
        return x;
}
```

[5] All'indietro, nel senso che parte trovando l'ultimo termine di $\mathbf{x}$ e va all'indietro fino al primo x_1. Il contrario rispetto alla *forward substitution*.

5.3 QR decomposition

Fattorizzata la matrice $\mathbf{A} = \mathbf{QR}$ di un sistema lineare $\mathbf{Ax} = \mathbf{b}$ la soluzione di tale sistema è data da:

$$\mathbf{x} = \mathbf{R}^{-1}(\mathbf{Q}^T\mathbf{b}) \tag{19}$$

Il calcolo del vettore temporaneo $\mathbf{y} = \mathbf{Q}^T\mathbf{b}$ richiede $O(n^2)$ operazioni per la moltiplicazione matrice-vettore, mentre il calcolo di $\mathbf{x} = \mathbf{R}^{-1}\mathbf{y}$ si può eseguire attraverso un algoritmo di sostituzione all'indietro con un tempo computazionale sempre di $O(n^2)$ operazioni. Il costo computazionale maggiore, dunque, è quello della fattorizzazione (pari a $O(n^3)$ operazioni).

6 Determinante

6.1 Formula di Leibniz

il determinante di una matrice $\mathbf{A}$ di dimensione $n \times n$ è trovato tramite la seguente formula:

$$\det(\mathbf{A}) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} A_{i\,\sigma(i)} \tag{20}$$

dove S_n è l'insieme di tutte le permutazioni σ dell'insieme $1, 2, 3, \ldots, n$, mentre $\operatorname{sgn}(\sigma)$ denota il segno della permutazione (+1 se è una permutazione pari, -1 se è dispari). Questo metodo ha un elevato costo computazionale, pari a $O(N \cdot N!)$.

6.2 Sviluppo di Laplace

Questo metodo risulta efficace solamente se la matrice contiene un gran numero di zeri. Scegliendo una riga i si calcola il determinante come:

$$\det(\mathbf{A}) = \sum_{j=1}^{n} A_{ij}\, C_{ij} \tag{21}$$

dove C_{ij} è il complemento algebrico, ovvero è pari a $(-1)^{i+j}$ per il determinante del minore di ordine $n-1$ ottenuto dalla matrice $\mathbf{A}$ eliminando la riga i-esima e la colonna j-esima. Il metodo risulta efficiente in caso si scelga una riga i che contiene il maggior numero di zeri possibile.

Lo stesso sviluppo può essere eseguito su una determinata colonna.

6.3 Algoritmo di Gauss

Nel caso di matrici triangolari (siano esse triangolari superiori o inferiori) il determinante è pari al prodotto degli elementi sulla diagonale principale. Questo deriva anche dallo sviluppo di Laplace.

7 Inverse matrix

L'algoritmo peggiore in termini computazionali per trovare la matrice inversa fa uso della matrice aggiunta

$$\mathbf{A}^{-1} = \frac{\operatorname{adj}(\mathbf{A})}{\det(\mathbf{A})} \tag{22}$$

La matrice $\operatorname{adj}(\mathbf{A})$ è calcolata tramite la trasposta della matrice dei cofattori $\operatorname{adj}(\mathbf{A}) = \operatorname{cof}(\mathbf{A})^T$. La matrice dei cofattori $\operatorname{cof}(\mathbf{A})$ ha elementi:

$$\operatorname{cof}_{ij}(\mathbf{A}) = (-1)^{i+j} \det\left(\mathbf{M}_{ij}\right) \tag{23}$$

dove $\mathbf{M}_{ij}$ è la matrice minore di $\mathbf{A}$, ottenuta eliminando la i-esima riga e la j-esima colonna della matrice $\mathbf{A}$.

8 Eigen

metodo	si applica a	produce	costo
Lanczos algorithm	Hermitian	m largest/smallest eigenpair	
Power iteration	general	eigenpair with largest value	$O(n^2)$
Inverse iteration	general	eigenpair with value closest to μ	
Rayleigh quotient iteration	hermitian	any eigenpair	
Preconditioned inverse iteration	Hermitiana	any eigenpair	
Bisection method	real symmetric tridiagonal	any eigenvalue	
Laguerre iteration	real symmetric tridiagonal	any eigenvalue	
QR algorithm	Hassenberg	all eigenvalues and all eigenpairs	$O(n^2)$ e $6n^3 + O(n^2$

8.1 QR algorithm

L'algoritmo più affidabile e maggiormente utilizzato è quello inventato da John G.F. Francis alla fine degli anni '60 che sfrutta la decomposizione QR.

L'algoritmo parte dalla matrice $\mathbf{A}$ di cui vogliamo calcolare gli autovalori e iterativamente arriva a triangolarizzarla. Allo step iniziale $k = 0$ la matrice $\mathbf{A}_0 = \mathbf{A}$ è posta pari alla matrice di input. Al generico step k si procede alla decomposizione QR $\mathbf{A}_k = \mathbf{Q}_k\mathbf{R}_k$. Quindi allo step successivo la matrice $\mathbf{A}_{k+1}$ è trovata tramite:

$$\mathbf{A}_{k+1} = \mathbf{R}_k\mathbf{Q}_k = \mathbf{Q}_k^{-1}\mathbf{Q}_k\mathbf{R}_k\mathbf{Q}_k = \mathbf{Q}_k^{-1}\mathbf{A}_k\mathbf{Q}_k = \mathbf{Q}_k^T\mathbf{A}_k\mathbf{Q}_k \tag{24}$$

Notare che tutte le matrici $\mathbf{A}_k$ sono simili e quindi hanno sempre gli stessi autovalori. L'algoritmo è numericamente stabile perché procede tramite trasformazioni simili ortogonali. Sotto certe condizioni le matrici $\mathbf{A}_k$ convergono ad una matrice triangolare, la *forma di Schur* di $\mathbf{A}$ (gli autovalori sono gli elementi sulla diagonale).

In questa forma grezza il calcolo delle iterazioni può essere computazionalmente oneroso e questo può essere ridotto trasformando inizialmente la matrice $\mathbf{A}$ in una forma di Hassenberg superiore. Questa trasformazione costa $\frac{10}{9}n^3 + O(n^2)$ operazioni aritmetiche usando una riduzione di Householder. Il calcolo della decomposizione QR di una matrice di Hassenberg superiore ha un costo di $6n^2 + O(n)$. Inoltre essendo la forma di Hassenberg una sorta di triangolare superiore questo riduce anche il numero di iterazioni per arrivare a convergenza.

A Random numbers

Algoritmi di generazione dei numeri casuali

A.1 Sampling from Gaussian distribution

metodi di sampling descritti sul mio quaderno dei ponti integrali, con qualche indicazione in piu'