Appunti del Corso

Laboratorio di Trattamento Numerico dei Dati Sperimentali

Leonardo Alchieri

Università degli Studi di Milano

Indice

Prefazione

Questi appunti sono stati prodotto da Leonardo Alchieri per se stesso. Sono pieni di errori e imprecisioni, oltre a essere incompleti in alcune parti. Si sconsiglia fortemente chiunque volesse utilizzarli come unico materiale didattico per il corso; si suggerisce l'utilizzo come *appunti* di ripasso.

Si noti che le sezioni degli appunti sono organizzate sia tematicamente, che per data della lezione. Questo fa si che in alcune istanze vi siano delle ripetizioni, o alle volte alcuni argomenti vengono esplicati nel corso di più lezioni.

Capitolo 1

Introduzione

1.1 6/10/2017

In questa lezione si rivede un po' tutto, dall'inizio.

In programmazione si prende da una **scrittura** sorgente, scritta in un particolare linguaggio, e poi viene compilata dal calcolatore (nel nostro caso usiamo come compilatore gcc), che, semplificato, esegue 3 processi:

- preprocessing - tutte le istruzioni che devono essere preparate (tutto ciò che ha # davanti)
- compilazione - si trasforma quello che è scritto in linguaggio macchina, tipicamente un object file
- linking - si connettono tra di loro di diversi object files creati

In ultima parte si ha quindi l'**esecuzione**, cioè si dice al compilatore di eseguire i file compilato (cioè quando si usa il comando ./<nome_ del_ file>.

Vediamo un esempio:

```
#include <iostream>        //input - output su codice
#include <fstream>        // input - output su file

using namespace std;         //non si richiede di porre
    std::  davanti a tutto

int main(){                 //scope del main
    double a = 2;
    double b = 3;

    double result = a + b;

    cout<< "THE SUM IS "<< result <<endl;    //le funzioni
        cin e cout sono in grado di determinare quale tipo
        si stia scrivendo a video.
```

```
    fstream outfile;                    //questa funzione
        serve a scrivere i risultati anche su file esterno

    outfile.open("ent.txt" ; ios::out);    //outfile \'e
        un oggetto di tipo fstream. Il metodo open permette
         di associare all'oggetto outfile ad un determinato
         file, nel coso 'ent.txt'

    outfile << result <<endl;           //si dice al
        compilatore di scrivere result dentro ad outfile,
        che e' poi linkato con il file

    outfile.close();
    outfile.clear();

    return 0;
}
```

Listing 1.1: sum.cxx

Si noti che la parte di codice dentro due parentesi graffe viene definito ***scope***. Il simbolo # è una **direttiva di pre-processing**: esse servono a comunicare al compilatore di andare a "raccogliere" i fali desiderati, come librerie di sistema (i.e. `iostream` e `fstream`) o altri *header file* (`.h`), prima della compilazione vera e propria del programma.

Se si volesse utilizzare una funzione propria e non di sistema:

```
# include "my library.h"
```

Se uno andasse a cercare, all'interno del Computer sono contenute le librerie già scritte si possono andare a trovare.

In teoria, si potrebbe creare un proprio **namespace**, (diverso da std::), dove andare a richiamare le funzioni della standard library sotto la nomenclatura `std::`.

Analizziamo la struttura di definizione di una variabile:

```
( TYPE ) nome _ variabile = valore
```

questo tipo di struttura è sempre utilizzato all'interno del C++. I tipi possono essere molteplici:

- **int**
- **unsigned int**
- **float**
- **double**

```
#include <iostream>
#include <fstream>

using namespace std;

int main(){

    double array[5] = {1,2,3,4,5};        //creo un
        vettore di 5 numeri in double

    cout<< array[4] << endl;            //dico di stampare
         uno specifico elemento del vettore

    return 0;
}
```

Listing 1.2: Esempio di creazione di un array. Si ricordi che gli indici di un array partono sempre da 0 e vanno fino alla grandezza del vettore.

```
#include <iostream>
#include <fstream>

using namespace std;

int main(){

    int k = 3;
    cout << k << endl;
    cout << &k << endl;         //dico al Computer di
        scrivere la posizione della variabile, cioe' la
        casella di memoria riservata alla variabile int  k

    int * pk;                    //questa Ã¨ una variabile
         che contiene un indirizzo di memoria ad un intero.
    k=3;

    pk = &k;

    cout<< *pk <<endl;           //viene scritto a schermo
        il valore 3, poiche' pk punta alla stessa casella
        di memoria di &k.

    * pk = 10;                 //l'operatore *, operatore
        di dereferenziazione, Ã¨ l'inverso di &. Sto
        dicendo cioÃ¨ di prendere il valore dato dall'
        indirizzo del puntatore
    cout << k << endl;           //poiche' sopra ho
        modificato con l'operatore di dereferenziazione il
        valore contenuto nella casella di memoria a cui
        puntava pk, sarÃ  scritto a schermo 10

    return 0 ;
```

```
}
```

Listing 1.3: Esempio di utilizzo di puntatori

Il calcolatore, quando assegna una ”casella” a una variabile, segna anche la posizione; cioè nella stessa cella di memoria è presente sia il valore che la posizione della stessa. Per accedere all'indirizzo di memoria della casella si usa l'**operatore di referenza**, & . Si noti che le posizioni sono riportare in numeri esadecimali.

