Elaborato di Fisica Computazionale

A.A 2024/2025

Andrea Rossi N. 897139

18 dicembre 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, sopratutto 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.

Indice

Numeri e approssimazioni 1

1.1 Introduzione: Numeri

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 ϵ *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 (IEE 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: $3.40 \cdot 10^{38}$
- ϵ: $\sim 10^{-7}$

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

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

Richiesta Scriverete un programma C che esegua le seguenti operazioni:

```
define f in single precision = 1.2e34
for loop with 24 cycles:
    f *= 2
    print f in scientific notation
repeat for d in double precision, starting from 1.2e304
define d in double precision = 1e-13
for loop with 24 cycles:
    d /= 2
    print d and 1+d in scientific notation
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`

Il codice sorgente scritto utilizza funzionalità base del linguaggio C. [1]

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

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 variaili 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 σ_x, σ_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:

```
calculate (0.7 + 0.1) + 0.3 and print 16 digits
calculate 0.7 + (0.1 + 0.3) and print 16 digits
define xt = 1.e20; yt = -1.e20; zt = 1
calculate (xt + yt) + zt
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. $(0.7 + 0.1) + 0.3 =_? 0.7 + (0.1 + 0.3)$:

   ```
   Output: 1.1000000238418579, 1.1000000238418579
   ```

   La somma risulta associativa.
2. $[10^{20} + (-10^{20})] + 1 =_? 10^{20} + [(-10^{20}) + 1]$:

   ```
   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 possegono 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 describile 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 ~ 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.

1.3 Introduzione: Approssimazioni

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.

1.4 Esercizi

1.4.1 Funzione esponenziale

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

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

dove ϵ 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 Figura 6.3.

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 Figura 1.2.
2. Sempre dalla Figura 1.2 si può notare come l'errore aumenti all'allontanarsi da x = 0 e cominci a differire dall'errore teorico. Ciò si intuscisce 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 Figura 1.2 per $n = 4$ e l'errore tende a 0 molto più velocemente.

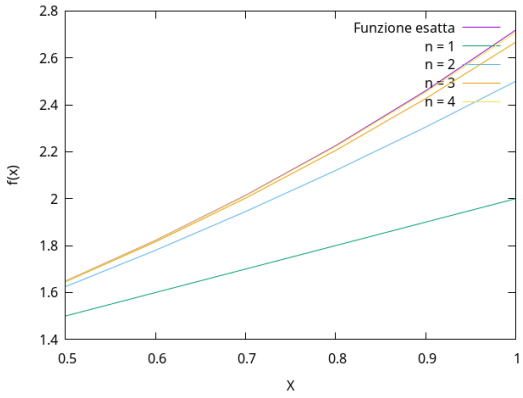


Figura 1.1: Confronto tra funzione esponenziale e la n-esima approssimazione

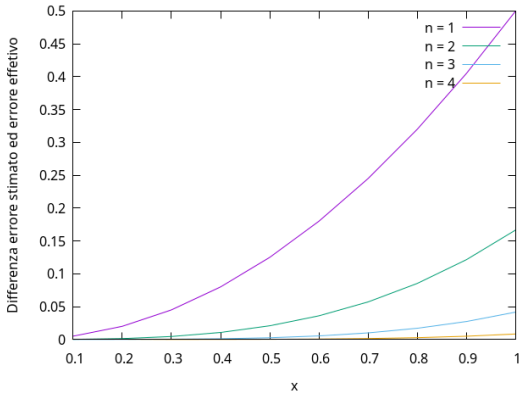


Figura 1.2: Differenza tra errore teorico ed errore ottenuto

1.4.2 Problema di Basilea

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

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

Richiesta Calcolare la seguente somma

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

con

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

1. Calcolare la somma in precisione singola usando l'ordine normale $n = 1, 2, \ldots, N$
2. Calcolare la somma in precisione singola usando l'ordine inverso $n = N, \ldots, 2, 1$
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:

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

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

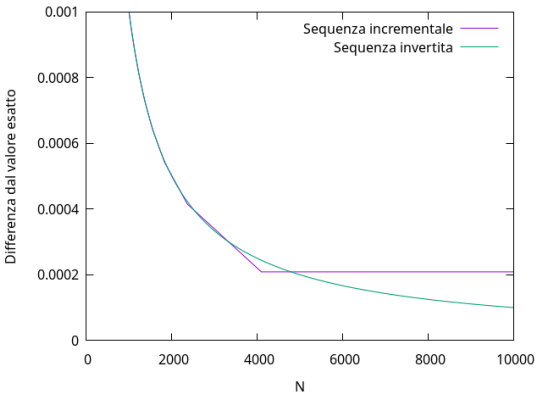


Figura 1.3: $|S(N) - \pi^6/2|$ per valori grandi di N, si nota un singolare andamento della somma a partire da valori $x \approx 4000$, **la stessa cosa non succede invece per numeri a doppia precisione**

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

Matrici 2

2.1 Introduzione

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

- `tensor_obj.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` contentente varie funzionalità utili e gli algoritmi creati per ogni esercizio.

Le cartelle corrispettive dei vari esercisi 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.

2.2 Esercizi

2.2.1 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*:

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

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     LOG_ASSERT(A.Cols() == A.Rows(), "A must be a square matrix",
       utils::ERROR);
19     LOG_ASSERT(Ama otteniamo le seguenti interpolazioni:.Cols() ==
        b.Rows(), "A and b are dimensionally incompatible",
20             utils::ERROR);
21     LOG_ASSERT(IsUpperTriangular(A), "A is not upper triangular",
       utils::ERROR);
22
23
24     ... // Definizione delle variabili
25
26     /*
27     * Si itera rispetto alle colonne di b
28     * si risolve il sistema per ogni colonna
29     * utile per il calcolo della matrice inversa
30     * nei prossimi esercizi
31     */
32     for (int k = 0; k < b.Cols(); k++)
33     {
34         // Formula di backward substitution citata precedentemente
35         for (int i = N - 1; i >= 0; i--)
36         {
37             sum = 0;
38             for (int j = i + 1; j <= N - 1; j++)
39             {
```

```
40                sum += A(i, j) * solution(j, k);
41            }
42
43            solution(i, k) = (b(i, k) - sum) / A(i, i);
44        }
45    }
46
47    return solution;
48 }
```

Analisi risultati

File necessari `backward_subst.cpp`

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

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

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

2.2.2 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 3×3 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:

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

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. ($\vec{b} \longrightarrow \mathbb{I}$)

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:

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

e controllare la soluzione.

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

```
    ... // Asserts e definizioni

    // Utilizziamo le formule sopra citate
    for (int j = 0; j < A.Rows() - 1; j++)
    {
        for (int i = j + 1; i < A.Cols(); i++)
        {
            // lambda
            scalar = -A(i, j) / A(j, j);

            // Effettuiamo la stessa combinazione lineare
            // sulle righe
            A.LinearCombRows(i, j, scalar, i);
            b.LinearCombRows(i, j, scalar, i);
        }
    }

    // Effettuiamo la backward substitution
    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:

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

Il controllo si svolge in maniera equivalente al precedente esercizio.

2.2.3 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 otteniene:

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

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*.

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

La decomposizione LU può essere implementata come segue:

```cpp
template <typename T> TensorPair<T> LUDecomposition(Tensor<T>
    const &A)
{
    ... // Asserts e definizioni

    // U viene costruita da una "deep copy" di A
    auto U = Tensor<T>(A);

    // Scegliamo diagonale di L unitaria
    for (int i = 0; i < U.Rows(); i++)
    {
        L(i, i) = 1;
    }

    // Effettuiamo l'eliminazione di gauss
    for (int j = 0; j < U.Rows() - 1; j++)
    {
        for (int i = j + 1; i < U.Cols(); i++)
        {
            scalar = -U(i, j) / U(j, j);

            // Lo scalare e' effettivamente un elemento di L
            L(i, j) = -scalar;

            // Riduciamo U
            U.LinearCombRows(i, j, scalar, i);
        }
    }

    return std::make_pair(L, U);
}
```

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

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

```cpp
template <typename T> T DeterminantFromLU(Tensor<T> const &A)
{
    // Prendiamo il secondo elemento della coppia
    auto U = std::get<1>(LUDecomposition(A));
    T det = 1;

    // Il determinante di una matrice triangolare
    // viene calcolato dagli elementi sulla diagonale
    for (int i = 0; i < A.Rows(); i++)
    {
```

```
        // Assumiamo che L abbiamo 1 sulla diagonale
        det *= U(i, i);
    }

    return det;
}
```

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.

Intepolazione 3

3.1 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) [1].

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

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

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`.

3.2 Esercizi

3.2.1 Esercizio 6

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   auto mat = tensor::Tensor<T>::SMatrix(values.size());
6
7   for (int i = 0; i < values.size(); i++) {
8     for (int j = 0; j < values.size(); j++) {
```

```
      mat(i, j) = pow(values[i], j);
    }
  }

  return mat;
}
```

