

# SLAP

## Simple Linear Algebra Package

Andrea Marchi

10 luglio 2023

## Indice

<b>1</b>	<b>Data type</b>	<b>1</b>
<b>2</b>	<b>Basic operations</b>	<b>2</b>
2.1	Equality . . . . .	3
2.2	Addition and sottration . . . . .	3
2.3	Multiplication . . . . .	3
2.3.1	Scalar-matrix multiplication . . . . .	3
2.3.2	Vector-matrix multiplication . . . . .	3
2.3.3	Matrix-matrix multiplication . . . . .	3
<b>3</b>	<b>Gauss elimination</b>	<b>5</b>
3.1	Basic operations . . . . .	5
3.1.1	Row swapping . . . . .	5
3.1.2	Scalar multiplication of rows . . . . .	6
3.1.3	Row addition . . . . .	6
<b>4</b>	<b>Solve linear systems</b>	<b>6</b>
4.1	Gauss-Jordan elimination . . . . .	6
4.2	LU(P) decomposition . . . . .	6
4.3	QR decomposition . . . . .	7
<b>A</b>	<b>Random numbers</b>	<b>9</b>
A.1	Sampling from Gaussian distribution . . . . .	9

## 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:

```
1 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:

```

1 typedef struct _matd{
2     unsigned int n_rows;
3     unsigned int n_cols;
4     double *data; // row-major matrix data array
5 } 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):

```

1 matd* new_matd(unsigned int num_rows, unsigned int num_cols)
2 {
3     int i;
4     // create a new double matrix
5     if(num_rows == 0) { /*SLAP_ERROR(INVALID_ROWS);*/ return 0; } // dovrebbe ritornare
        NULL
6     if(num_cols == 0) { /*SLAP_ERROR(INVALID_COLS);*/ return 0; } // dovrebbe ritornare
        NULL
7
8     matd *m = calloc(1, sizeof(*m)); // allocate space for the struct
9     // CONTROLLARE LA MATRICE CREATA ( SLAP_CHECK(m) )
10    m->n_rows = num_rows;
11    m->n_cols = num_cols;
12    m->data = calloc(m->n_rows*m->n_cols, sizeof(*m->data));
13    // CONTROLLARE I DATI CREATI ( SLAP_CHECK(m->data) )
14    for(i=0; i<num_rows*num_cols; i++) m->data[i] = 0; // set to zero
15
16    return m;
17 }
18
19 void free_mat(matd *matrix)
20 {
21     free(matrix->data); // delete the data
22     free(matrix); // delete the data structure
23 }

```

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

```

1 double matd_get(matd matrix, unsigned int row, unsigned int col) {
2     return matrix.data[row*matrix.n_cols + col]; // row-major
3 }
4 void matd_set(matd matrix, unsigned int row, unsigned int col, double val) {
5     matrix.data[row*matrix.n_cols + col] = val; // row-major
6 }

```

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.

<sup>1</sup>Usare le macro mi permetterebbe di risparmiare tempo nella allocazione dei parametri delle funzioni. Negli algoritmi potrei usare l'accesso diretto all'array data.

## 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:

$$C_{ij} = \sum_{k=1}^m A_{ik} B_{kj} \quad (1)$$

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.

---

**Algorithm 1** Semplice algoritmo per la moltiplicazione tra matrici

---

**Require:** matrix  $\mathbf{A}$ , matrix  $\mathbf{B}$

```
1:  $\mathbf{C} \leftarrow$  matrix of the appropriate size
2: for  $i \leftarrow 1$  to  $n$  do                                ▶ nell'implementazione parte da 0 e arriva a  $n - 1$ 
3:   for  $j \leftarrow 1$  to  $p$  do                                ▶ stessa considerazione dell'indice  $i$ 
4:      $sum \leftarrow 0$ 
5:     for  $k \leftarrow 1$  to  $m$  do
6:        $sum \leftarrow sum + A_{ik} B_{kj}$ 
7:     end for
8:      $C_{ij} \leftarrow sum$ 
9:   end for
10: end for
```