Un variabile puntatore (cioè di tipo **<TYPE> ***), può essere associata anche un array, vettore:

```
double * point = array;
```

Per il calcolatore un array non risulta altro che un puntatore al primo elemento della sequenza: esso viene trattato quindi come una quantità di memoria contigua. Quando si va ad utilizzare un qualsiasi tipo di array, difatti il calcolatore "si porta appresso" solamente il puntatore alla prima area di memoria.

```
#include <iostream>
#include <fstream>

using namespace std;

int main(){
    int N;
    cin >> N;

    double * array = new double[N];        //sto dicendo
        di definire l'array a posteriori. L'operatore new
        mi alloca dinamicamente la memoria: questa memoria
        viene quindi riservata non nella stack, cioÃ¨
        quella che viene usata alla compilazione, ma nella
        ip, cioÃ¨ quella riservata all'esecuzione.

    for(...) {
        ...
    }

    delete [] array;
    return 0;
}
```

Listing 1.4: Esempio di allocazione dinamica

Definire la lunghezza di un array a posteriori tramite una variabile, si sbaglia: è importante però che la lunghezza dell'array sia definita quando si compila il programma; con una variabile a posteriore, il compilatore non sa cosa fare. Quindi IL SEGUENTE CODICE E‘ SBAGLIATO:

int N;
cin » N;
double array[N];

Nel momento in cui permettiamo al programma di "uccidere" la variabile array creata (cioè all'uscita dello scope), dico al calcolatore di cancellare il puntatore alla variabile. Rimane cioè un'area di memoria allocata nel programma: mi sono cioè "ucciso" il puntatore, ma ho lasciato la memoria, che rimane inaccessibile. Ogni volta che si alloca dinamicamente un vettore, bisogna forzare l'istruzione di cancellare l'aria di memoria associata al puntatore:

```
delete[] array;
```

```
#include <iostream>
#include <fstream>

using namespace std;

double medie ( int , double *);

int main(){
    int N;
    cin>> N;

    double * array = new double [N];

    [...]          //input da file nell'array

    cout << medie (N, array) << endl;        //sto
        utilizzando una funzione medie

    return 0;
}

medie ( int n , double * aux) {            //funzione, che
     puo essere accessibile anche ad altri file. Si
    potrebbe anche mettere dentro anche un'altra
    double sum;

    for(...){                //cicli per la media
        ...
    }

    return sum / n;
}
```

1.2 13/10/2017

1.2.1 Overloading

Si vede ora il concetto di **overloading**.

```
double medie ( double e, double b );
double medie ( int N, double * V );
double medie ( int N, int * V);
```

Listing 1.5: Esempio di tre funzioni in overloading

Si supponga che, in uno stesso programma, si voglia una stessa funzione (i.e. medie) per fare più cose. Il compilatore è in grado di riconoscere la differenza tra le variabili che vengono mandate alle funzioni, e rimandare alla funzione corretta.

Si possono quindi definire più funzioni con uno stesso nome, aventi metodi differenti: questa caratteristica, propria del `C++`, non è però presente in tutti i linguaggi di programmazione - per esempio è assente nel `C`.

La differenziazione tra le diverse variabili avviene durante la **fase di compilazione** del programma, e non in esecuzione.

1.2.2 Template

Vediamo ora l'approccio all'utilizzo dei **template**.

```
template <typename V> double medie( int N, V a) {
    (...)       //esecuzione del codice
    return sum/N;
}
```

Listing 1.6: Esempio banale dell'uso di una funzione con un generico template

Il Template è in grado di riconoscere se viene mandato dentro un intero int oppure un double/float. V dunque risulta essere il tipo di variabile che viene mandato dentro alla funzione.

Questo è molto utile quando, più avanti, andremo ad utilizzare l'oggetto vector, che permette di "inscatolare" qualsiasi tipo di elemento.

1.2.3 Macro

```
#ifndef _MEDIA_H_
#define _MEDIA_H_

double medie ( int, double *);

```

```
#endif  /*_MEDIA_H*/        //nota che #endif non vuole
    alcun argomento
```

Listing 1.7: Esempio di utilizzo di macro

Ogni file `.h` che viene utilizzato, può essere complicato impacchettando quando è contenuto nel file attraverso delle **macro**, specifiche direttive di **direttive di pre-processing**. Quello che si compie è andare a definire un particolare tipo di variabile **macro**, tramite il comando `define`. Come si può notare nell'esempio sopra, è presente davanti a questa il comando `ifndef`: questo comando fa si che si esegua quanto "scritto" sotto a esso solamente se la condizione viene soddisfatta, nello specifico se è stata o meno definita una variabile. Difatti, il procedimento computazionale che avviene è il seguente: una prima volta di lettura dell'*header file*, poiché non è stata definita alcuna variabile, è soddisfata la condizione di ingresso; a questo punto viene definita la variabile, il compilatore "legge" il codice seguente, fino alla condizione `endif`. A questo punto, se per qualche motivo il compilatore andasse a prendere lo stesso *header file*, la seconda volta la condizione di ingresso non verrebbe soddisfatta - è stata già definita la variabile - e quindi viene completamente saltato tutto il pezzo di codice contenuto fino a `endif`.

1.2.4 Visibilità

```
#include <iostream>

int pippo = 0;            //variabile definita a scope
    globale (si puo' fare)

int main(){
    int pluto = 1;

    for( int k=0; k<10; k++){
        int pluto = 0;            //sto ridefinendo la
            variabile! Dentro il for vale 0 ma fuori varr\'
            a 1
        int paperino = 0;
    }

    cout << paperino;        //errore!!! paperino esiste
        solo nello scope del for.
    cout << pippo;           //stampa a schermo 0
    cout<< pluto;           //stampa a schermo 1
}
```

Listing 1.8: Esempio degli scope e della visibilità delle variabili al suo interno.

Si noti l'importanza della definizione delle variabili dentro degli scope locali; queste sono facile da usare e da controllare. Le variabili globali invece sono molto più difficili da trattare, e spesso si applica l'utilità del namespace per lavorarci.

