

# **Elaborato di Fisica Computazionale**

**A.A 2024/2025**

Andrea Rossi N. 897139

October 29, 2024



# Prefazione

Nel seguente elaborato verrà illustrata l'analisi degli argomenti proposti nel corso di Fisica Computazionale.

Di seguito sono riportate le informazioni richieste per una fruizione completa del documento:

**Codice sorgente e dati** Il codice sorgente ed i dati analizzati nei vari esercizi sono reperibili al seguente [repository Github](#) nelle cartelle dei capitoli omonimi.

**Lingua del codice** La lingua utilizzata nel codice sorgente sarà l'inglese per avere una maggiore coesione sintattica con i linguaggi di programmazione utilizzati, la spiegazione generale del codice sarà presente nell'elaborato, per ulteriori informazioni riferirsi alla documentazione, presente nel repository.

**Introduzione dei moduli** Verranno ripetute, soprattutto nella parte introduttiva dei vari capitoli, i punti chiave recuperati dalle risorse disponibili sull'e-learning del corso. Esse saranno riassuntive favorendo le precisazioni e lo studio degli esercizi per ottenere un quadro completo dei vari argomenti.



# Contents

|  |            |
|--|------------|
| <b>Preface</b>   | <b>iii</b> |
| <b>Contents</b>  | <b>v</b>   |
| <b>1 Numeri</b>  | <b>1</b>   |
| 1.1 Rappresentazione . . . . .                                       | 1          |
| 1.2 Esercizi . . . . .   | 1          |
| 1.2.1 Precisione . . . . .   | 1          |
| 1.2.2 Propagazione degli errori . . . . .                            | 2          |
| <b>2 Approssimazioni</b>   | <b>5</b>   |
| 2.1 Introduzione . . . . .   | 5          |
| 2.2 Esercizi . . . . .   | 5          |
| 2.2.1 Funzione esponenziale . . . . .                                | 5          |
| 2.2.2 Problema di Basilea . . . . .                                  | 6          |
| <b>3 Matrici</b>   | <b>9</b>   |
| 3.1 Introduzione . . . . .   | 9          |
| 3.2 Esercizi . . . . .   | 9          |
| 3.2.1 Soluzione di sistemi lineari con matrici triangolari . . . . . | 9          |
| 3.2.2 Eliminazione di Gauss . . . . .                                | 11         |
| 3.2.3 Decomposizione LU . . . . .                                    | 13         |
| <b>4 Intepolazione</b>   | <b>17</b>  |
| 4.1 Implementazione preliminaria . . . . .                           | 17         |
| 4.2 Esercizi . . . . .   | 17         |
| 4.2.1 Metodo diretto e polinomio di Newton . . . . .                 | 17         |
| 4.2.2 Funzione di Runge . . . . .                                    | 19         |
| 4.2.3 Esercizio 3? . . . . .   | 19         |
| <b>5 ODE: Equazioni differenziali ordinarie</b>                      | <b>21</b>  |
| 5.1 Esercizi . . . . .   | 21         |
| 5.1.1 Oscillatore approssimato . . . . .                             | 21         |
| 5.1.2 Oscillatore reale . . . . .                                    | 21         |
| 5.1.3 Attrattore di Lorenz . . . . .                                 | 22         |
| 5.1.4 Sistema a tre corpi . . . . .                                  | 22         |



## 1.1 Rappresentazione

La rappresentazione numerica a cui il calcolo scientifico fa riferimento principalmente è quella dei numeri reali; nell'ambito informatico tale rappresentazione utilizza il concetto di numeri a virgola mobile come standard: i numeri reali vengono rappresentati attraverso una notazione scientifica in base due tramite la seguente formula:

$$(-1)^S \left( 1 + \sum_n M_n 2^{-n} \right) \cdot 2^E$$

Dove:

$S$  è il valore booleano per il **segno**

$M$  è la parte decimale detta **mantissa**

$E = e - d$  è l'**esponente** con  $d$  (offset),  $e$  (esponente dopo offset)

## 1.2 Esercizi

### 1.2.1 Precisione

#### Nozioni teoriche

**Definizione** La *precisione di macchina* (o  *$\epsilon$  di macchina*) è la differenza tra 1 e il numero successivo rappresentabile dato il numero di bit richiesti, esso sarà dunque:

$$\epsilon = 2^{-M}$$

Nello standard dei numeri a virgola mobile (IEEE 754) si studiano principalmente due sottoclassi di numeri i cui nominativi nei linguaggi C-like sono:

**float** numero a singola precisione (32 bit di memoria):

- ▶  $M$ : 23 bit
- ▶  $E$ : 8 bit
- ▶ Valore massimo:  $3.40 \cdot 10^{38}$
- ▶  $\epsilon$ :  $\sim 10^{-7}$

**double** numero a doppia precisione (64 bit di memoria):

- ▶  $M$ : 52 bit
- ▶  $E$ : 11 bit
- ▶ Valore massimo:  $1.8 \cdot 10^{308}$
- ▶  $\epsilon$ :  $\sim 10^{-16}$

|   |   |
|---|---|
| 1.1 Rappresentazione . . . . .            | 1 |
| 1.2 Esercizi . . . . .                    | 1 |
| 1.2.1 Precisione . . . . .                | 1 |
| 1.2.2 Propagazione degli errori . . . . . | 2 |

**Richiesta** Scriverete un programma C che esegua le seguenti operazioni:

```

1  define f in single precision = 1.2e34
2  for loop with 24 cycles:
3      f *= 2
4      print f in scientific notation
5  repeat for d in double precision, starting from 1.2e304
6  define d in double precision = 1e-13
7  for loop with 24 cycles:
8      d /= 2
9      print d and 1+d in scientific notation
10 repeat for single precision

```

Esaminare il range minimo e massimo e il ruolo dell'errore di macchina.