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).

```
template <typename T>
std::vector<T> DirectCoefficients(func::FunctionData<T> const &f)
    {
  auto f_tensor = tensor::Tensor<T>::FromData(f.F());

  tensor::Tensor<T> vande_matrix = VandermondeMatrix(f.X());
  tensor::Tensor<T> values =
      tensor::GaussianElimination(vande_matrix, f_tensor);

  // Trasforma l'oggetto tensore in std::vector
  return values.RawData();
}
```

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

```
template <typename T>
std::vector<T> NewtonCoefficients(func::FunctionData<T> const &f)
    {
  int N = f.Size();

  std::vector<T> a(N);

  auto A = tensor::Tensor<double>::SMatrix(N);

  // Rempiamo la prima colonna con i valori della funzione
  for (int i = 0; i < N; i++) {
    A(i, 0) = f.F(i);
  }

  // Calcoliamo le differenze divise
  for (int j = 1; j < N; j++) {
    for (int i = 0; i < N - j; i++) {
      A(i, j) = (A(i + 1, j - 1) - A(i, j - 1)) / (f.X(i + j) - f.
    X(i));
    }
  }

  // Estrapoliamo i coefficienti
  for (int i = 0; i < N; i++) {
    a[i] = A(0, i);
  }

  return a;
}
```

Considerazioni L'algoritmo è direttamente implementato rispetto a lla 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 2x2 in una vettore contiguo: questo permette un ottimizzazzione 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 coefficient. 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.

Punto 1

Analisi e conclusioni Dalla implementazione dei metodi sopra citati e dai dati forniti dal problema otteniamo la figura Figura 3.1.

Come si puo' notare il polinomio di Newton nel caso (1) e (2) non attraversa il punto x_5, poiche' esso garantisce solamente di passare esattamente nei punti dati inizialmente. Il caso(3) e' invece un caso particolare, infatti, sembra che il polinomio di Newton passi esattamette per x_5 (almeno entro l'errore in doppia precisione), questo puo' essere spiegato se il polinomio coincide esattamente con la funzione campionata oppure in generale per altri casi specifici altamente non banali. (per esempio, la funzione e' tale che la sua espansione in serie coincida in quel punto con il polinomio di newton).

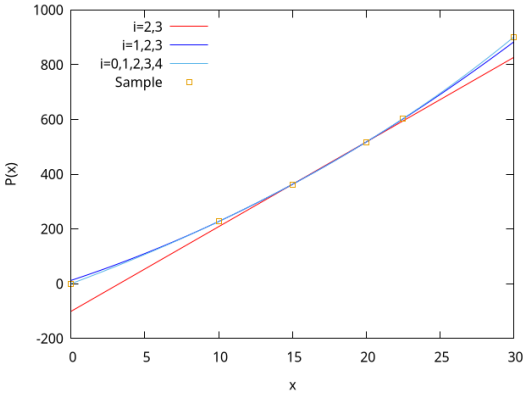


Figura 3.1: Confronto tra i vari sample

Punto 2: Funzione di Runge

Analisi dei risultati Dalla Figura 3.2 si puo' osservare il fenomeno di Runge, il carattere oscillatorio si presenta in entrambi i metodi: nel metodo di Newton si osserva un carattere oscillatorio piu' accentuato rispetto al metodo diretto, questo e' dovuto al fatto che il metodo di Newton e' un metodo iterativo che tende a convergere al polinomio di interpolazione, mentre nel metodo diretto il problema e' meno notevole.

Attraverso l'utilizzo dei nodi di Chebyshev si evince (Figura 3.3) che il problema presentatosi precedentemente e' risolto, si nota ancora piu' chiaramente anche che il metodo diretto dopo circa $x = 0$ diventa *ill-conditioned*: il vantaggio ottenuto nel caso dei nodi fissi (per valori inferiori a 0) dal metodo diretto e' completamente superato dal metodo di Newton con il nuovo campionamento.

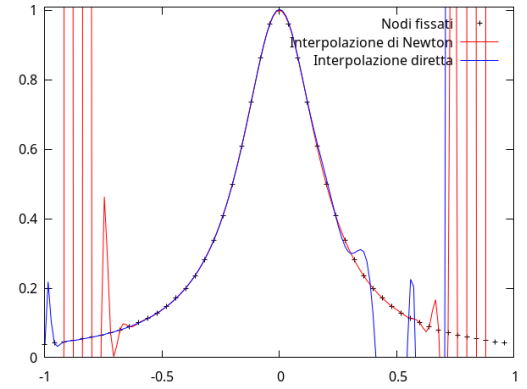


Figura 3.2: Nodi equidistanti

Conclusioni Durante il campionamento di una funzione e' di fondamentale importanza la scelta dei nodi ed il metodo utilizzato: i risultati ottenuti, infatti, possono variare drasticamente.

3.2.2 Esercizio 7

Nozioni teoriche Un'ulteriore via per interpolare una funzione e' di non considerare un interpolazione globale, ma una locale tra i punti di campionamento vicini. Si puo' quindi decidere di utilizzare una funzione

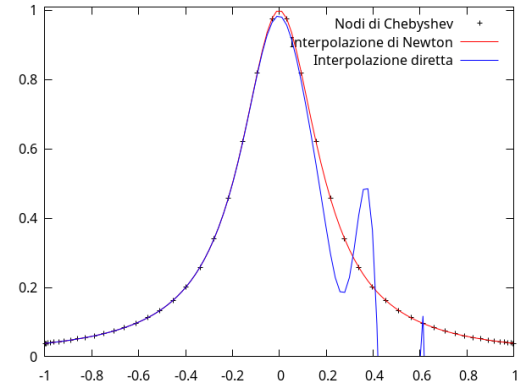


Figura 3.3: Nodi di Chebyshev

linare, quadratica, ... per unire i vari punti, il metodo viene chiamato Spline.

Implementazione L'implementazione di un algoritmo di grado n segue in modo induttivo dai casi di $n = 1, 2, 3, \ldots$: si impongono continuita' delle derivate e valori ai parametri liberi (per esempio si impone che la derivata sia nulla nel caso quadratico).

Nel caso $n = 1, 2$ svolto l'algoritmo generale non e' strettamente necessario e si possono riempire direttamente le matrici ottenute nelle note, in caso di n piu' grandi una ristrutturazione del codice sarebbe necessaria.

Per completezza si include di seguito l'algoritmo generale:

```
1  for i = 0; i < n do
2      for a = 0; a < q do
3          M_{iq,iq+a} ← (x_i)^a
4          M_{iq+1,iq+a} ← a(x_i)^{a-1}   Imponiamo la continuita' della derivata
             prima
5      b_i ← f_i
6      b_{i+1} ← 0   Prendiamo come convenzione 0
7  if q > 2 then
8      for i = 0; i < n − 1 do
9          for a = 1; a < q do
10             Condizione di continuita' tra le derivate di ordine superiore
11             M_{iq+2,iq+a} ← a(x_{i+1})^{a-1}
12             M_{iq+2,(i+1)q+a} ← −a(x_{i+1})^{a-1}
```

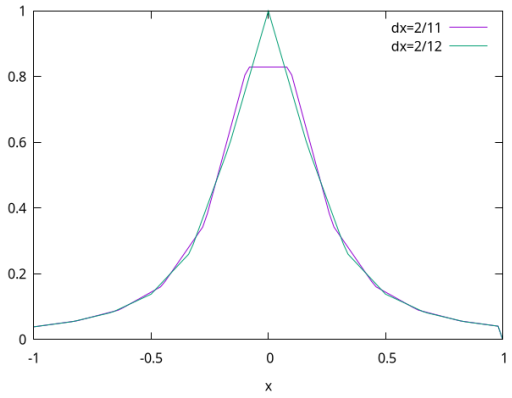


Figura 3.4: Spline lineare

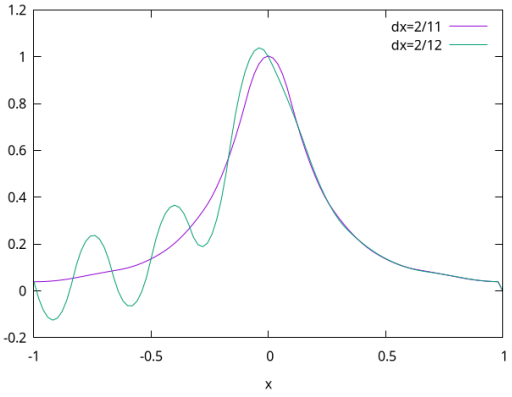


Figura 3.5: Spline quadratica

Analisi risultati

Spline lineare L'analisi di Figura 3.4 e' triviale: la spline lineare unisce semplicemente i punti campionati, la differenza sta quindi nel fatto che nei differenti step di campionamento; in un caso si campiona 0 ma non nell'altro caso.