1.2.5 Namespace

```
namespace test1 {
    int e = 1;

}

namespace test2 {
    int e =5;
}

int main(){

    cout << test1 :: e;         //mi rida' a schermo il
        valore 1
    cout << test2 :: e;         //mi rida' a schermo il
        valore 5

}
```

Listing 1.9: Esempio di definizione di un namespace

L'operatore di **scope**, cioè il **namespace**, permette di definire qualcosa posto all'interno di un determinato scope. Nell'esempio, quando scrivo `testt1 :: e`, dico alla macchina di andare a prendere la variabile `e` posta dentro lo *scope test1*.

Usando il comando `using namespace (nome_ namespace)`, si forza la convenzione di usare un certo scope. È quanto si va sempre a fare quando definiamo `using namespace std;`, cioè si dice al computer di andare a prendere il `cout` (e le altre cose) poste dentro lo scope di sistema (che è nella libreria `<iostream> std`).

Capitolo 2

Programmazione ad oggetti

2.1 20/10/2017

Vediamo come primo esempio il calcolo di un campo elettrico formato da un dipolo.

Si può pensare il CampoVettoriale come un oggetto dove è depositata una posizione e il valore del campo in quel punto.

Tutta l'informazione viene immagazzinata in una sequenza di oggetti e classi, non dentro il main. Si "maschera" la complicazione delle classi all'interno di altri file: si sostituisce il codice piano in maniera razionale, organizzandolo in classi (pezzi di codice), che sono connessi in una certa maniera a quello che si vuole fare.

Si prova ora a scrivere il codice in maniera più precisa:

```
#ifndef _POSIZIONE_H
#define _POSIZIONE_H

class posizione {
public:

    //nota che il nome dei costruttori deve essere lo
        stesso della classe
    posizione();                              //queste
        funzioni sono dette costruttori
    posizione ( double x, double y, double z);    //
        overloading (si possono rimandare o una posizione
        oppure niente)

    double distanza (const posizione &) const;    //una
        funzione per calcolare la distanza tra 2 posizioni

    double getX () const {return m_x ; } ;        //si usa
         const perch\' e il metodo non deve essere in grado
         di modificare il valore
    double getY () const {return m_y ; } ;
```

```
    double getZ () const {return m_z ; } ;

    void setX (double x) { m_x = x ;};        //si sta
        usando una implementazione in-line (la funzione \'e
         talmente semplice)
    void setY (double x) { m_x = x ;};        //queste
        funzioni servono per cambiare i valori DOPO che
        sono stati costruiti
    void setZ (double x) { m_x = x ;};        // ; si
        usano, cio\'e, quando nel main voglio modificarli
        successivamente

private:
    double m_x, m_y, m_z;
};
```

Listing 2.1: posizione.hpp

```
#include "posizione.hpp"

posizione :: posizione () {        // se non vengono
    passati argomenti, si imposta la posizione a 0
    m_x=0;
    m_y=0;
    m_z=0;
}

posizione :: posizione (double x, double y, double z) {
    m_x = x;            //si costruiscono i valori che
        vengon passati.
    m_y = y;
    m_z = z;
}

double posizione :: distanza (const posizione & P ) const
    {
    double dist = sqrt ( pow ( m_x - P.getX(), 2) + pow (
        m_y - P.getY(), 2) + pow ( m_z - P.getZ(), 2) ;
        // ricorda il <cmath>.
        // questa funzione permette di calcolare la
            distanza. P \'e il secondo punto (di cui con
            get prendo le coordinate)
    return dist;
}
```

Listing 2.2: posizione.cpp

```
#include "posizione.hpp"

int main() {
    posizione P1;                              //creo un
        oggetto di tipo posizione (a cui non passo
        argomenti). Di default, a va a prendere il
        costruttore vuoto
```

```
    posizione P2 (1,1,1);                    //creo un
        oggetto, a cui dico di prendere il costruttore con
        argomenti
    posizione * P3 = new posizione (2,2,2);    //ho
        definito in questo caso il punto con un puntator

    cout << P1.gettX() << P2.getY() << P3 -> getZ() ;
    cout << P1.distanza(P2);

    posizione P4 (P1);                    // si chiama
        copy constructor, cio\'e si costruisce P4 come
        costruito P1
    posizione P5;
    P5 =P4;                              //l'operazione \'e
        diversa dal copy constructor; in questo caso pone
        le stesse coordinate a P5


    return 0;
}
```

Listing 2.3: main_ cpos.cpp

```
#ifndef _CAMPO_H
#define _CAMPO_H

#include "posizione.hpp"

class CampoVettoriale : public  posizione {    //dico che
    la classe CampoVettoriale eredita dalla classe
    posizione tutte le sue funzioni dentro public
    // aggiungo solo quello che la classe CampoVettoriale
        ha rispetto alla classe posizione.
public:
    CampoVettoriale (double x, y, z, Fx, Fy, Fz);
    // la prima definizione definisce il campo vettoriale
        in una generica coordinata (x,y,z) che deve essere
        data (cio\'e non \'e legata a un punto P come sotto
        )
    CampoVettoriale ( posizione P, double Fx, double Fy,
        double Fz);        //nota che si definiscono i
        costruttori in over-loading
    //la seconda definizione permette di costruire un
        CampoVettoriale sopra una posizione che si ha giÃ
        a dispozione
private:
    double m_Fx, m_Fy, m_Fz;
}

#endif  /* _CAMPO_H*/
```

Listing 2.4: campo.hpp

```
CampoVettoriale :: CampoVettoriale (double x, y, z, Fx, Fy
    , Fz)  : posizione (x,y,z) {
    m_Fx = Fx;
    m_Fy = Fz;
    m_Fz = Fy;
}
```

Listing 2.5: `campo.cpp`

Nel primo file `posizione.hpp`, si esegue le costruzione e definizione di tutti i metodi della classe posizione. Come si può notare, essa presente due differenti istanze: una `public` e una `private`; tutto ciò che è contenuto nella prima istanza - generalmente costruttore e altri metodi - può essere utilizzato da codice esterno alla classe - in particolare nel `main`; tutto ciò che invece viene definito all'interno dell'istanza `private` ha utilizza ristretto esclusivamente alla classe stessa - si sta difatti utilizzando la proprietà di **incapsulamento** dell'**object oriented programming** in `C++`.