### Implementazione e osservazioni

**File necessari** sorgente: number\_precision.c, dati: number\_precision.dat

1: Per formattare il codice secondo la richiesta del problema si usi la definizione `EXERCISE_FORMAT`, altrimenti verrà utilizzata una formattazione più compatta per leggere in maniera più diretta i dati, si consiglia di utilizzare quest'ultima per comprendere l'analisi sottostante

Il codice sorgente scritto utilizza funzionalità base del linguaggio C. <sup>1</sup>

**Analisi e conclusioni** Dai dati ottenuti si possono notare in maniera esaustiva varie proprietà dei numeri a virgola mobile:

1. Esiste un *valore massimo* sia per singola ( $\sim 3 \cdot 10^{38}$ ) sia per doppia precisione ( $\sim 2 \cdot 10^{308}$ ), superato esso viene mostrato un valore esatto *inf* definito dallo standard descritto in precedenza;
2. I numeri hanno un *errore macchina* dettato dalla capienza di memoria della mantissa;
3. Come mostrerà più precisamente la prossima sezione, l'errore viene *propagato* nella somma:
  - $1 + f_{mult}$  perde completamente l'informazione su  $f_{mult}$
  - $1 + d_{mult}$  la conserva soltanto per le prime iterazioni;

### 1.2.2 Propagazione degli errori

**Nozioni teoriche** E' immediato notare come i numeri a virgola mobile possano essere rappresentati come variabili casuali con errore associato, derivante dalla precisione di macchina.

Prendiamo in esame una funzione  $f(x, y)$  dove  $x, y$  sono variabili casuali indipendenti con rispettivo errore  $\sigma_x, \sigma_y$ , allora l'errore su  $f$  sarà:

$$\sigma_f^2 = \left( \frac{\partial f}{\partial x} \right)^2 \sigma_x^2 + \left( \frac{\partial f}{\partial y} \right)^2 \sigma_y^2$$

Assumendo ora  $f = x + y$  otteniamo:

$$\sigma_f^2 = \sigma_x^2 + \sigma_y^2$$

Notiamo immediatamente quindi che se  $x \gg y$  allora  $\sigma_f \approx \sigma_x$  quindi si perde l'informazione su  $y$  nella somma.



**Richiesta** Si scriva in C un programma che esegua le seguenti operazioni:

```

1  calculate (0.7 + 0.1) + 0.3 and print 16 digits
2  calculate 0.7 + (0.1 + 0.3) and print 16 digits
3  define xt = 1.e20; yt = -1.e20; zt = 1
4  calculate (xt + yt) + zt
5  calculate xt + (yt + zt)

```

Esaminare la non-associatività dell'addizione e il ruolo degli errori di arrotondamento.

### Implementazione e osservazioni

**File necessari** sorgente: error\_propagation.c

In base alle richieste l'output è il seguente:

1.  $(0.7 + 0.1) + 0.3 \stackrel{?}{=} 0.7 + (0.1 + 0.3)$ :

Output: 1.10000000238418579, 1.10000000238418579

La somma risulta associativa.

2.  $[10^{20} + (-10^{20})] + 1 \stackrel{?}{=} 10^{20} + [(-10^{20}) + 1]$ :

Output: 1.0000000000000000, 0.0000000000000000

La somma risulta non associativa.

**Analisi** Utilizzando le formule discusse si può studiare la propagazione dell'errore nella somma. In essa la propagazione dipende dall'errore assoluto dei singoli addendi. Assumendo numeri a singola precisione e ricordando che  $\sigma_x \approx \epsilon \sim 10^{-7}$ , si ottengono i seguenti casi:

1. Per i valori 0.7, 0.1, 0.3 l'ordine di grandezza è lo stesso, quindi, tutti i valori posseggono un errore assoluto  $\Delta x \sim 10^{-8}$ ; propagando l'errore nella somma si ottiene dunque  $\Delta_{output} \sim 3 \cdot \Delta x$  in accordo con i risultati.
2. Il risultato è descrivibile come un caso limite nell'errore di propagazione rispetto alla singola precisione, infatti,  $10^{20}$  avrà un errore assoluto di  $\sim 10^{13}$  mentre 1 di  $10^{-7}$ !

La spiegazione dell'output ottenuto, dunque, si basa sulla differenza tra ordini di grandezza dei diversi addendi:

- Nel termine a sinistra vengono sommati prima numeri con errore assoluto paragonabile. Si ottiene quindi  $\sim 0$  che sarà poi sommato con un numero avente errore assoluto simile a 1.
- Nel termine a destra, invece, si sommano due valori con venti ordini di grandezza di differenza: l'errore assoluto di  $10^{20}$  prevale e si perde qualsiasi informazione nella somma per termini:

$$x \ll 10^{20} \Rightarrow x + 10^{20} \sim 10^{20}$$

Segue che 1 sarà ignorato nella somma a destra.