Spline quadratica Il caso in Figura 3.5 e' molto piu' particolare, la differenza di campionamento, infatti, causa l'emergere di un carattere oscillatorio nella interpolazione. La spiegazione e' facilmente intuibile se si guarda l'interpolazione da destra verso sinistra: dovendo passare per $(0, 1)$ e dovendo valere $S'_{x_i} = S'_{x_{i+1}} = 0$ allora la spline sovrastima il valore di $f(0)$. Questo fenomeno e' noto come *overshooting*. La *sovrastima* causa nei punti successivi (che devono soddisfare le regole precedenti e quindi devono passare per i punti di campionamento) all'andamento osservato. Questo non si verifica nel primo campionamento poiche' viene mantenuta la simmetria della funzione.

Autovalori e autovettori 4

4.1 Introduzione ed esercizio 8

File necessari `eigenvalues.hpp`

Esercizio Per illustrare esaustivamente le tecniche numeriche utilizzate si considerino i vari punti dell'esercizio 8 partendo dalla matrice seguente:

$$A = \begin{pmatrix} 4 & -i & 2 \\ i & 2 & 2+7i \\ 2 & 2-7i & -2 \end{pmatrix}$$

4.2 Punto 1: Power method

Nozioni teoriche Il primo metodo e' definito *Power method* e consiste nel calcolare l'autovalore piu' grande di una matrice A e l'autovettore associato.

Dato un qualsiasi vettore per il teorema spettrale se A e' diagonalizzabile allora ogni vettore $\mathbf{x}_0$ puo' essere scritto sulla base dagli autovettori $\mathbf{v}_n$ di A.

Prendiamo dunque il vettore:

$$\mathbf{y}_n = \mathbf{A}^n \mathbf{x}_0 = \sum_m^N c_m \lambda_m^n \mathbf{v}_m = \lambda_{max} \sum_m^N c_m \left(\frac{\lambda_m}{\lambda_{max}} \right)^n \mathbf{v}_m$$

Per numeri sufficientemente grandi di N e se il rapporto $\frac{\lambda_m}{\lambda_{max}}$ e' minore di 1 allora il termine $\left(\frac{\lambda_m}{\lambda_{max}} \right)^n$ tende a 0 e quindi il termine dominante e' λ_{max}.

Quindi

$$\lim_{n \to \infty} \mathbf{y}_n = c_0 \lambda_{max} \mathbf{v}_0$$

Visto che $\mathbf{y}_0$ e' multiplo di $\mathbf{v}_0$ esso stesso e' autovettore .

Per ottenere l'autovalore calcoliamo il cosiddetto *Coefficiente di Rayleigh*

$$\mathsf{C}_R = \frac{\mathbf{y}_n^T \mathbf{A} \mathbf{y}_n}{\mathbf{y}_n^T \mathbf{y}_n} \xrightarrow{n=\infty} \frac{\mathbf{v}_0^T \lambda_{max} \mathbf{v}_0}{\mathbf{v}_0^T \mathbf{v}_0} = \lambda_{max}$$

Shifted Power Method Per casi particolari (principalmente autovalori simmetrici e opposti, come si vedra' nella ricerca degli zeri di polinomi ortonormali) si utilizza il *Shifted Power Method*: si introduce uno shift $A + \alpha I$ per rendere lo spettro positivo e si reshiftano gli autovalori di $\lambda - \alpha$ ad algoritmo finito. Questo evita il carattere oscillatorio che si puo' estaurare quando $\lambda_i = -\lambda_j$. In questo capitolo assumeremo $\alpha = 0$.

Implementazione La documentazione dell'algoritmo e' esaustiva e non necessita di ulteriori spiegazioni.

Risultati si ottiene il seguente output:

```
Power Eigenvalue: (8.45188,-2.22045e-16)
Power Eigenvector:
 (0.387073,-0.143579)
 (0.350492,0.615922)
 (0.55364,-0.144353)

Ax=
 (0.387073,-0.143579)
 (0.350492,0.615922)
 (0.55364,-0.144353)
```

Dove tra parentesi si indica la parte reale e immaginaria rispettivamente, inoltre il metodo funziona correttamente visto che $Ax = \lambda x$ (λ e' molplicato nell'autovettore ma essendo una costante cio' e' ininfluente).

4.3 Punto 2: Inverse Power method

Nozioni teoriche Analogamente per il metodo precedente si puo' calcolare l'autovalore piu' piccolo di una matrice A e l'autovettore associato, in questo caso si invertira' Ae si raccogliera' il termine piu' piccolo λ_{min} mentre $\left(\frac{\lambda_{min}}{\lambda_n}\right)^n \to 0$, lo stesso ragionamento si puo' anche fare per lo shift in questo caso esso verra' utilizzato per raggiungere piu' velocemente il risultato.

Risultati Si ottiene:

```
 Inverse Power Eigenvalue: (3.1896,-6.93889e-18)
Inverse Power Eigenvector:
 (0.00850804,0.906366)
 (0.370705,-0.0809101)
 (0.0370076,-0.181906)

Ax=
 (0.00850804,0.906366)
 (0.370705,-0.0809101)
 (0.0370076,-0.181906)
```

Il risultato e' inoltre verificato dal controllo di **Ax**.

4.4 Punto 3: studio della convergenza

Risultati Il risultato e' visibile nella figura 4.1.

Analisi e conclusioni Come si puo' osservare il metodo inverso converge molto piu' velocemente del metodo diretto. Questo pero' dipende dagli autovalori della matrice e dal rapporto λ_n/λ_{max} e λ_{min}/λ_n. Per esempio, presa la matrice A con $a_{00} = 8$ gli autovalori sono molto piu' vicini e si ottiene la figura 4.2.

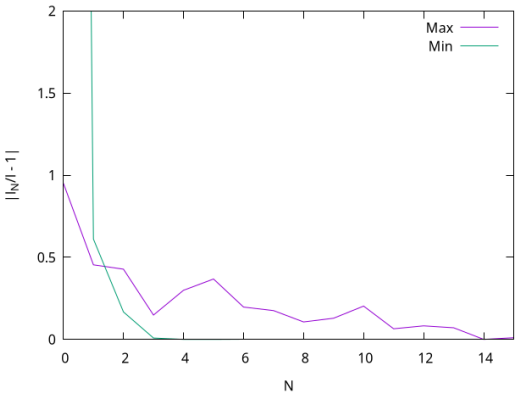


Figura 4.1: Confronto tra la velocita' di convergenza dei due metodi

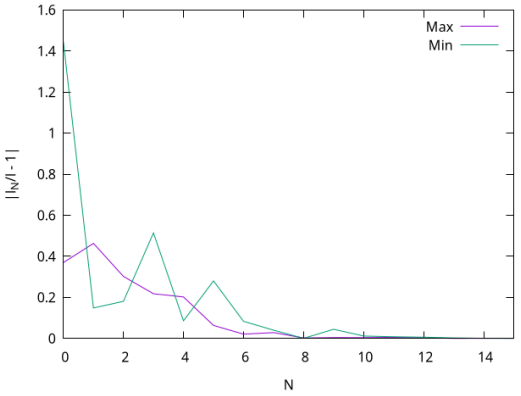


Figura 4.2: Confronto tra la velocita' di convergenza dei due metodi, con $a_{00} = 8$

4.5 Punto 4: Deflation method

Nozioni teoriche Attraverso l'utilizzo del *Power method* possiamo inoltre ottenere tutti gli autovettori e autovalori di una matrice A tramite il *Deflation method*: trovato l'autovalore massimo possiamo rimuoverlo dallo spettro della matrice A in questo modo otterremo il secondo valore massimo e cosi' via per tutti gli autovalori, ovviamente cio' non potrebbe funzionare nel caso del metodo inverso poiche', sottraendo l'autovalore alla matrice, otterremmo una matrice singolare e dunque non invertibile.

Per far cio' si puo' rimuovere iterativamente la proiezione di $\mathbf{v}_0$ corrente da A:

$$A' = A - \lambda_{max}\mathbf{v}_0\mathbf{v}_0^T$$

Implementazione In maniera simile alla precedente la documentazione e le funzione della classe `Tensor<T>` rendono l'implementazione triviale.

Risultati Si ottengono i seguenti autovalori:

```
1 (8.45188,0)
2 (-7.64148,0)
3 (3.1896,6.93889e-18)
```

Si nota banalmente che l'autovalore λ_{min} corrisponde con quello trovato nel punto 2. Unica considerazione e' la parte immaginaria che in questo caso risulta 0 per l'autovalore massimo ma, essendo il valore precedente $< \epsilon_{double}$, si puo' considerare 0.