Oltre alle due istanze, `public` e `private`, descritte sopra, esiste inoltre una terza istanza, `protected`: questa si comporta esattamente come il `private`, cioè mantenendo le variabili ivi definite incapsulate, ma permette la condivisione delle stesse con tutte le **classi figlie** - o derivate - della classe stessa.

Poiché le variabili definite nell'istanza `private` non sono modificabili da codice esterno alla classe, da un punto di vista pratico - e come mostrato anche nell'esempio, si creano dei metodi `public` che ne permettino la modifica tramite richiamo di funzione.

Le funzioni `getX(Y,Z)` restituiscono dentro il main il valore della posizione. Si noti che la definizione `const` protegge da eventuali tentativi di modifica della variabile all'interno del metodo.

I **costruttori** sono delle speciale funzioni che vengono richiamate per costruire l'oggetto. Questi tipi di funzioni non hanno alcun tipo di ritorno, perché servono per costruire l'oggetto. (Nota che sopra si sta definendo posizione per overloading).

Senza un costruttore, si ha un costruttore di default (che però non si sa come vada a costruire gli oggetti).

Il campo vettoriale deve essere una posizione (x,y,z) e una terna che rappresenta il valore del campo lungo le coordinate.

2.2 27/10/2017

Si formalizza l'utilizzo dei costruttori delle classi; si possono eseguire differenti cose:

- Nessuna implementazione: in questo caso, e in presenza di variabili nel `private`, il compilatore va in automatico ad allocare generiche aree di

memoria per costruire tali variabili; questo non permette quindi di conoscere a priori i valori contenuti in tali caselle di memoria.

- Implementare un costruttore, che prenda in ingresso alcune variabili che serviranno a "costruire" le varie variabili presenti nell'istanza `private`.
- Si può eseguire overloading del costruttore, cioè scrivo differenti costruttori in maniere differenti. Per esempio si potrebbe rimettere anche un costruttore senza argomenti.
- Si usa un copy constructor, con un operatore di assegnazione (=) di default.

Capitolo 3

Ricerca zeri di una funzione

3.1 3/11/2017

Si mostrano ora i metodi numerici per il calcolo di **zeri di funzioni**. Questi metodi consistono nell'approssimazione, a diversi gradi, del valore reale, e della determinazione dello stesso errore commesso.

Teorema 3.1.1 *Bolzano-Weierstrass*
Data f continua su $[a, b]$ tale che $f(a) \cdot f(b) < 0$,

$$\Rightarrow \exists x_0 \in [a, b] : \ f(x_0) = 0$$

3.1.1 Metodo di bisezione

Il più semplice metodo per il calcolo degli zeri di una funzione, e difatti anche la più banale dimostrazione del *Teorema* sopra enunciato, è il **metodo di bisezione**.

Si esplica prima il concetto di **velocità di convergenza**:

$$x_n \xrightarrow[n \to +\infty]{} x_*$$

*con **ordine p** se :*

$$\forall k > 0, \ \exists c > 0 : \ \frac{|x_{n+1} - x_*|}{|x_n - x_*|^p} \leq c, \ \forall n \geq k$$

Questo metodo è così strutturato:

- Si controlla la validità del *Teorema degli zeri* in un certo intervallo $[a, b]$.
- Si calcola il valore della funzione nel punto medio di tale intervallo, cioè $f(\frac{b-a}{2})$.

- Tale valore risulta quindi l'approssimazione dello zero cercato. Si procede poi successivamente in maniera iterativa, diminuendo sempre più gli intervalli di ricerca.

Come si può facilmente notare, questo tipo di metodo presenta un andamento dell'errore, al passo n-esimo, del tipo $err \simeq \frac{b-a}{2^n}$. Esso è infatti un metodo non particolamente veloce per la ricerca di una zero, ma ha la propriteà di garantire sempre che si ritrovi uno zero - cioè il metodo è convergente per qualsiasi funzione continua su qualsiasi intervallo.

Vedremo in tal fine altri due algoritmi per un miglior calcolo degli zeri: **metodo delle secanti** e **metodo di Newton**; quello che però succede è che solo il metodo di bisezione permette di avere convergenza globale: negli altri due metodi, si potrebbe avere divergenza per alcuni casi.

A seconda di p, la convergenza viene detta:

- $p = 1$, convergenza lineare
- $1 < p < 2$, convergenza superlineare
- $p = 2$, convergenza quadratica

Vediamo l'ordine di convergenza del metodo di bisezione:

$$|x_{n+1} - x_*| \simeq \frac{b-a}{2^{n+1}}$$

$$|x_{n+1} - x_*| \leq c|x_n - x_*|^p$$
$$\Rightarrow \frac{b-a}{2^{n+1}} \leq c\left(\frac{b-a}{2^n}\right)^p$$

cioè:

$$\frac{b-a}{2} \leq c\frac{(b-a)^p}{2^{n(p-1)}} \;\Rightarrow \forall n,\; p = 1 \wedge c = \frac{1}{2}$$

3.1.2 Metodo delle secanti

Con il metodo delle secanti si può però raggiungere una migliore convergenza.