**Conclusioni** Nel manipolare numeri in un calcolatore l'operazione eseguita, la precisione e la differenza in ordine di grandezza dei numeri partecipanti devono essere tenuti sempre in considerazione specialmente nelle addizioni.

## 2.1 Introduzione

Le approssimazioni di funzioni rivestono un ruolo fondamentale in ambito scientifico, specialmente nella fisica computazionale, dove spesso non è possibile risolvere esattamente le equazioni che descrivono i fenomeni fisici.

Queste approssimazioni permettono di semplificare funzioni complesse attraverso metodi numerici, rendendo più accessibile la loro analisi e il calcolo delle soluzioni. Tali tecniche consentono di ottenere stime accurate di grandezze fisiche che altrimenti sarebbero difficili da trattare analiticamente, con una precisione dipendente dalla complessità del modello e dalla quantità di risorse computazionali disponibili.

Alcuni dei metodi più comuni includono l'approssimazione polinomiale: tra cui le serie di Taylor, argomento di questa sezione; o tecniche come l'interpolazione che sarà l'argomento della sezione pertinente.

|       |                             |   |
|-------|-----------------------------|---|
| 2.1   | Introduzione . . . . .      | 5 |
| 2.2   | Esercizi . . . . .          | 5 |
| 2.2.1 | Funzione esponenziale . . . | 5 |
| 2.2.2 | Problema di Basilea . . .   | 6 |

## 2.2 Esercizi

### 2.2.1 Funzione esponenziale

**Nozioni teoriche** La funzione esponenziale è esprimibile in serie di McLaurin come:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} + \epsilon$$

dove  $\epsilon$  rappresenta l'errore commesso nell'approssimare la funzione:

$$\epsilon \approx \frac{x^{n+1}}{(n+1)!}$$

ove  $n$  è il grado della serie di Taylor.

**Richiesta** Si consideri la funzione  $f(x) = \exp(x)$  nell'intervallo  $x \in [0.1, 1]$ .

Scrivere un programma in C che calcoli il fnzionale di approssimazione corrispondente (in doppia precisione)

$$g_n(x) = \sum_{i=0}^n \frac{x^i}{i!}$$

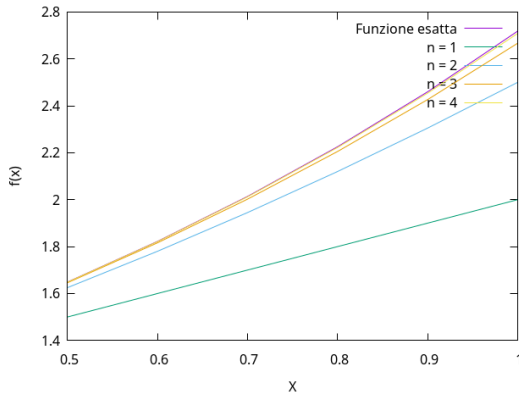
1. Verificare che l'errore scala approssimativamente come  $\frac{x^{n+1}}{(n+1)!}$  per  $n = 1, 2, 3, 4$ .

2. L'errore  $|f(x) - g(x)|$ , nell'intervallo dato in  $x$ , differisce da  $\frac{x^{n+1}}{(n+1)!}$ : perché e per quali valori di  $x$ ?

### Implementazione e osservazioni

**File necessari** sorgente: `exp_approx.c`, dati: `exp_approx_n.dat`  
 $n = 1, 2, 3, 4$

La funzione esponenziale ottenuta approssimando si può osservare in Figure 2.1.

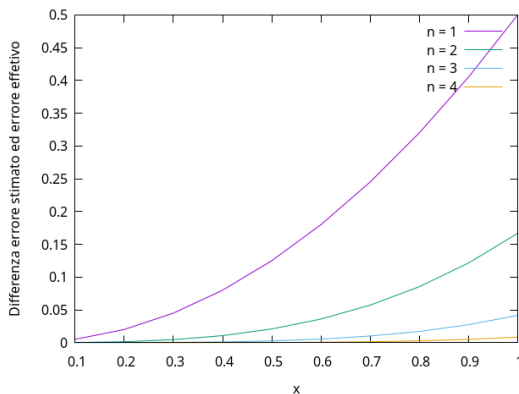


**Figure 2.1:** Confronto tra funzione esponenziale e la  $n$ -esima approssimazione

1. L'errore scala effettivamente come  $\frac{x^{n+1}}{(n+1)!}$  per valori vicini a 0 e per valori di  $n$  maggiori, come si può osservare in Figure 2.2.
2. Sempre dalla Figure 2.2 si può notare come l'errore aumenti all'allontanarsi da  $x=0$  e cominci a differire dall'errore teorico. Ciò si intuisce studiando la condizione per cui continui  $\epsilon_{teorico} \approx \epsilon_{reale}$  è soddisfatta:

$$O(x^n) \rightarrow o(x^{n+1}) \rightarrow \epsilon_{teorico} \ll 1$$

La condizione dipende da  $n$  e  $x$ : per valori bassi di  $n$ ,  $x^n$  prevale sul fattoriale e la differenza da  $\epsilon_{teorico}$  è maggiormente visibile. Come si può vedere in Figure 2.2 per  $n = 4$  e l'errore tende a 0 molto più velocemente.