Ricerca degli zeri 5

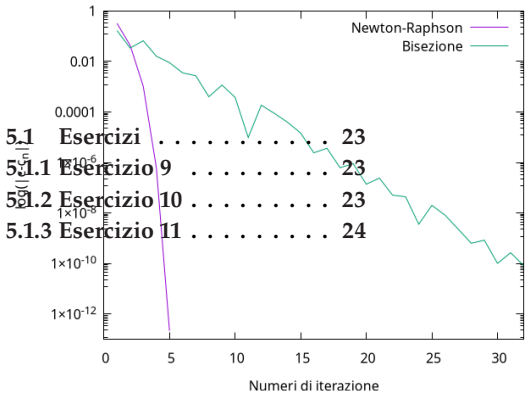


Figura 5.1: Convergenza dei due meetodi

5.1 Esercizi

5.1.1 Esercizio 9

1. Dato $-x^2+x+\frac{1}{2}=0$ le radici sono $x_\pm = \frac{-1\pm\sqrt{1-4(-1)(1/2)}}{-2} = \frac{1\pm\sqrt{3}}{2}$. Dal metodo di bisezione otteniamo `1.36603` nell'intervallo $[0.8, 1.6]$ come verificato dalla teoria, il medesimo risultato si ottiene per il metodo di Newton-Raphson.
2. TODO:

3, 4 Dalla Figura 5.1 si osserva che il metodo di bisezione converge piu' lentamente rispetto a quello di Newton-Raphson. Inoltre si osserva che il metodo di Bisezione converge linearmente ($\sim 2^{-n}$).

5.1.2 Esercizio 10

Convergenza lenta Lo zero della funzione x^2 e' banalmente 0 se consideriamo l'espressione analitica dell'errore del metodo di Newton-Raphson otteniamo:

$$|0 - x_{n+1}| = x_n - \frac{x_n^2}{2x_n} = x_n - \frac{1}{2}x_n = \frac{1}{2}x_n$$

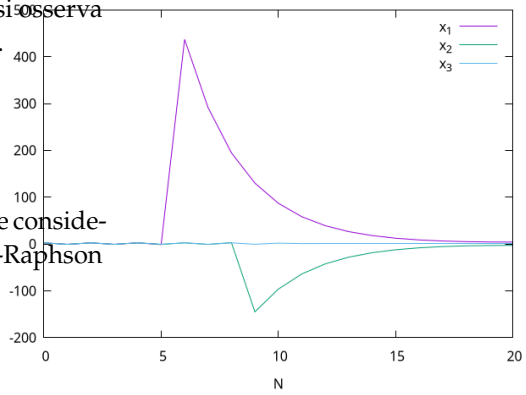


Figura 5.2: Carattere oscillatorio del metodo di Newton per le condizioni iniziali del terzo punto, x_1, x_2, x_3 sono i valori in ordine proposti nelle note

Quindi l'errore del metodo di Newton-Raphson e' lineare in questo caso specifico. In effetti come si puo' osservare dal file `ex_10_1.dat` si ottiene un valore costante di `0.5` come da teoria (a meno di errori di precisione successivi).

In conclusione la convergenza del metodo di Newton-Raphson, nonostante in generale sia quadratica, puo' essere diversa da quella prevista, a causa delle proprieta' analitiche della funzione in esame.

Cicli Nei casi in cui si vuole studiare la convergenza del metodo di Newton-Raphson si puo' studiare il frattale associato. Con un semplice programma python (`newton_fractal.py`) si possono studiare le aree di convergenza del metodo data la funzione.

TODO

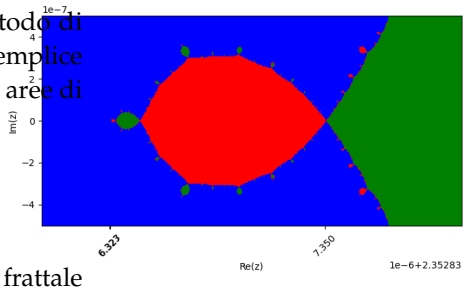


Figura 5.3: Frattale di Newton associato al terzo punto, colori diversi corrispondono a zeri diversi, si noti che due dei valori sono quasi sovrapposte in questa scala

Condizioni iniziali Anche in questo caso studiamo i risultati e il frattale associato alla funzione ottenendo Figura 5.2 e Figura 5.3.

Dei valori iniziali relativamente simili cadono in delle regioni di convergenza diverse, basterebbe un leggero spostamento della condizione iniziale per ottenere uno zero differente.

Il carattere oscillatorio e' spiegabile analogamente al punto precedente.

5.1.3 Esercizio 11

Implementazione L'implmentazione e' diretta:

```
// ... Asserts e definizione della funzione

int n_coeffs = coefficients.Rows() - 1;

auto coeffs_matrix = tensor::Tensor<T>::SMatrix(n_coeffs);

// Si riempie la matrice
for (int i = 0; i < n_coeffs; i++) {
  if (i < n_coeffs - 1) {
    coeffs_matrix(i, i + 1) = 1.0;
  }

  coeffs_matrix(n_coeffs - 1, i) = -coefficients(i) /
  coefficients(n_coeffs);
}

// Si utilizza il Deflation Method con un termine correttivo
  alpha
auto solution = eigen::PowerMethodDeflation(coeffs_matrix, n,
  alpha);

// Sorting e return...
```

Analisi risultati Dopo aver aggiunto uno shift del *Power method* rispettivamente di $\alpha_1 = 0.01, \alpha_2 = 0.1$. Si ottengono i seguenti valori

```
Zeri di Legendre:
-0.973907
-0.865063
-0.67941
-0.433395
-0.148874
0.148874
0.433395
0.67941
0.865063
0.973907
Check:
2.59017e-07
4.42142e-08
-1.97815e-11
-5.12745e-11
7.348e-09
1.31299e-09
8.9706e-12
2.01226e-11
-4.42287e-08
-2.59039e-07
```

```
Zeri di Hermite:
-2.3506
-1.33585
0.00120427
0.436077
1.33585
2.3506
0.433395
-0.433395
0
7.16395e-322
Check:
1.81899e-12
-6.25278e-13
-119.999
-3.86358e-13
1.02318e-12
1.81899e-12
```

L'algoritmo funziona dunque correttamente.

Equazioni differenziali ordinarie | 6

6.1 Introduzione

Un'equazione differenziale ordinaria (ODE) è un'equazione differenziale che dipende da una sola variabile indipendente, x. Le sue incognite consistono in una o più funzioni $y(x)$ e coinvolgono le derivate di queste funzioni. Consideriamo l'equazione esplicita del primo ordine:

$$y' = f(x, y),$$

La soluzione per ODE di ordine superiore può essere ottenuta trasformandole in un sistema di ODE di primo ordine.

In generale, possiamo quindi risolvere un sistema di n equazioni differenziali al primo ordine:

$$\begin{cases} y_1'(x) = f_1(x, y_1, \dots, y_n), & y_1(0) = a_1 \\ \vdots & \\ y_n'(x) = f_n(x, y_1, \dots, y_n), & y_n(0) = a_n \end{cases} \rightarrow \mathbf{y}' = \mathbf{f}(x, \mathbf{y})$$

6.1.1 Metodi numerici

I metodi studiati durante il corso vengono denominati *a singolo passo*: la soluzione numerica viene calcolata con il seguente approccio:

$$y_{n,i+1} = y_{n,i} + h\Phi(\mathbf{y}, x; h; \mathbf{f})$$

Dove n indica la coordinata di $\mathbf{y}$ e i il numero del passo, e Φ è un funzionale.

Si può tradurre in C++ il funzionale Φ in un oggetto di tipo `std::function` e successivamente si può implementare il metodo di risoluzione generale a singolo passo nel seguente modo:

```
... // Dichiarazione funzione

// Vettore delle conditioni iniziali
Tensor<T> y = initial\_conds\_;

// Allocazione del tensore risultante
Tensor<T> result =
    Tensor<T>::Matrix(
                coords_range_.Nodes().size(),
                initial_conds_.Rows()
    );

// Si sceglie il funzionale in base al metodo scelto
auto method_func = GetMethodFunction(method_);
```

```
15
16  int step_index = 0;
17
18  for (const auto &t : coords_range_) {
19    // Si aggiunge alla riga specificata lo stato
20    // precedente/iniziale del sistema
21    for (int i = 0; i < y.Rows(); ++i) {
22      result(step_index, i) = y(i);
23    }
24
25    // Si calcola lo stato successivo
26    y = y + method_func(t, y) * coords_range_.Step();
27
28    step_index++;
29  }
30
31  return result;
```

`result` è una matrice con entrate

$$\begin{aligned} R_{ij} &= y_j(t_i) \\ &i \in \{n \in \mathbb{N} : t_0 \leq t_n \leq t_{max}\} \\ &j \in [0, \dim \mathcal{Y}] \quad \mathbf{y} \in \mathcal{Y} \end{aligned}$$

Metodo di Eulero La procedura consiste nell'osservare che $f(x, y(x))$ è uguale alla pendenza $y'(x)$ della soluzione. L'idea di base è di discretizzare la derivata con una differenza finita, scegliendo un $h \neq 0$ piccolo in modo tale che:

$$y(x + h) = y(x) + hf(x, y(x)) + O(h^2).$$

Allora, troncando ordini quadratici, otteniamo che

$$\Phi = f(x, y(x))$$

Metodo RK2 ed RK4 Per ottenere metodi di ordine superiore, si deve costruire una funzione $\Phi(x, y; h; f)$ che approssimi meglio la serie di Taylor di $\Delta(x, y; h; f)$. Per esempio, possiamo partire con:

$$\Phi(x, y; h; f) = a_1 f(x, y) + a_2 f(x + p_1 h, y + p_2 h f(x, y)), \tag{6.1}$$

Espandendo n volte Φ dove Φ è stato generalizzato espandendo ricorsivamente il ragionamento in 6.1 si confrontano poi i coefficienti con l'espansione di taylor di y e si ottengono dei vincoli su di essi, non unici.