Questo metodo funziona così: al primo passaggio si definiscono due punti, x_0 e x_1„ e si approssima la ricerca dello zero con una secante; cioè lo zero, x_2, diventa il punto in cui la secante tra x_0 e x_1 interseca l'asse delle x; a questo punto si guarda $f(x_2)$ e si va avanti. Vediamo una formalizzazione dell'algoritmo.

$$P(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0)$$

$$P(x) = 0 \;\Rightarrow\; x = x_1 - \frac{f(x_1)}{f(x_1) - f(x_0)}(x_1 - x_0) \;= x_2$$

iterando il procedimento si giunge alla formula generale:

$$x_{k+1} = x_k - \frac{f(x_k)}{f(x_k) - f(x_{k-1})}(x_k - x_{k-1})$$

3.1.3 Metodo di Newton

Il **metodo di Newton**, detto anche *delle tangenti*, risulta invece:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

In tal caso, quello che si sta eseguendo è la tangente a un punto e si interseca sull'asse delle x per trovare uno zero. Si può verificare che la **convergenza** di questo metodo sia **quadratica**, ma non è globalmente garantita; quello che si può fare è porre protezioni all'algoritmo, che però ovviamente ne riducono la velocità di calcolo.

Per il calcolo della derivata si può solamente utilizzare la definizione, procedimento che non è particolarmente efficacie:

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h}$$

Il principale problema che si riscontra è dovuto alla scelta della variabile h: infatti, il calcolatore, essendo finito, avrà un errore di arrotondamento intrinseco, che limita la grandezza del numero che si può prendere il considerazione

$$e_R \sim \varepsilon_M \frac{f(x_0)}{h}$$

Inoltre, si esegue in questo calcolo un errore di troncamento, dovuto all'approssimazione della funzione:

$$e_T \sim \frac{1}{2} h f''(x_0)$$

Unendo questi due errori si ha una stima dell'errore totale che si commette:

$$e_{TOT} \sim \epsilon_M \frac{f(x_0)}{h} + \frac{1}{2} h f''(x_0)$$

Volendo, si potrebbe cercare di ottimizzare questo errore, cioè:

$$\frac{de_{TOT}}{dh} = 0$$

e si ottiene che:

$$e_{TOT}^{(best)} \simeq k\sqrt{\epsilon_M}$$

In teoria, per un migliore calcolo della derivata, si potrebbe eseguire uno Sviluppo di Taylor ad ordini più grandi.

Esistono inoltre altre tecniche, dette **metodi di estrapolazione di Richardson**, ma il loro studio è oltre gli obiettivi di questo corso.

Vediamo cosa fare per scrivere un codice che permetta di eseguire queste cose:

```
class Parabola{
public:
    double Evaluate (double x) {return pow(m_a* x,2) + m_b
        *x +m_c;};

private:
    double m_a, m_b, m_c;
};
```

Listing 3.1: `parabola.hpp`

```
class Bisezione{
public:
    double CercaZeri (double Xmin, double Xmax, const
        Parabola *P);


};
```

Listing 3.2: `bisezione.hpp`

```
double Bisezione:: CercaZeri (double Xmin, double Xmax,
    const Parabola *P){

    //questo metodo accetta per\'o solamente una classe di
        tipo "Parabola". Sarebbe meglio per\'o
       determinarlo per tanti tipi di classi

    double zero;
    if( P -> Evaluate(Xmin) * P -> Evaulate(Xmax) < 0) {
        [...]
    }
    return zero;
}
```

Listing 3.3: `bisezione1.cpp`

Si usa qua successivamente la proprietà del **polimorfismo**, tipica del C++: un puntatore a una classe derivata è compatibile con un puntatore a classe madre.

```
class FunzioneBase {
public:
    virtual double Evaluate (double x) = 0;

    // =0 significa che deve essere implementato nelle
       classi figlie; esso \'e detto metodo virtual puro (
       cio\'e questo metodo non fa niente)

    //virtual serve per richiamare pi\'u volte il nome
       Evualuate sulle classi derivate; devo per\'o
       scriverlo, altrimenti quando punto a una classe
       derivata che ha questo metodo, quello non
       funzionerebbe

};
```

Listing 3.4: `FunzioneBase.hpp`

```
//classe astratta
class Parabola: public FunzioneBase{   //sfrutto il
   polimorfismo
public:
    double Evaluate (double x) {return pow(m_a* x,2) + m_b
       *x +m_c;};

private:
    double m_a, m_b, m_c;
};
```

Listing 3.5: `parabola.hpp` con polimirfismo

```
FunzioneBase * f = new Parabola(...);  //questo \'e il
   polimorfismo

cout << f -> Evaulate(10) << endl;  //in questo caso va a
   usare il metodo della classe Parabola

//si noti che tale per fare questa cosa, si possono usare
   solo i puntatori
```

Listing 3.6: `main_ p.cpp`

```
class Bisezione{
public:
    double CercaZeri (double Xmin, double Xmax, const
       FunzioneBase *P);
    //qua sfrutto bene il polimorfismo

    //in questo caso, possono passare sia la classe "
       Parabola" che qualsiasi altra classe; scrivere il
       medoto di questa maniera fa si che esso sia
       indipendente dalla classe che viene invocata.
```

```
};
```

Listing 3.7: `bisezione.hpp`

Una classe che include almeno un **metodo virtuale puro** (con =0 alla fine, cioè che deve per forza lavorare con delle funzioni derivate) è detta **classe astratta**: questo permette di rappresentare una generica funzione.

Capitolo 4

Makefile

Disclaimer: parte di quanto scritto qua sotto potrebbe essere sbagliato o inesatto, in quanto frutto soprattutto di ricerca personale dopo le lezioni.

4.1 10/11/2017

Si mostra ora una maniera abbreviata per la compilazione di file attraverso l'uso del *Makefile*:

```
LDLIBS = -lstdc++
#specifica una libreria di tipo c++

CXXFLAGS = -g

esercizio: main.cc #file.o#
    g++ -Wall -o $@ $^
%.o:  %.cc %.hh
    g++ -Wall -c -o $@ $<
```

Listing 4.1: Costruzione di un Makefile abbreviato

Si esplicano le diverse funzionalità di quest script. L'uso di `CXXFLAGS = -g` serve per attivare di default, nella compilazione, le funzionaltà di debugging nel programma; su sistemi *Unix*, come *Linux*, il programma di *debugging* presente è richiamato attraverso il comando `gdb`; su **MacOs**, esiste un programma simile noto come *LLDB*.