**Figure 2.2:** Differenza tra errore teorico ed errore ottenuto

### 2.2.2 Problema di Basilea

**Nozioni teoriche** Il problema di Basilea consiste nel calcolare il valore della serie armonica generalizzata:

$$S(N) = \sum_{n=1}^N \frac{1}{n^2} \xrightarrow{N \rightarrow \infty} \zeta(2) = \frac{\pi^2}{6}$$

**Richiesta** Calcolare la seguente somma

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \frac{\pi^2}{6} = \lim_{N \rightarrow \infty} S(N)$$

con

$$S(N) = \sum_{n=1}^N \frac{1}{n^2}$$

1. Calcolare la somma in precisione singola usando l'ordine normale  $n = 1, 2, \dots, N$
2. Calcolare la somma in precisione singola usando l'ordine inverso  $n = N, \dots, 2, 1$
3. Studiare la convergenza di entrambe in funzione di  $N$ , tracciando  $\left|S(N) - \frac{\pi^2}{6}\right|$
4. Ripetere i punti 1, 2 e 3 in precisione doppia

## Osservazioni e conclusioni

**Singola precisione** I valori ottenuti per numeri a singola precisione sono:

$$S_{incr}(N) \approx 1.6447253 \quad \text{per } N = 6000$$

$$S_{inv}(N) \approx 1.6447674 \quad \text{per } N = 6000$$

Il risultato è spiegabile in maniera equivalente a 1.2.2: l'ordine della somma conta nella propagazione di errori in numeri a virgola mobile. Infatti:

- Nella somma incrementale, il valore di partenza è  $1/1^2 = 1$  (il valore *più grande* della somma con errore  $\epsilon \approx 10^{-7}$ ), dato che la somma si propaga con gli errori assoluti degli addendi, considerando  $N = 4000$ :

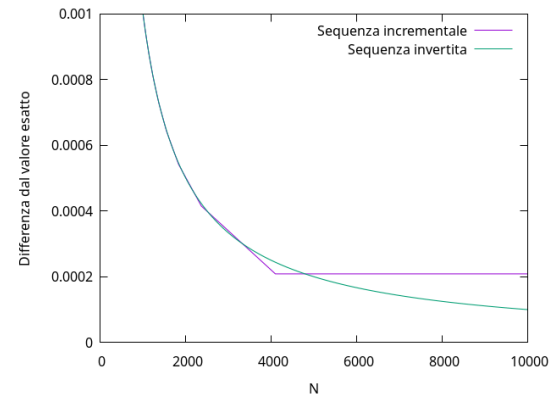
$$\epsilon_{tot} \approx N\epsilon \approx 4 \cdot 10^{-4}$$

Che è circa lo stesso ordine di grandezza in cui inizia l'andamento costante della somma. Per  $N > 4000$  la somma perde le informazioni su numeri piccoli poichè l'errore propagato è maggiore del valore sommato.

- La somma invertita, invece, inizia con il valore più piccolo della serie, per esempio  $1/4000^2 \approx 6.25 \cdot 10^{-8}$ , e propaga con errori sempre maggiori ma inferiori alle cifre significative del valore successivo (più grande). Ciò comporta che la perdita di informazioni non è abbastanza significativa per causare errori di arrotondamento notevoli.

**Doppia precisione** In doppia precisione l'errore macchina è ancora minore e l'effetto diventa trascurabile subentrano ulteriori errori (per numeri di  $5 \cdot 10^5$ ) non causati dalla precisione del numero ma da limiti del programma scritto o del compilatore/interprete utilizzato.

In conclusione, si sottolinea, come nel capitolo precedente, l'importanza di considerare l'ordine della somma e la precisione del calcolo in problemi numerici.



**Figure 2.3:**  $|S(N) - \pi^6/2|$  per valori grandi di  $N$ , si nota un singolare andamento della somma a partire da valori  $x \approx 4000$ , **la stessa cosa non succede invece per numeri a doppia precisione**



### 3.1 Introduzione

**Struttura del codice** Da questo capitolo in poi, il codice sorgente utilizzerà come linguaggio primario C++. Le librerie necessarie prima di proseguire sono le seguenti:

- ▶ `tensor.hpp` versione modificata di `matrix.h` disponibile su e-learning: l'header è stato generalizzato per interpretare sia vettori sia matrici rendendo le operazioni compatibili fra i due e facilitando il successivo svolgimento degli esercizi.
- ▶ `tensor_utils.hpp` contenente varie funzionalità utili e gli algoritmi creati per ogni esercizio.

Le cartelle corrispettive dei vari esercizi conterranno solo la richiesta e i dati proposti, mentre le funzionalità interne degli algoritmi verranno implementate principalmente in `tensor_utils.hpp` per facilitare il riutilizzo nei moduli successivi.