Utilizzando questo ragionamento si possono ottenere, scegliendo arbitrariamente i coefficienti liberi:

RK2 dove il procedimento per il calcolo è:

$$\begin{cases} k_1 = f(x, y), \\ \Phi = f(x + \frac{1}{2}h, y + \frac{1}{2}hk_1), \end{cases}$$

RK4 dove il procedimento per il calcolo è:

$$\begin{cases} k_1 = f(x, y), \\ k_2 = f\left(x + \frac{1}{2}h, y + \frac{1}{2}hk_1\right), \\ k_3 = f\left(x + \frac{1}{2}h, y + \frac{1}{2}hk_2\right), \\ \Phi = f(x + h, y + hk_3), \end{cases}$$

6.2 Implementazione preliminaria

Classe `ODESolver` La classe `ODESolver` ha lo scopo di risolvere un qualsiasi sistema di equazioni differenziali al primo ordine.

L'approccio generale in tutti gli esercizi sarà quello di inizializzare tutti i componenti necessari per l'approssimazione numerica, inserirli nell'oggetto `ODESolverBuilder` per costruire l'oggetto `ODESolver` e risolvere il sistema attravero metodo `Solve()` citato nella sezione precedente:

1. Viene fornita una funzione di tipo `ode::Function<T>` la quale rappresenta il sistema di equazioni differenziali.
   Esempio: Una particella carica in un campo elettrostatico uniforme monodimensionale sarà descritta dall'equazione differenziale:

   $$\begin{cases} \dot{x} = v \\ \dot{v} = \frac{q}{m}E \end{cases}$$

   La funzione di tipo `ode::Function<T>` sarà:

```
            // Tempo
auto f = [](T t,
           // Condizioni al tempo t
           Tensor<T> const& y) {
    // const T q = ..., m = ..., E = ...

    auto y_next = tensor::Tensor<double>::Vector(2);

    // dx/dt
    y_next(0) = y(1);
    // dv/dt
    y_next(1) = q / m * E;

    return y_next;
};

```

2. Vengono scelte le condizioni iniziali e immagazzinate in un oggetto di tipo `tensor::Tensor<T>`.
3. Viene scelto l'intervallo di tempo e lo step utilizzato (si dichiara un oggetto di tipo `func::Range<T>` di tipo `kFixed`)
4. Si sceglie il metodo numerico da utilizzare
5. Si risolve numericamente l'equazione differenziale chiamando il metodo `Solve()` ottenendo la matrice `result`.

6.3 Esercizi

6.3.1 Oscillatore approssimato

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

$$\ddot{\theta}(t) = -\theta(t) \tag{6.2}$$

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.

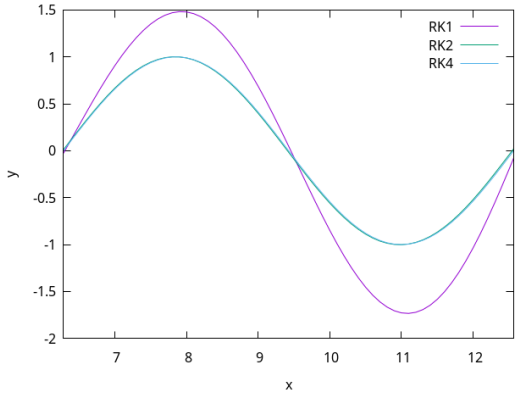


Figura 6.1: Confronto tra soluzioni numeriche con step $h = 0.1$

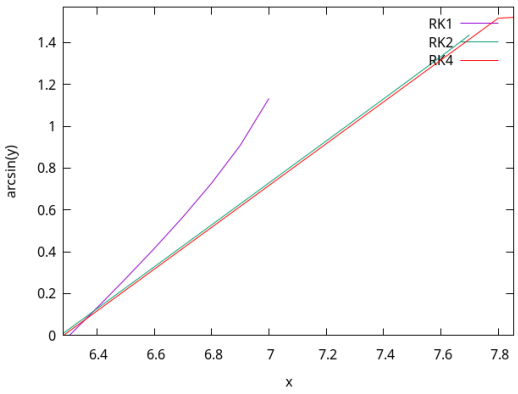


Figura 6.2: Visualizzazione alternativa per le soluzioni numeriche (step $h = 0.1$)

Osservazioni e analisi risultati

1. Risolvendo l'esercizio secondo il ragionamento citato in 5.2 si ottengono i grafici 6.1 e 6.2.
   Come si può notare il metodo di Eulero è già incoerente con il dominio dell'immagine della funzione $\sin(x)$ per valori in $x \in (2\pi, 4\pi)$ mentre i metodi di ordine successivo sono molto più precisi già a valori bassi di x e di h.
2. La risoluzione analitica è immediata:

   $$\theta(t) = Ae^{\lambda t} \rightarrow \lambda^2 + 1 = 0$$
   $$\theta(t) = Ae^{it} + Be^{-it}$$

   $$\begin{cases} \theta(0) = 0 \\ \dot{\theta(0)} = 1 \end{cases} \Rightarrow \begin{cases} A + B = 0 \\ Ai - Bi = 1 \end{cases} \Rightarrow \theta(t) = \sin(t)$$

   Un modo per studiare l'errore è quello di valutare la differenza $\eta(t_{max}, h) - \theta(t_{max})$ (poichè l'effetto di propagazione dell'errore sulla approssimazione è massimo per t_{max}).
   Campionando $(e(t_{max}, h_i), h_i)$ si ottiene la figura 6.3 si può notare che, come dettato dalla teoria, il metodo RK1 scala come h, RK2 come h^2 e RK4 come h^4.

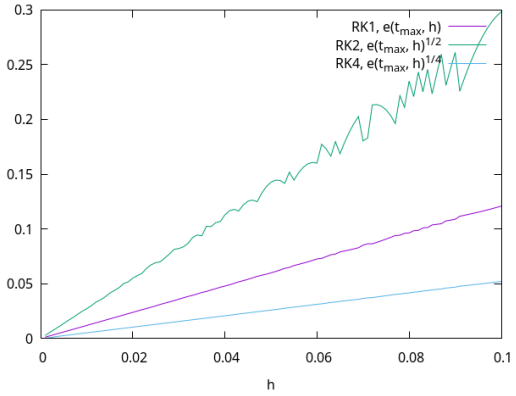


Figura 6.3: Associando alla funzione $e(t_{max}, h)$ l'inverso della funzione teoricamente associata si ottiene una relazione di proporzionalità diretta tra le funzioni inverse e h

6.3.2 Oscillatore reale

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

$$\ddot{\theta}(t) = -\sin\theta(t) \tag{6.3}$$

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

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

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

3. e un termine forzante:

$$\ddot{\theta}(t) = -\sin\theta(t) - \gamma\dot{\theta}(t) + A\sin\frac{2}{3}t$$

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

Implementazione

File necessari `damped_oscillator.cpp`

L'implmentazione necessita solamente di scrivere il sistema di equazioni differenziali, di seguito e' proposta l'implementazione del terzo punto:

```
tensor::Tensor<double> ForcedOscillator(double t,
                                        tensor::Tensor<double>
    const &y) {
  auto dydt = tensor::Tensor<double>::Vector(2);

  // Dividiamo l'equazione differenziale di secondo ordine
  // in due di primo ordine
  dydt(0) = y(1);

  // Le costanti vengono passate globalmente
  dydt(1) = -sin(y(0)) - kGamma * y(1) + kA * sin((2.0 / 3) * t);

  return dydt;
}
```

Osservazioni e analisi risultati

1. Si ottengono i seguenti risultati per il primo punto, si utilizza inoltre, per limitare i plot ottenuti, che il metodo utilizzato sia *RK4*:

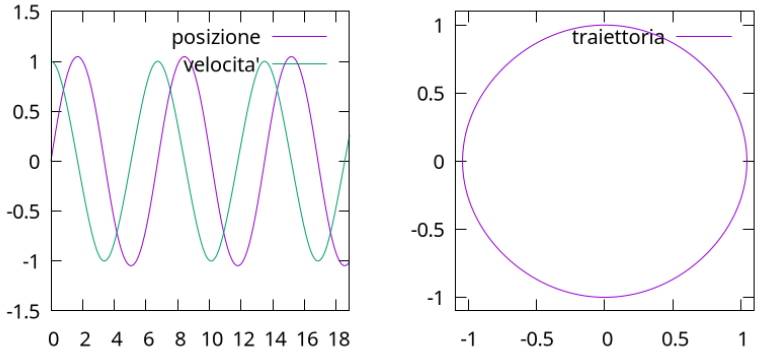


Figura 6.4: Oscillatore ideale imperturbato ($\theta(0) = 0, \dot{\theta}(0) = 1$), nel caso di equilibrio stabile. Nella prima figura si possono osservare $\theta(t)$, $\dot{\theta}(t)$, nel secondo $\dot{\theta}(\theta)$, lo spazio delle fasi

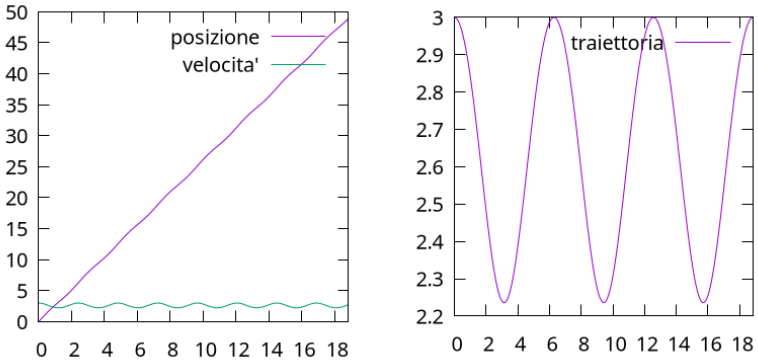


Figura 6.5: Oscillatore ideale imperturbato ($\theta(0) = 0, \dot{\theta}(0) = 3$), nel caso non in equilibro.

Come si puo' osservare lo spazio delle fasi e' composto principalmente da due casi di traiettorie: la prima in Figura 6.4 corrisponde ad un ellisse; mentre, se l'energia cinetica supera l'energia potenziale di carattere oscillatoria, si osserva il caso non in equilibrio: al di sopra e al di sotto delle possibili traiettorie ellissoidali si ottengono traiettorie sinusoidali come in Figura 6.5. I risultati corrispondono correttamete con la teoria.

2. Nel caso dell'oscillatore smorzato si ottengono i seguenti risultati:

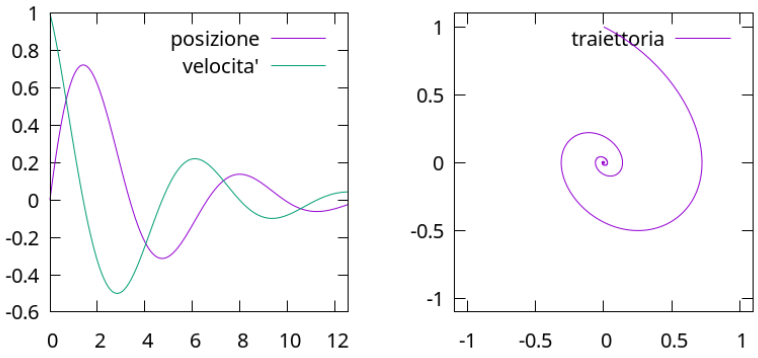


Figura 6.6: Oscillatore ideale smorzato ($\theta(0) = 0, \dot{\theta}(0) = 1$), nel caso $\gamma = 0.5$.

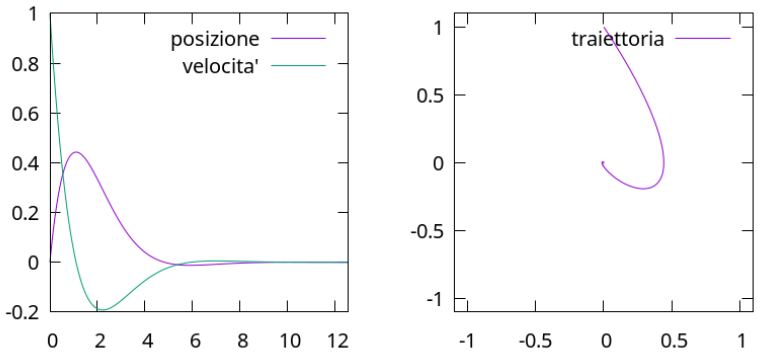


Figura 6.7: Oscillatore ideale imperturbato ($\theta(0) = 0, \dot{\theta}(0) = 1$), nel caso $\gamma = 1.5$.

Si osserva in questo caso che il termine di smorzamento influenza la traiettoria delle oscillazioni, in particolare, per $\gamma = 0.5$ si osserva un'oscillazione smorzata, mentre per $\gamma = 1.5$ si osserva un'oscillazione smorzata simil-critica. Lo spazio delle fasi assume una traiettoria a spirale: la spiegazione fisica e' molto semplice,

lo smorzamento porta ad un continuo rallentamento del sistema che e' destinato (qualsiasi siano i valori iniziali) a convergere ad un punto. Per questa ragione non vengono inclusi casi di valori iniziali differenti poiche' si otterra' sempre un carattere oscillatorio iniziale per sistemi inizialmente non stabili ma che cadranno necessariamente in una spirale simile alle sovracitate.

3. Si studia,infine, il caso di smorzamento forzato:

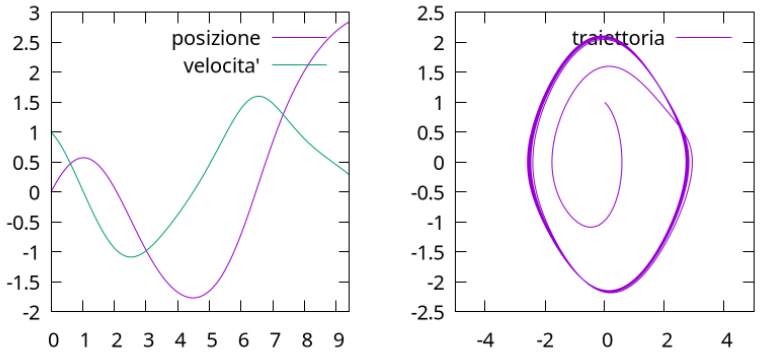


Figura 6.8: Oscillatore forzato ($\theta(0) = 0, \dot{\theta}(0) = 1$), nel caso $\gamma = 0.5, A = 0.5$.

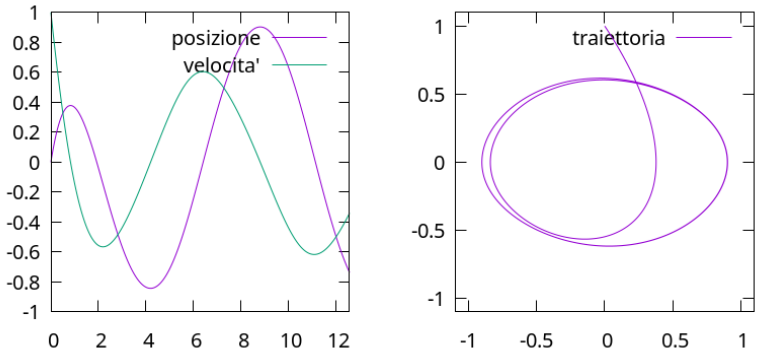


Figura 6.9: Oscillatore forzato ($\theta(0) = 0, \dot{\theta}(0) = 1$), nel caso $\gamma = 1.5, A = 1.5$.

In Figura 6.8 e in Figura **??** si evince chiaramente l'effetto del termine forzante: la traiettoria inizialmente a spirale si dilata e torna ad un caso ellissoide coerentemente con il caso in equilibrio imperturbato. La forzante, il termine A, definisce la nuova ampiezza di oscillazione stabile, mentre il termine γ definisce la velocità di convergenza al nuovo equilibrio.

6.3.3 Attrattore di Lorenz

Richiesta Studiare il sistema di ODE:

$$\begin{cases} \dot{x}(t) = 10(y(t) - x(t)) \\ \dot{y}(t) = 28x(t) - y(t) - xz(t) \\ \dot{z}(t) = -8/3z(t) + xy(t) \end{cases} \tag{6.4}$$

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).