Successivamente, si dice invece al compilatore come creare i file prima oggetto `.o`, e quindi l'eseguibile `esercizio`. Con l'uso del simbolo `%` viene indicato che si va a compilare un generico file che abbia lo stesso nome dell'associato file `.cc` e `.hh`. Sotto, si indica dunque al compilatore come eseguire questa costruzione: attraverso `-Wall` si indica si utilizzare quando

scritto sopra per la compilazione, `$@` indica il target desiderato (e quindi il *file* oggetto), mentre `$<` indica il fatto che questa sia una prima dipendenza.

Successivamente, si indica dunque al compilatore come andare a creare il *file* eseguibile desiderato, attraverso tutti i *file* oggetto già a disposizione. In tal caso, bisogna però specificare la presenza del file `main.cc`, cioè quello in cui sia presente la funzione `main` del programma; inoltre, l'uso di `$^` indica il fatto che questa costruzione non abbia dipendenza e utilizze costruzioni che presentano dipendenze.

Capitolo 5

Calcolo di Integrali

5.1 17/11/2017

5.1.1 Metodi numerici

Teorema 5.1.1 (esistenza e unicità del polinomio interpolante) *Dati n punti di interpolazione, (x_i, f_i) $i = 1, \dots, n$ con $x_i \neq x_j$ $\forall i \neq j$,*

$$\Rightarrow \exists! p \in P_n : \ p(x_i) = f_i, \ \forall i = 1, \dots, n$$

Una maneria per poter approssimare la mia funzione, si può creare una combinazione lineare di polinomi di Lagrange:

$$l_j^{(n)}(x) = \prod_{k=1 \wedge k \neq j}^{n+1} \frac{x - x_k}{x_j - x_k}$$

Quello che dunque si ottiene è qualcosa del tipo:

$$\int_a^b f(x)dx \simeq \int_a^b \sum_{i=1}^{n+1} f(x_i) l_i^n dx = \sum_{i=1}^{n+1} f(x_i) \int_a^b l_i^n dx$$

L'esecuzione di tale procedimento è noto come **Formula di Newton-Cotes**.

Dunque, si ottiene che si deve solamente eseguire l'integrale dei Polinomi di Lagrange. Vediamo dunque alcuni casi particolari di questa riscrittura generale.

- $n = 0$ allora polinomio di grado 0, e dunque si ha la **mid point formula**, cioè:

$$f(x_M) = f_M$$

$$\Rightarrow \int_a^b f(x)dx \simeq \int_a^b P_0(x)dx = \int_a^b f_M dx = (b-a) f_M$$

- $n = 1$, allora polinomio di grado 1, e dunque $n + 1 \to 2$. Si hanno in questo caso due polinomi di lagrange:

$$l_1(x) = \frac{x-b}{a-b}\ l_2(x) = \frac{x-a}{b-a}\ \Rightarrow P_1(x) = f_a\frac{x-b}{a-b} + f_b\frac{x-a}{b-a}$$

  L'integrale dunque risulta:

$$\int_a^b f(x)dx \simeq \int_a^b P_1(x)dx = \int_a^b \left(f_a\frac{x-b}{a-b} + f_b\frac{x-a}{b-a}\right)dx = \frac{f_a+f_b}{2}(b-a)$$

  cioè si è approssimata l'area come quella di un **trapezio**. Questa formula è dunque chiamata **Formula dei Trapezi**.

- $n = 2$, si hanno dunque 3 punti in cui valutare la funzione.

$$a, x_M, b\ \Rightarrow\ \int_a^b f(x)dx \simeq \frac{b-a}{3}\left(f(x) + 4f(x_M) + f(b)\right)$$

  Questa formula è detta **Formula di Cavalieri-Simpson**.

Cerchiamo di determinare **l'errore** di interpolazione quando utilizziamo un polinomio interpolante. Questo è affermato dal seguente teorema:

Teorema 5.1.2 *Data una formula di Newton-Cotes su nodi $x_i = a + ih$, con $h = \frac{b-a}{n}$, con $i = 1, \ldots, n$:*

- *Se n è pari e $f \in \mathcal{C}^{(n+2)}([a,b])$, l'errore risulta:*

$$\varepsilon = \frac{f^{(n+2)}(\eta)h^{n+3}}{(n+2)}\int_a^b \prod_{j=0}^{n} t(t-j)dt$$

- *Se n è dispari e $f \in \mathcal{C}^{(n+3)}([a,b])$, l'errore risulta:*

$$\varepsilon = \frac{f^{(n+1)}(\xi)h^{n+2}}{(n+1)}\int_a^b \prod_{j=0}^{n} (t-j)dt$$

Una formula che ha **grado di precisione** k è *esatta* quando la funzione integranda è polinomio di grado k ed $\exists$ il polinomio di grado $k+1$ per cui l'errore è *non nullo*.

Si nota facilmente che le formule di grado pari risultano di un grado di precisione più grande.