### 3.2 Esercizi

#### 3.2.1 Soluzione di sistemi lineari con matrici triangolari

**Nozioni teoriche** I sistemi lineari con matrici triangolari sono la tipologia più semplice da risolvere.

Presa una matrice  $A$  triangolare superiore:

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ 0 & a_{11} & a_{12} \\ 0 & 0 & a_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}$$

Otteniamo immediatamente

$$x_2 = \frac{b_2}{a_{22}}$$

si sostituisce ora ricorsivamente  $x_2$  nella seconda equazione e si ottiene

$$x_1 = \frac{b_1 - a_{12}x_2}{a_{11}}$$

e così via. Si ottiene quindi in generale la seguente formula, detta di *Backward substitution*:

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

|       |  |    |
|-------|--|----|
| 3.1   | Introduzione . . . . .   | 9  |
| 3.2   | Esercizi . . . . .   | 9  |
| 3.2.1 | Soluzione di sistemi<br>lineari con matrici<br>triangolari . . . . . | 9  |
| 3.2.2 | Eliminazione di Gauss . . .  | 11 |
| 3.2.3 | Decomposizione LU . . .  | 13 |

**Richiesta** Scrivere una funzione che accetti una matrice triangolare superiore e implementi la *Backward substitution*.

Risolvere il sistema lineare  $U\vec{x} = \vec{b}$  per

$$U = \begin{pmatrix} 2 & 1 & 1 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{pmatrix} \quad \vec{b} = (1, -1, 4)$$

Verificare che la soluzione sia corretta.

**Implementazione e stile di struttura tipica del codice** La funzione può essere trovata in *tensor\_utils.hpp*.

```

1 |
2 | /*
3 | * Viene utilizzato T generico per rappresentare la precisione
4 | * delle entrate matriciali, riferirsi alla documentazione di
5 | * Tensor per ulteriori informazioni (tensor.hpp)
6 | *
7 | * Grazie alla generalizzazione a tensore b viene rappresentata
8 | * anche come matrice se necessario
9 | */
10 | template <typename T> Tensor<T> BackwardSubstitution(Tensor<T>
    const &A, Tensor<T> const&b)
11 | {
12 |     /*
13 |     * Negli algoritmi saranno presenti vari assert a fini di
14 |     * debugging, si aggiunge il parametro -DNDEBUG durante la
15 |     * compilazione per evitare questi ulteriori controlli
16 |     */
17 |
18 |     assert(A.Cols() == A.Rows());
19 |     assert(A.Rows() == b.Rows());
20 |     assert(IsUpperTriangular(A));
21 |
22 |     ... // Definizione delle variabili
23 |
24 |     /*
25 |     * Si itera rispetto alle colonne di b
26 |     * si risolve il sistema per ogni colonna
27 |     * utile per il calcolo della matrice inversa
28 |     * nei prossimi esercizi
29 |     */
30 |     for (int k = 0; k < b.Cols(); k++)
31 |     {
32 |         // Formula di backward substitution citata precedentemente
33 |         for (int i = N - 1; i >= 0; i--)
34 |         {
35 |             sum = 0;
36 |             for (int j = i + 1; j <= N - 1; j++)
37 |             {
38 |                 sum += A(i, j) * solution(j, k);
39 |             }
40 |
41 |             solution(i, k) = (b(i, k) - sum) / A(i, i);
42 |         }

```



```

43     }
44
45     return solution;
46 }

```

### Analisi risultati

**File necessari** backward\_subst.cpp

**Risultato e controllo** Inserendo  $U$  e  $b$  proposti dall'esercizio si ottiene il seguente risultato:

$$\begin{bmatrix} 2 & 1 & 1 \\ 0 & 1 & -2 \\ 0 & 0 & 1 \end{bmatrix} \mathbf{x} = \begin{bmatrix} 1 \\ -1 \\ 4 \end{bmatrix} \Rightarrow \mathbf{x} = \begin{bmatrix} -5 \\ 7 \\ 4 \end{bmatrix}$$

Per controllare il risultato basta moltiplicare  $U$  per il risultato ottenuto e verificare che sia uguale a  $b$ .

### 3.2.2 Eliminazione di Gauss

**Nozioni teoriche** Il metodo di eliminazione di gauss utilizza le operazioni elementari delle matrici le quali lasciano invariate le soluzioni del sistema lineare. L'idea è quella di ridurre la matrice  $A$  in una matrice triangolare superiore  $U$  e di applicare a  $b$  le stesse operazioni elementari. Successivamente è possibile applicare la backward substitution per trovare la soluzione del sistema.

Prendiamo una matrice  $A$  generica  $3 \times 3$  senza perdere di generalità:

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \end{bmatrix}$$

possiamo applicare le seguenti operazioni elementari, ottenendo

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ 0 & a'_{11} & a'_{12} \\ 0 & a'_{21} & a'_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_0 \\ b'_1 \\ b'_2 \end{bmatrix}$$

iterando il processo si ottiene la matrice  $U$  triangolare superiore e la matrice  $b$ :

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} \\ 0 & a'_{11} & a'_{12} \\ 0 & 0 & a''_{22} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_0 \\ b'_1 \\ b''_2 \end{bmatrix}$$

In generale si otterrà che per  $a$  e dunque per  $b$ :

$$a_{ij} = a_{ij} - \frac{a_{ik}}{a_{kk}} a_{kj} \quad b_i = b_i - \frac{a_{ik}}{a_{kk}} b_k$$

Estendendo  $b$  ad una matrice (e quindi  $X$ ), la formula diventa:

$$b_{ij} = b_{ij} - \underbrace{\frac{a_{ik}}{a_{kk}}}_{\lambda} b_{kj}$$

Questo risultato sarà utile per calcolare l'inversa di una matrice. ( $\vec{b} \rightarrow \mathbb{I}$ )

**Richiesta** Scrivere una funzione dedicata per la risoluzione di sistemi lineari tramite eliminazione di Gauss, applicando la backward substitution successivamente.