---

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

---

**Require:** matrix **A**, matrix **B**

```

1: C  $\leftarrow$  matrix of the appropriate size
2: tile size  $T \leftarrow \sqrt{N}$ 
3: for  $I \leftarrow 1$  to  $n$  in steps of  $T$  do
4:   for  $J \leftarrow 1$  to  $p$  in steps of  $T$  do
5:     for  $K \leftarrow 1$  to  $m$  in steps of  $T$  do
6:       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  $\mathbf{C}_{I:I+T, J:J+T}$ 
7:         for  $j \leftarrow J$  to  $\min(J + T, p)$  do
8:            $sum \leftarrow 0$ 
9:           for  $k \leftarrow K$  to  $\min(K + T, m)$  do
10:             $sum \leftarrow sum + A_{ik} B_{kj}$ 
11:          end for
12:           $C_{ij} \leftarrow C_{ij} + sum$ 
13:        end for
14:      end for
15:    end for
16:  end for
17: 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

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

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 **A**  $\in \mathbb{R}^{n \times m}$ , matrix **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:    $\mathbf{C} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \end{bmatrix} \mathbf{B} = \begin{bmatrix} \mathbf{A}_1 \mathbf{B} \\ \mathbf{A}_2 \mathbf{B} \end{bmatrix}$  ▷ split A horizontally
4: else if  $\max(n, m, p) = p$ 
5:    $\mathbf{C} = \mathbf{A} \begin{bmatrix} \mathbf{B}_1 & \mathbf{B}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{A} \mathbf{B}_1 \\ \mathbf{A} \mathbf{B}_2 \end{bmatrix}$  ▷ split B vertically
6: else ▷ ( $\max(n, m, p) = m$ )
7:    $\mathbf{C} = \begin{bmatrix} \mathbf{A}_1 & \mathbf{A}_2 \end{bmatrix} \begin{bmatrix} \mathbf{B}_1 \\ \mathbf{B}_2 \end{bmatrix} = \mathbf{A}_1 \mathbf{B}_1 + \mathbf{A}_2 \mathbf{B}_2$  ▷ 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<sup>2</sup> 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 (premultiplicando 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.

#### 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 un 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:

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

<sup>2</sup>Primo elemento non nullo della riga partendo da sinistra verso destra.

### 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:

$$\text{matd\_row\_smul\_r} = \left( \begin{bmatrix} 1 & 2 & 3 \\ 0 & 2 & 4 \\ 2 & 1 & 9 \end{bmatrix}, 1, 2.0 \right) = \begin{bmatrix} 2 & 4 & 6 \\ 0 & 4 & 8 \\ 4 & 2 & 18 \end{bmatrix}$$

### 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->
6         n_cols+i];
7     return 1;
8 }
```

Esempio:

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

## 4 Solve linear systems

Risolvere il sistema di equazioni lineari:

$$\mathbf{A} \mathbf{x} = \mathbf{b} \quad (3)$$

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  $\mathbf{A}$ :

$$\mathbf{x} = \mathbf{A}^{-1} \mathbf{b} \quad (4)$$

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.

### 4.1 Gauss-Jordan elimination

### 4.2 LU(P) decomposition

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

$$\mathbf{A} = \mathbf{L} \mathbf{U} \quad \rightarrow \quad \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ 0 & U_{22} & U_{23} \\ 0 & 0 & U_{33} \end{bmatrix} \quad (5)$$

In pratica questa fattorializzazione 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:

$$\mathbf{P} \mathbf{A} = \mathbf{L} \mathbf{U} \quad (6)$$

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  $\mathbf{A}$  è 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 (6) 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 fattorializzazione dove non solo le righe, ma anche le colonne vengono considerate e si chiama *LU Factorialization with full pivoting*.