Vediamo alcuni esempi:

- **Midpoint**
  - defisco $h = \frac{b-a}{N}$

GRADO	PUNTI	ERRORE
0	1	$-\frac{f''(\xi)}{24}(b-a)^3$ *midpoint*
1	2	$-\frac{f''(\eta)}{12}(b-a)^3$ *trapezi*
2	3	$-\frac{f^{(4)}(\xi)}{90}(b-a)^5$ *Simpson*

Figura 5.1: Grado di approssimazione dell'integrale e relativo errore

  - $x_k = a + \left(k + \frac{1}{2}\right) h \quad k = 0, \ldots, N-1$
  - $\int_a^b f(x)dx \simeq h(f(x_0) + f(x_1) + \cdots + f(x_{N-1}))$
  - l'errore in questo caso andrà come $\varepsilon \sim h^2$

- **Trapeziode**
  - definisco $h = \frac{b-a}{N}$
  - $x_k = a + kh \quad k = 0, \ldots, N$
  - $\int_a^b f(x)dx \simeq h\left(\frac{1}{2}f(x_0) + f(x_1) + \cdots + f(x_{N-1}) + \frac{1}{2}f(x_N)\right)$

- **Simpson**
  - definisco $h = \frac{b-a}{N}$
  - $x_k = a + kh \quad k = 0, \ldots, N$
  - $I_N = \frac{h}{3}\left(f(x_0) + 4f(x_1) + 2f(x_2) + \ldots + 4f(x_{N-3}) + 2f(x_{N-2}) + 4f(x_{N-1}) + f(x_N)\right)$

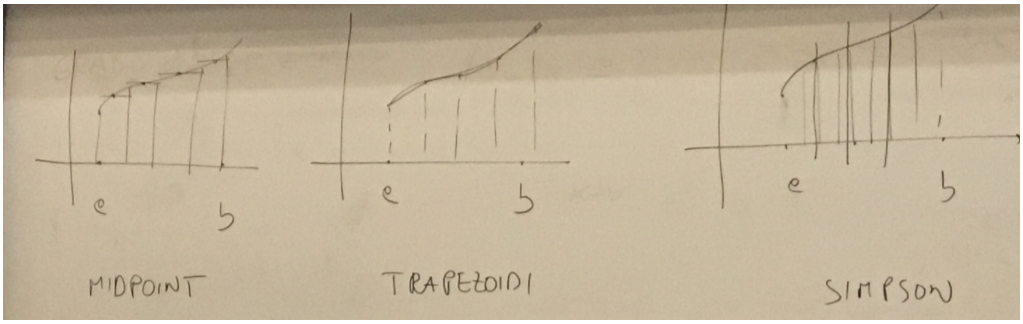

Figura 5.2: Rappresentazione Grafica delle approssimazioni

Idealmente, si vorebbe calcolare un algoritmo con una precisione desiderata, in modo tale che il calcolatore compili il programma fino ad averla raggiunta.

Vediamolo per calcolare un integrale a precisione fissata per il trapezoide:

- $\varepsilon = k_1 h^2 + k_2 h^4 + k_3 h^6 + \ldots$

$$I - I_h = k_1 h^2 + k - 2h^4 + \ldots$$

$$I - I_{h/2} = k_1 \left(\frac{h}{2}\right)^2 + k_2 \left(\frac{h}{2}\right)^4 + \dots$$

dove I ovviamente è il valore reale dell'integrale. Sottrando la seconda dalla prima, si ottiene:

$$I_{h/2} - I_h = k_1 h^2 - \frac{1}{4} k_1 h^2 + h^4 =$$

$$= \frac{3}{4} k_1 h62 + h^4$$

Dunque:

$$\varepsilon \simeq k_1 h^2 = \frac{h}{3}(I_{h/2} - I_h)$$

5.2 1/12/2017

5.2.1 Metodi Monte-Carlo

Oltre ai metodi numerici per il calcolo di integrali visti, esistono tecniche di integrazione basate sulle generazione di numeri casuali, detti **metodi Monte-Carlo**. La loro applicazione è particolamente utile nel calcolo di integrali in più dimensioni, in quanto, come si vede in seguito, la loro precisione rimane invariata all'aumentare delle dimensioni utilizzate.

Vediamo cosa avevamo eseguito l'altra volta:

- Abbiamo implementato un **generatore congruente lineare**:

```
double Random::rnd(){
    unsigned int x;
    x = (m_a*m_seed + m_c) % (m_m);
    m_seed = x;
    return (double)x / (m_m - 1);
}
```

Listing 5.1: Metodo per generare numeri pseudo-casuali dentro la classe "Random"

  Si noti che, volendo, dentro le librerie di **ROOT** sono presenti già dei metodi per creare dei numeri casuali.

- L'uso del metodo della trasformata per determinare un numero casuale secondo una data distribuzione di probabilità.

Esiste una tecnica che permette il passaggio alla Gaussiana in 2 dimensioni:

$$f(x_1, x_2) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x_1^2}{2}} \frac{1}{\sqrt{2\pi}} e^{-\frac{(x_1^2 + x_2^2)}{2}}$$

Con il metodo della trasformata, che consiste nell'integrazione due volte:

$$F(x_1, x_2) = \int_{-\infty}^{x_1} dx'_1 \int_{-\infty}^{x_2} dx'_2 \frac{1}{\sqrt{2\pi}} e^{-\frac{(x'^2_1 + x'^2_2)}{2}}$$

Si può eseguire in questo caso un cambio di variabile, lungo le coordinate polari (r, θ):

$$F(r, \theta) = \int_0^{\theta} \left(\int_0^{r} \frac{r}{'} 2\pi e^{-\frac{r'^2}{2}} dr' \right) d\theta' = \frac{\theta}{2\pi} \left(1 - e^{-\frac{r^2}{2}} \right)$$

Definendo $t \in [0,1] : \ t = \frac{\theta}{2\pi}$ e $s \in [0,1] : \ (1-s) = 1 - e^{-\frac{r^2}{2}}$, e tornando quindi alle variabili x_1 e x_2:

$$x'_1 = \sqrt{-2 \ln s} \cos(2\pi t)$$

$$x'_2 = \sqrt{-2 \ln s} \sin(2\pi t)$$

Questa è nota come **trasformazione di Box-Müller**.

In questa maniera si va a distribuire le due variabili casuali x'_1 e x'_2 secondo una distribuzione Gaussiana.