Risolvere il seguente sistema lineare:

$$\begin{bmatrix} 2 & 1 & 1 \\ 1 & 1 & -2 \\ 1 & 2 & 1 \end{bmatrix} \mathbf{X} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} \Rightarrow \mathbf{X} = \begin{bmatrix} 8 \\ -2 \\ 2 \end{bmatrix}$$

e controllare la soluzione.

**Implementazione** Evitando verbosità la parte fondamentale del codice è la seguente:

```

1      ... // Asserts e definizioni
2
3      // Utilizziamo le formule sopra citate
4      for (int j = 0; j < A.Rows() - 1; j++)
5      {
6          for (int i = j + 1; i < A.Cols(); i++)
7          {
8              // lambda
9              scalar = -A(i, j) / A(j, j);
10
11             // Effettuiamo la stessa combinazione lineare
12             // sulle righe
13             A.LinearCombRows(i, j, scalar, i);
14             b.LinearCombRows(i, j, scalar, i);
15         }
16     }
17
18     // Effettuiamo la backward substitution
19     return BackwardSubstitution(A, b);

```

**Analisi risultati**

**File necessari** gauss\_elim.cpp

**Soluzione** Data la matrice  $A$  e il vettore  $b$  proposti dall'esercizio si ottiene il seguente risultato:

$$\begin{bmatrix} 2 & 1 & 1 \\ 1 & 1 & -2 \\ 1 & 2 & 1 \end{bmatrix} \mathbf{X} = \begin{bmatrix} 8 \\ -2 \\ -2 \end{bmatrix} \Rightarrow \mathbf{X} = \begin{bmatrix} 4 \\ -2 \\ 2 \end{bmatrix}$$

Il controllo si svolge in maniera equivalente al precedente esercizio.

### 3.2.3 Decomposizione LU

**Nozioni teoriche** Il compito della decomposizione LU di una matrice è il seguente: prendiamo una matrice  $A$  invertibile, allora essa è scomponibile in due matrici triangolari  $L$ ,  $U$  inferiori e superiori rispettivamente:

$$A = LU = \begin{bmatrix} L_{00} & 0 & 0 \\ L_{10} & L_{11} & 0 \\ L_{20} & L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} U_{00} & U_{01} & U_{02} \\ 0 & U_{11} & U_{12} \\ 0 & 0 & U_{22} \end{bmatrix}$$

La decomposizione LU non è unica quindi si assume per semplicità che la diagonale sia unitaria ( $L_{ii} = 1$ ).

Moltiplicando  $L$  ed  $U$  si ottiene una matrice che può essere ridotta tramite il metodo di Gauss: per comparazione si ottiene che la matrice ridotta è  $U$  e i termini di  $L$  sono i termini scalari moltiplicativi utilizzati per ridurla.

Ottenuta la decomposizione e provando a cercare di risolvere un sistema lineare si ottiene:

$$A\mathbf{X} = \mathbf{b} \Rightarrow LU\mathbf{X} = \mathbf{b} \Rightarrow L(U\mathbf{X}) = \mathbf{b}$$

Concludiamo che una matrice decomposta può essere risolta, risolvendo i sistemi lineari associati alle matrici triangolari utilizzando i metodi di backward e forward substitution.

**Richiesta** Partendo dalla funzione che esegue l'eliminazione gaussiana, scrivi una nuova funzione che accetti una matrice e calcoli  $L$  e  $U$ . Testare sulla matrice dell'Esercizio 2.

Qual è il determinante di una matrice triangolare (inferiore o superiore)? Dalla decomposizione LU scritta sopra, derivare la formula per calcolare il determinante e scrivere un programma che calcoli il determinante di una matrice a partire dalla decomposizione LU.

### Implementazione

**File necessari** tensor\_utils.hpp

Conviene in questo algoritmo definire la seguente *alias*.

```
1 // Coppia di tensori L e U
2 template <typename T>
3 using TensorPair = std::pair<Tensor<T>, Tensor<T>>;
```

La decomposizione LU può essere implementata come segue:

```
1 template <typename T> TensorPair<T> LUDecomposition(Tensor<T>
   const &A)
2 {
3     ... // Asserts e definizioni
4
5     // U viene costruita da una "deep copy" di A
6     auto U = Tensor<T>(A);
7
8     // Scegliamo diagonale di L unitaria
9     for (int i = 0; i < U.Rows(); i++)
10    {
11        L(i, i) = 1;
12    }
13
14    // Effettuiamo l'eliminazione di gauss
15    for (int j = 0; j < U.Rows() - 1; j++)
16    {
17        for (int i = j + 1; i < U.Cols(); i++)
18        {
19            scalar = -U(i, j) / U(j, j);
20
21            // Lo scalare e' effettivamente un elemento di L
22            L(i, j) = -scalar;
23
24            // Riduciamo U
25            U.LinearCombRows(i, j, scalar, i);
26        }
27    }
28
29    return std::make_pair(L, U);
30 }
```

Preso una matrice decomposta tramite LU allora possiamo ottenerne facilmente il determinante (considerando che il determinante di una matrice triangolare è il prodotto degli elementi sulla sua diagonale).