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

Usando la decomposizione (6) il sistema di equazioni (3) diventa:

$$\mathbf{A} \mathbf{x} = \mathbf{b} \quad \rightarrow \quad \mathbf{P} \mathbf{A} \mathbf{x} = \mathbf{P} \mathbf{b} \quad \rightarrow \quad \mathbf{L} \mathbf{U} \mathbf{x} = \mathbf{P} \mathbf{b} \quad (7)$$

Introducendo il sistema di equazioni ausiliario  $\mathbf{y} = \mathbf{U} \mathbf{x}$  troviamo la soluzione  $\mathbf{x}$  del sistema lineare (3) risolvendo i due sistemi:

$$\mathbf{y} = \mathbf{U} \mathbf{x} \quad (8)$$

$$\mathbf{L} \mathbf{y} = \mathbf{P} \mathbf{b} \quad (9)$$

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<sup>3</sup> (*backward substitution*), dato che la matrice  $\mathbf{U}$  è triangolare superiore.

```

1 matd *lu_solve(matd_lup *lu, matd* b)
2 {
3     matd *Pb, *x, *y;
4     if(lu->U->n_rows != b->n_rows || b->n_cols != 1) return NULL;
5     Pb = matd_mul(lu->P, b);
6
7     y = lu_solvefwd(lu->L, Pb); // Solve L*y = P*b using forward substitution
8     x = lu_solvebck(lu->U, y); // Solve U*x=y using backward substitution
9
10    free_mat(y); // free memory space of the temporary matrices
11    free_mat(Pb);
12    return x;
13 }
```

### 4.3 QR decomposition

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

$$\mathbf{A} = \mathbf{Q} \mathbf{R} \quad (10)$$

dove  $\mathbf{Q}$  è una matrice ortogonale<sup>4</sup> e  $\mathbf{R}$  una matrice triangolare superiore. Per trovare la decomposizione (10) si usa l'algoritmo di Gram-Schmidt.

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

$$\mathbf{A} = \begin{bmatrix} | & | & | \\ \mathbf{a}_1 & \mathbf{a}_2 & \mathbf{a}_3 \\ | & | & | \end{bmatrix} \quad \mathbf{Q} = \begin{bmatrix} | & | & | \\ \mathbf{q}_1 & \mathbf{q}_2 & \mathbf{q}_3 \\ | & | & | \end{bmatrix} \quad (12)$$

<sup>3</sup>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*.

<sup>4</sup>Una matrice si dice *ortogonale* se

$$\mathbf{Q} \mathbf{Q}^T = \mathbf{Q}^T \mathbf{Q} = \mathbf{I} \quad (11)$$

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

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:

$$\left\{ \begin{array}{l} \mathbf{q}_1 = \frac{\mathbf{a}_1}{\|\mathbf{a}_1\|} \\ \mathbf{q}_2 = \frac{\mathbf{a}_2^\perp}{\|\mathbf{a}_2^\perp\|} \\ \mathbf{q}_3 = \frac{\mathbf{a}_3^\perp}{\|\mathbf{a}_3^\perp\|} \end{array} \right. \quad \text{dove} \quad \begin{array}{l} \mathbf{a}_2^\perp = \mathbf{a}_2 - \langle \mathbf{a}_2, \mathbf{q}_1 \rangle \mathbf{q}_1 \\ \mathbf{a}_3^\perp = \mathbf{a}_3 - \langle \mathbf{a}_3, \mathbf{q}_1 \rangle \mathbf{q}_1 - \langle \mathbf{a}_3, \mathbf{q}_2 \rangle \mathbf{q}_2 \end{array} \quad (13)$$

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  $3 \times 3$  è pari a:

$$\mathbf{A} = \begin{bmatrix} | & | & | \\ \mathbf{a}_1 & \mathbf{a}_2 & \mathbf{a}_3 \\ | & | & | \end{bmatrix} = \begin{bmatrix} | & | & | \\ \frac{\mathbf{a}_1}{\|\mathbf{a}_1\|} & \frac{\mathbf{a}_2^\perp}{\|\mathbf{a}_2^\perp\|} & \frac{\mathbf{a}_3^\perp}{\|\mathbf{a}_3^\perp\|} \\ | & | & | \end{bmatrix} \begin{bmatrix} \|\mathbf{a}_1\| & \langle \mathbf{a}_2, \mathbf{q}_1 \rangle & \langle \mathbf{a}_3, \mathbf{q}_1 \rangle \\ 0 & \|\mathbf{a}_2^\perp\| & \langle \mathbf{a}_3, \mathbf{q}_2 \rangle \\ 0 & 0 & \|\mathbf{a}_3^\perp\| \end{bmatrix} \quad (14)$$

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



## **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'