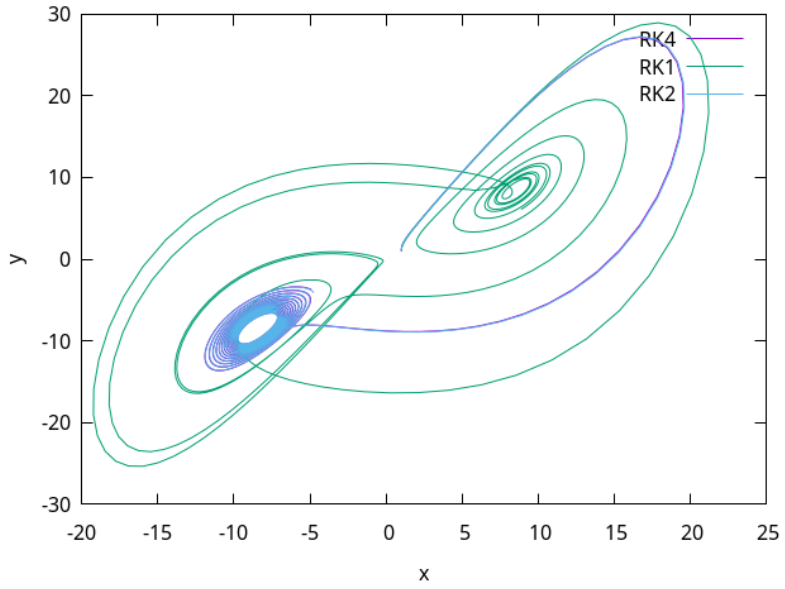
2. Tracciare il grafico di (x, y), (x, z) e (y, z).

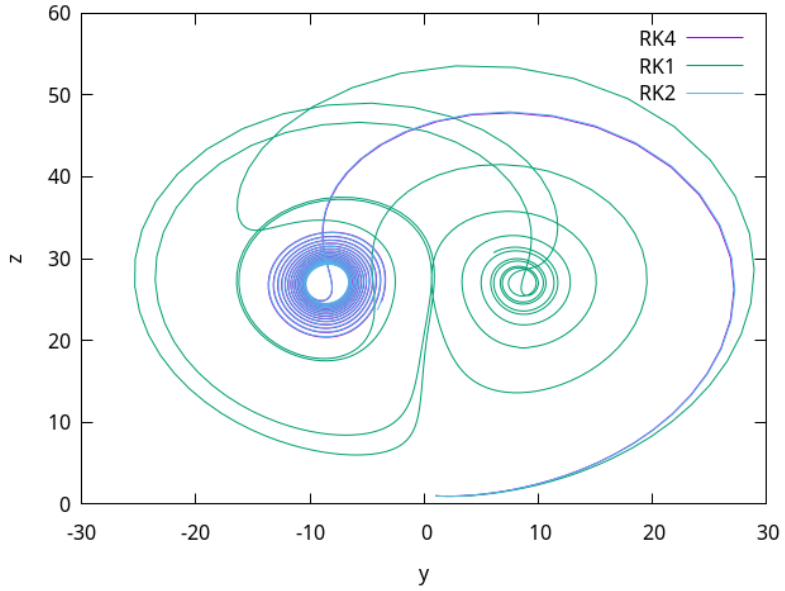
Implementazione

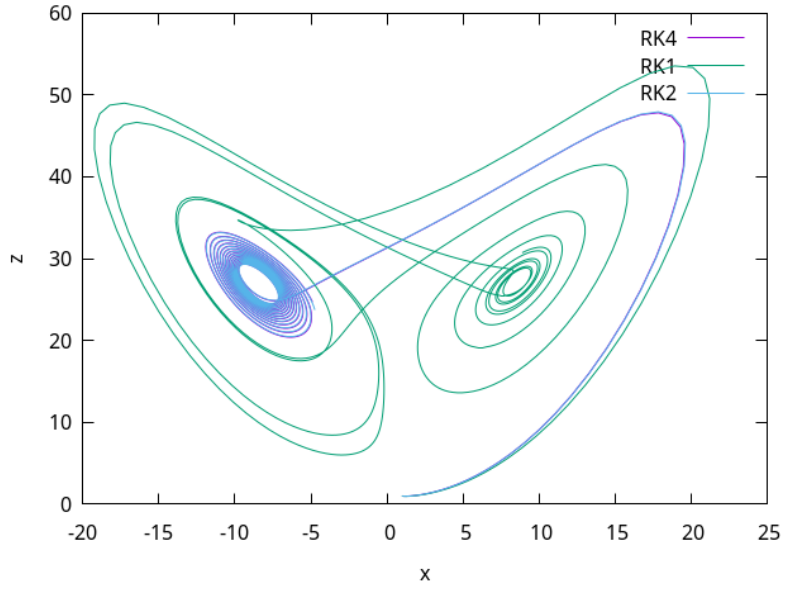
File necessari `lorenz.cpp`

L'implementazione del sistema di equazioni differenziali è immediata essendo gia' al primo ordine.

Osservazioni e analisi risultati Considerando un passo $\Delta t = 0.01$, un intervallo di tempo $t \in [0, 10]$ e valori iniziali $\vec{r}(0) = (0, 0, 0), \dot{r}(0) = (1, 1, 1)$ si ottengono le seguenti sezioni della soluzione:







Il risultato ottenuto e' il noto *Attrattore di Lorenz*, e' inoltre risaputo dalla teoria che il sistema di equazioni differenziali descritto sia caotico: piccole variazioni nei valori iniziali portano a grandi variazioni nel sistema, come si puo' notare dai grafici ottenuti, infatti la differenza dell'ordine del metodo (sopratutto nel metodo di Eulero) porta una variazione considerevole nello step successivo che porta quindi ad un risultato considerevolmente diverso dato un intervallo di tempo adeguato rispetto al differenza tra gli ordine dei metodi considerati. Inoltre, si osserva che il sistema e' invariante per rotazioni, infatti, le traiettorie ottenute sono simmetriche rispetto all'asse z.

6.3.4 Sistema a tre corpi

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

$$\ddot{\vec{x}}_i = \sum_{j \neq i} m_j \frac{\vec{x}_j - \vec{x}_i}{|\vec{x}_j - \vec{x}_i|^3} \tag{6.5}$$

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

- $m_1 = m_2 = m_3 = 1$
  $\vec{x}_1 = (1, 0, 0), \quad \dot{\vec{x}}_1 = (0, 0.15, -0.15)$
  $\vec{x}_1 = (-1, 0, 0), \quad \dot{\vec{x}}_1 = (0, -0.15, 0.15)$
  $\vec{x}_1 = (0, 0, 0), \quad \dot{\vec{x}}_1 = (0, 0, 0)$

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

1. Risolvere numericamente il sistema di equazioni.

2. Tracciare il grafico dell'energia totale del sistema in funzione del tempo.

Implementazione

File necessari `gravitation.cpp`

Nella matrice della soluzione si sceglie di ordinare la soluzione ad ogni time step nel seguente modo:

$$\begin{pmatrix} r_{1,x}(t_0) & r_{1,y}(t_0) & \dots & r_{3,z}(t_0) & v_{1,x}(t_0) & \dots & v_{3,z}(t_0) \\ \vdots & & & & & & \\ r_{1,x}(t_n) & r_{1,y}(t_n) & \dots & r_{3,z}(t_n) & v_{1,x}(t_n) & \dots & v_{3,z}(t_n) \end{pmatrix}$$

La funzione che definisce il sistema dovra' quindi tradurre l'equazione 6.5 in un sistema di equazioni differenziali al primo ordine che rispettino l'ordine della matrice:

```
...
  // Calcoliamo la accelerazione di interazione per ogni corpo
  auto grav_accell = [&](int i, int j) -> tensor::Tensor<double> {
    auto r_ij = tensor::Tensor<double>::Vector(3);
    for (int k = 0; k < 3; k++) {
      // Utilizziamo l'ordine specificato nella matrice
      r_ij(k) = y(3 * i + k) - y(3 * j + k);
    }

    double dist = r_ij.Norm();
    double dist_cubed = std::pow(dist, 3);

    auto accell = tensor::Tensor<double>::Vector(3);

    for (int k = 0; k < 3; ++k) {
      accell(k) = -masses(j) * r_ij(k) / dist_cubed;
    }

    return accell;
  };

  for (int i = 0; i < 3; ++i) {
    for (int k = 0; k < 3; ++k) {
      // Vengono aggiornate le posizioni
      dydt(3 * i + k) = y(3 * (i + 3) + k);
    }

    auto acc = tensor::Tensor<double>::Vector(3);
    // Utilizzando modulo 3 si riottengono gli indici delle
    posizioni
    // dei corpi e calcoliamo l'accelerazione
    auto acc1 = grav_accell(i, (i + 1) % 3);
    auto acc2 = grav_accell(i, (i + 2) % 3);

    for (int k = 0; k < 3; ++k) {
      // Si aggiornano le velocita'
      dydt(3 * (i + 3) + k) = (acc1 + acc2)(k);
    }
```

```
38 }
```

Si ottiene dunque la nuova riga della matrice dal vettore riga `dydt` allo step successivo.