$$\det A = \det LU = \det L \det U = \prod_{i=0}^{N-1} U_{ii}$$

```
1 template <typename T> T DeterminantFromLU(Tensor<T> const &A)
2 {
3     // Prendiamo il secondo elemento della coppia
4     auto U = std::get<1>(LUDecomposition(A));
5     T det = 1;
6
7     // Il determinante di una matrice triangolare
8     // viene calcolato dagli elementi sulla diagonale
9     for (int i = 0; i < A.Rows(); i++)
10    {
```

```

11 |         // Assumiamo che L abbiamo 1 sulla diagonale
12 |         det *= U(i, i);
13 |     }
14 |
15 |     return det;
16 | }

```

### Analisi risultati

**File necessari** lu\_decomp.cpp

Eseguendo il codice si ottiene il seguente risultato:

$$\begin{bmatrix} 2 & 1 & 1 \\ 1 & 1 & -2 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ 0.5 & 3 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 & 1 \\ 0 & 0.5 & -2.5 \\ 0 & 0 & 8 \end{bmatrix}$$

Da cui segue che il determinante è 8 come risulta dall'output.



## 4.1 Implementazione preliminaria

Prima di procedere è necessario specificare varie funzioni/classi appositamente create per la scrittura del codice e una più facile implementazione degli algoritmi.

**Classe `FunctionData`** La classe `FunctionData` ha lo scopo di conservare i dati numerici ottenuti dalla computazione delle varie funzioni (in questo modulo principalmente polinomi) <sup>1</sup>.

Essa ha come membri due vettori di tipo `std::vector` che conservano le informazioni di  $x$  e  $f(x)$ .

**Classe `Range`** La classe `Range` ha invece lo scopo di rappresentare una successione di punti  $\{a_i\}_{i \in (a,b) \subset \mathbb{R}}$  (la rappresentazione prevede intervalli inclusivi ed esclusivi).

Scopo principale della classe sarà quello di generare nodi di distanza finita e nodi di Chebyshev.

*Ulteriori informazioni sulle due classi possono essere trovate nella documentazione dedicata negli appositi header `function_data.hpp` e `range.hpp`.*

|       |                         |    |
|-------|-------------------------|----|
| 4.1   | Implementazione         |    |
|       | preliminaria . . . . .  | 17 |
| 4.2   | Esercizi . . . . .      | 17 |
| 4.2.1 | Metodo diretto e        |    |
|       | polinomio di Newton . . | 17 |
| 4.2.2 | Funzione di Runge . . . | 19 |
| 4.2.3 | Esercizio 3? . . . . .  | 19 |

1: Nell'implementazione della classe può essere trovata anche un'implementazione apposita di un iterator per facilitare l'utilizzo dei valori quando ciclati.

## 4.2 Esercizi

### 4.2.1 Metodo diretto e polinomio di Newton

**Nozioni teoriche** Il metodo più semplice per ottenere il polinomio di interpolazione è quello di ottenere

#### Implementazione metodo diretto

**File necessari** `interpolation.hpp`

Come primo step si ottiene la matrice di Vandermonde dalla sua definizione:

```

1 |
2 | // "values" sono i valori {x_1, x_2, ...} da inserire
3 | template <typename T>
4 | tensor::Tensor<T> VandermondeMatrix(std::vector<T> const &values)
5 | {
6 |     auto mat = tensor::Tensor<T>::SMatrix(values.size());
7 |     for (int i = 0; i < values.size(); i++) {
8 |         for (int j = 0; j < values.size(); j++) {
```

```

9         mat(i, j) = pow(values[i], j);
10    }
11 }
12
13 return mat;
14 }

```

Successivamente si risolve il sistema lineare utilizzando il metodo di Gauss (invertire la matrice necessiterebbe di un ulteriore eliminazione di Gauss ed ulteriore allocamento di memoria).

```

1 template <typename T>
2 std::vector<T> DirectCoefficients(func::FunctionData<T> const &f)
3 {
4     auto f_tensor = tensor::Tensor<T>::FromData(f.F());
5     tensor::Tensor<T> vande_matrix = VandermondeMatrix(f.X());
6     tensor::Tensor<T> values =
7         tensor::GaussianElimination(vande_matrix, f_tensor);
8
9     // Trasforma l'oggetto tensore in std::vector
10    return values.RawData();
11 }

```

**Implementazione metodo di Newton** Analogamente al metodo diretto calcoliamo i coefficienti per il polinomio di newton seguendo la formula citata precedentemente otteniamo.

```

1 template <typename T>
2 std::vector<T> NewtonCoefficients(func::FunctionData<T> const &f)
3 {
4     int N = f.Size();
5
6     std::vector<T> a(N);
7
8     auto A = tensor::Tensor<double>::SMatrix(N);
9
10    // Rempiamo la prima colonna con i valori della funzione
11    for (int i = 0; i < N; i++) {
12        A(i, 0) = f.F(i);
13    }
14
15    // Calcoliamo le differenze divise
16    for (int j = 1; j < N; j++) {
17        for (int i = 0; i < N - j; i++) {
18            A(i, j) = (A(i + 1, j - 1) - A(i, j - 1)) / (f.X(i + j) - f.X(i));
19        }
20    }
21
22    // Estrapoliamo i coefficienti
23    for (int i = 0; i < N; i++) {
24        a[i] = A(0, i);
25    }
26
27    return a;
28 }

```



**Considerazioni** L'algoritmo è direttamente implementato rispetto alla logica che abbiamo considerato per calcolare i coefficienti di Newton. Uno svantaggio di questo algoritmo sta nell'utilizzo dell'oggetto Tensor, infatti il funzionamento interno della classe traduce una matrice  $2 \times 2$  in un vettore contiguo: questo permette un'ottimizzazione della cache della cpu rispetto ad un vettore di vettori, ma nel caso utilizzato la matrice è solo occupata a metà, lasciando inizializzati a 0 molti valori non utilizzati nella computazione dei coefficienti. Un modo per evitare ciò sarebbe per esempio quello di implementare un oggetto ad hoc per il problema, ai fini concettuali, però, l'implementazione sarebbe la medesima.

**Analisi dei risultati** Dalla implementazione dei metodi sopra citati e dai dati forniti dal problema otteniamo le seguenti interpolazioni:

### 4.2.2 Funzione di Runge

#### 4.2.3 Esercizio 3?



## 5.1 Esercizi

### 5.1.1 Oscillatore approssimato

|  |    |
|--|----|
| 5.1 Esercizi . . . . .                   | 21 |
| 5.1.1 Oscillatore approssimato . . . . . | 21 |
| 5.1.2 Oscillatore reale . . . . .        | 21 |
| 5.1.3 Attrattore di Lorenz . . . . .     | 22 |
| 5.1.4 Sistema a tre corpi . . . . .      | 22 |

**Richiesta** Risolvere l'equazione differenziale dell'oscillatore armonico con i seguenti valori iniziali:

$$\ddot{\theta}(t) = -\theta(t) \quad (5.1)$$

con  $\theta(0) = 0$  e  $\dot{\theta}(0) = 1$ .

1. Utilizzare il metodo di Eulero, il metodo di Runge-Kutta del secondo ordine (RK2) e il metodo di Runge-Kutta del quarto ordine (RK4).
2. Ottenere la soluzione analitica  $\theta(t)$  e confrontarla con la soluzione numerica  $\eta(t; h)$  ottenuta con i tre metodi. Studiare l'errore:

$$e(t, h) = \eta(t, h) - \theta(t)$$

in funzione del passo  $h$  e verificare che i metodi abbiano l'ordine atteso.

### 5.1.2 Oscillatore reale

**Richiesta** Considerare l'equazione differenziale del pendolo senza l'approssimazione delle piccole oscillazioni:

$$\ddot{\theta}(t) = -\sin \theta(t) \quad (5.2)$$

con  $\theta(0) = 0$  e  $\dot{\theta}(0) = 1$ .

1. Risolvere numericamente l'equazione differenziale e tracciare il grafico di  $\theta(t)$  e  $\dot{\theta}(t)$ .
2. Ripetere l'esercizio includendo un termine di attrito:

$$\ddot{\theta}(t) = -\sin \theta(t) - \gamma \dot{\theta}(t)$$

3. e un termine forzante:

$$\ddot{\theta}(t) = -\sin \theta(t) - \gamma \dot{\theta}(t) + A \sin \omega t$$

con  $\gamma \in (0, 2)$  e  $\omega \in (0, 2)$ .

### 5.1.3 Attrattore di Lorenz

**Richiesta** Studiare il sistema di ODE:

$$\begin{cases} \dot{x}(t) = 10(y(t) - x(t)) \\ \dot{y}(t) = 28x(t) - y(t) - xz(t) \\ \dot{z}(t) = -8/3z(t) + xy(t) \end{cases} \quad (5.3)$$

1. Utilizzare il metodo di Eulero, il metodo di Runge-Kutta del secondo ordine (RK2) e il metodo di Runge-Kutta del quarto ordine (RK4).
2. Tracciare il grafico di  $x(t)$ ,  $y(t)$  e  $z(t)$ .

### 5.1.4 Sistema a tre corpi

**Richiesta** Studiare il sistema gravitazionale che obbedisce alle equazioni del moto:

$$\ddot{\vec{x}}_i = \sum_{j \neq i} m_j \frac{\vec{x}_j - \vec{x}_i}{|\vec{x}_j - \vec{x}_i|^3} \quad (5.4)$$

Nel caso di tre masse puntiformi,  $i = 1, 2, 3$ , con i seguenti parametri e condizioni iniziali:

$$\begin{aligned} m_1 = m_2 = m_3 = 1 \\ \vec{x}_1 = (1, 0, 0), \quad \dot{\vec{x}}_1 = (0, 0.15, -0.15) \\ \blacktriangleright \vec{x}_1 = (-1, 0, 0), \quad \dot{\vec{x}}_1 = (0, -0.15, 0.15) \\ \vec{x}_1 = (0, 0, 0), \quad \dot{\vec{x}}_1 = (0, 0, 0) \\ \\ m_1 = 1.6, m_2 = m_3 = 0.4 \\ \vec{x}_1 = (1, 0, 0), \quad \dot{\vec{x}}_1 = (0, 0.4, 0) \\ \blacktriangleright \vec{x}_1 = (-1, 0, 0), \quad \dot{\vec{x}}_1 = (0, -0.8, 0.7) \\ \vec{x}_1 = (0, 0, 0), \quad \dot{\vec{x}}_1 = (0, -0.8, -0.7) \end{aligned}$$

1. Risolvere numericamente il sistema di equazioni.
2. Tracciare il grafico dell'energia totale del sistema in funzione del tempo.