Osservazioni e analisi risultati Utilizzando il metodo $RK4$ e step $\Delta t = 0.01$ si ottengono i seguenti risultati:

1. Dalle figure Figura 6.10 e Figura 6.11 si osserva che, data la simmetria dei dati iniziali, la traiettoria dei corpi e' simmetrica rispetto all'asse x. Inoltre, si osserva che l'energia totale del sistema e' conservata, come ci si aspetta dal sistema gravitazionale, per valori non troppo grandi di t, piu' aumenta il tempo piu' il trend dell'energia totale diminuisce seguendo una retta di pendenza negativa, l'approssimazione del metodo $RK4$ dissipa energia per valori grandi di t; si conferma, inoltre, la presenza di un sistema legato data la presenza di una energia totale negativa.
2. Nei grafici Figura 6.12 e Figura 6.13 si osserva che il sistema e' caotico e legato. Si ripresenta nuovamente la pendenza negativa dell'energia totale seppure con un coefficiente relativamente basso non apprezabile a queste scale.

Conclusioni I metodi Runge-Kutta utilizzati negli esercizi non garantiscono la conservazione dell'energia del sistema: quando i corpi si avvicinano molto tra di loro si perdono molte informazioni sul sistema, quest'ultime e l'intervallo di step utilizzato non sono adeguatamente distribuiti rispetto alla forza esercitata tra i due corpi nello step corrente. Utilizzando degli intervalli di tempo in funzione della distanza tra i corpi si otterrebbero misure piu' precise, utilizzando, come nel seguente esercizio, uno step fisso nel calcolo si perdono informazioni quando i corpi sono piu' vicini e quindi molto piu' veloci. Oltre a questa soluzione personalmente ipotizzata si possono utilizzare metodi che garantiscono la conservazione di dell'energia: i *Metodi Simplettici*

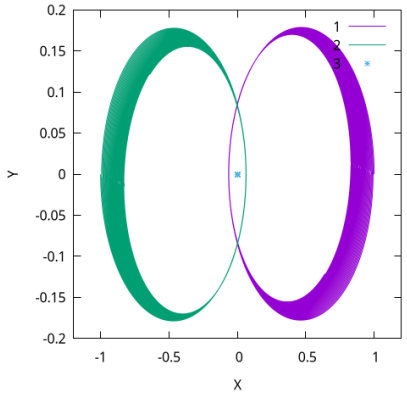


Figura 6.10: Sezione sul piano XY del primo punto

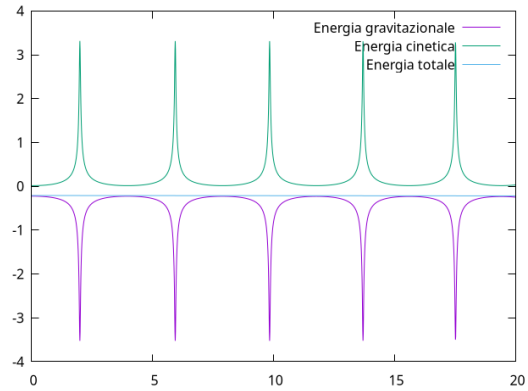


Figura 6.11: Grafico delle energie per il punto 1

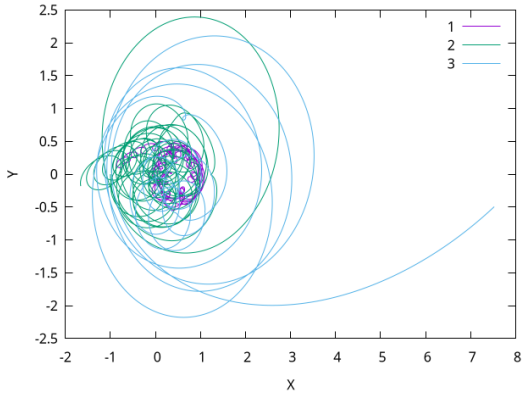


Figura 6.12: Sezione sul piano XY del primo punto

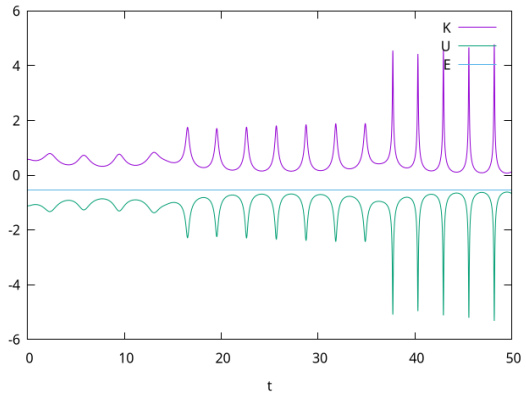


Figura 6.13: Grafico delle energie per il punto 2

Equazione di Schrödinger 7

7.1 Esercizi

7.1.1 Esercizio 16

Nozioni teoriche Si ottiene dunque, nel caso particolare del problema:

$$\left[-\frac{N^2}{16}K + \mathcal{V}\right]\phi = \mathcal{E}\phi \quad \mathcal{E} = EL, \mathcal{V} = VL$$

con $K_{ii} = -2, K_{j+1,j} = K_{j,j+1} = 1 \quad 0 \leq j < N-1,\ 0 \leq i < N$.
$V = -V_0$ per $1/4 \leq x \leq 3/4$

Implementazione

File necessari `ex_16.cpp`

L'implementazione segue direttamente dalla equazione sopracitata applicando il *Power Deflation Method*

Analisi dei risultati e conclusioni

1. Si veda `ex_16.cpp`
2. Come ci si aspetta si ottengono stati legati che qualitativamente possono essere considerati corretti: i nodi corrispondono all'energia dello stato; all'interno della parete la funzione d'onda scala come un'esponenziale; nei casi al di sopra della parete si ottengono oscillazioni a frequenza piu' alta per $E - V$ piu' grandi.
3.
4.
5.

7.1.2 Esercizio 17

Nozioni teoriche

Implementazione

Analisi dei risultati e conclusioni

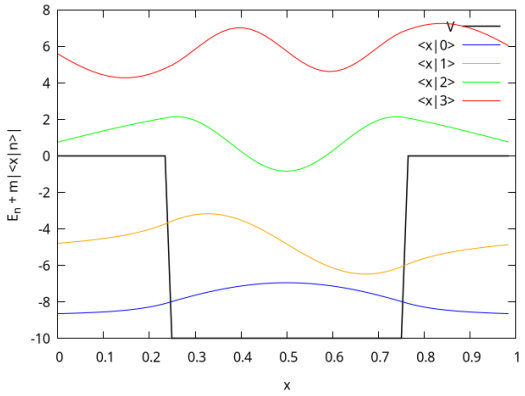


Figura 7.1: Autofunzioni della buca finita di potenziale

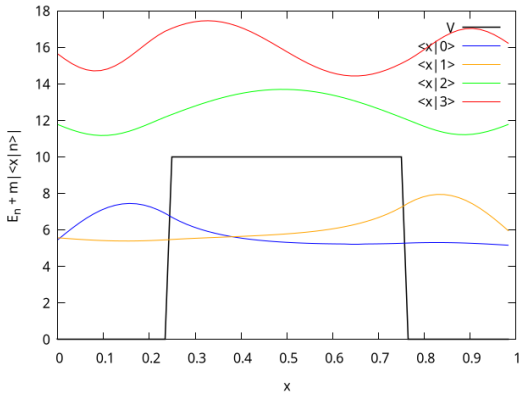


Figura 7.2: Autofunzioni della buca finita di potenziale

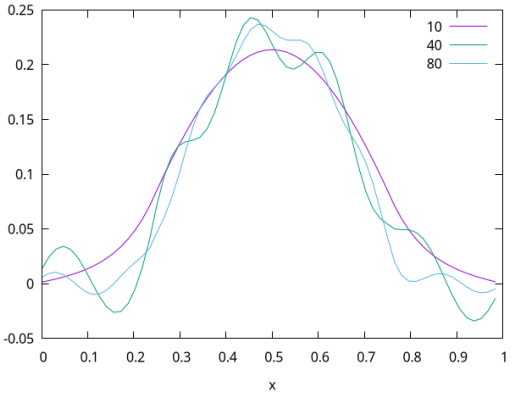


Figura 7.3: Autofunzioni della buca finita di potenziale

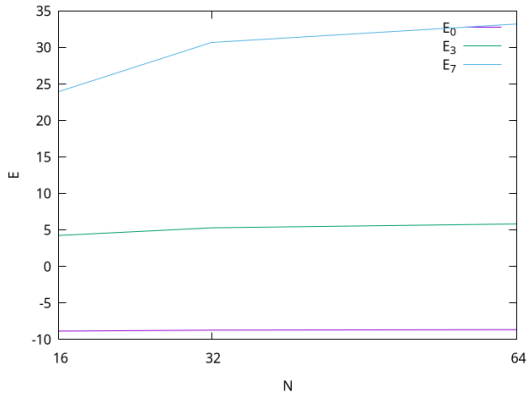


Figura 7.4: Autofunzioni della buca finita di potenziale

Metodi di integrazione 8