Il metodo **Accept-Reject**, già implementato in laboratorio, si basa sul fatto che vengono estratti più volte le x che hanno una probabilità maggiore: si genera una x, si genera una y, e si accetta x solamente se $y < f(x)$, con f funzione data. Questo è un metodo del tutto generale, che va bene per qualsiasi funzione (anche se bisogna conoscere il *range* di numeri e il massimo della funzione).

Teorema 5.2.1 (Legge dei grandi numeri) *Data una variabile aleatoria x e una distribuzione di probabilità $P(x)$, con valore di aspettazoine $\mu = \int_{-\infty}^{+\infty} xP(x)dx$ (somma finita se discreta) e varianza $\sigma = \int_{-\infty}^{+\infty} (x - \mu)^2 P(x)dx \ < +\infty$ (somma finita se discreta), se si costruisce la successione:*

$$Y_N = \frac{x_1 + \ldots + x_N}{N}$$

cioè la sommatoria N numeri,

allora

$$Y_N \xrightarrow[N \to +\infty]{} \mu$$

$$\sigma_N^2 \xrightarrow[N \to +\infty]{} \frac{\sigma^2}{N}$$

Teorema 5.2.2 (del Limite Centrale) *Sulle ipotesi della Legge dei Grandi Numeri, si ha che la variabile Y_N si distribuirà semrpre secondo una Gaussiana:*

$$P(Y_N) = \frac{1}{\frac{\sigma}{\sqrt{N}}\sqrt{2\pi}} e^{-\frac{(Y_N - \mu)^2}{2\frac{\sigma^2}{N}}}$$

Un buon metodo per calcolare un integrale può essere quello di **Hit or Miss** :

- estraggo $x \in [a, b]$ $unif.$
- estraggo $y \in [0, f_{max}(+\varepsilon)]$ $unif.$
- N_{ext}`++`
- `if (` $y < f(x)$`)` N_{hit} `++`

La stima dell'integrale risulta dunque:

$$\hat{I} = (b-a) f_{max} \frac{N_{hit}}{N_{ext}}$$

Si mostra ora come mai questo metodo converga esattamente all'integrale e come si possa avere una stima dell'errore che si va a commettere al passo N-esimo.

- Si prende una variabile aleatoria $x_i = \begin{cases} successo \ +1 \\ insuccesso \ 0 \end{cases}$
- $p = \frac{A_{int}}{A_{tot}}$ al sucesso e $(1-p)$ all'insuccesso.
- $\mu = p$
- Costruisco $Y_N = \sum_i x_i \ (= N_{hit})$
- Per il **Teorema del Limite Centrale**:

$$N\mu = Np = N\frac{A_{int}}{A_{tot}}$$

$$\Rightarrow \ A_{int} = \frac{N_{hit}}{N_{tot}} \cdot A_{tot}$$

- $\sigma^2_{Y_N} = N\sigma^2_x \ \Rightarrow \ \sigma_I = \frac{\sigma_{Y_N}}{N} A_{tot} = \frac{1}{\sqrt{N}} \sigma_x \cdot A_{tot}$

Vediamo ora il **Metodo della Media**, ancora più semplice degli altri:

- estrati $x_i \in [a, b]$ $unif.$
- calcolo $f(x_i)$
- Faccio N valutazioni distinte, ne faccio una media, la moltiplico per la larghezza dell'intervallo; questo approssima l'integrale:

$$\frac{\sum_i f(x_i)}{N}(b-a) = I_N \xrightarrow[N \to +\infty]{} I$$

5.3 6/12/2017

Riprendiamo brevemente il **metodo della media** (attenzione, diverso dal *mid-point*). Questo metodo prevede infatti un campionamento casuale, cioè di tipo statistico, mentre il *mid-point* ha un andamento regolare.

Avevamo sintetizzato brevemente come:

1. Estrarre $x_i \in [a, b]$ uniformemente
2. Calcolare $f(x_i)$
3. L'integrale risulta:

$$I_N = \frac{\sum_i f(x_i)}{N}(b-a) \xrightarrow[N\to+\infty]{} I$$

dove I_N è la stima dell'integrale con N estrazioni di punti

Mostriamo ora che lo stimatore I_N, con $N \to +\infty$ converga effettivamente al valore vero dell'integrale.

Consideriamo una variabile aleatoria x, distribuita con una certa distribuzione $P(x)$ (tendenzialmente noi lavoriamo con distribuzioni uniformi), applichiamo $f(x) = y$. Anche questa nuova variabile sarà distribuita secondo una sua distribuzione $G(y) = P(x)\frac{dx}{dy}$.

Si può quindi definire la variabile:

$$y_N = \frac{\sum_i y_i}{N} \xrightarrow[N\to=+\infty]{} < y >= \int_{-\infty}^{+\infty} yG(y)dy = \int_{-\infty}^{+\infty} f(x)P(x)dx$$

Per il **Teorema del Limite Centrale**.

Prendendo una distribuzione $P(x)$ del tipo:

$$P(x) = \begin{cases} \frac{1}{b-a} & x \in [a, b] \\ 0 & elsewhere \end{cases}$$

Quindi, il valore medio $< y >$ risulterà:

$$< y >= \frac{1}{b-a}\int_{-\infty}^{+\infty} f(x)dx$$

Da questo, se costruisco lo stimatore I_N come:

$$I_N = \frac{\sum_i f(x_i)}{N}(b-a) \xrightarrow[N\to+\infty]{} \int_a^b f(x)dx$$

ovviamente se la distribuzione dei numeri causuali sia di tipo uniforme.

Il **Teorema del Limite Centrale** garantisce anche l'errore che si compie:

$$\sigma^2_{I_N} \to \frac{\sigma^2_f}{N}(b-a)$$

Quello che si può dire sarà che la *deviazione standard* che si associa all'integrale risulta:

$$\sigma_I = \frac{\sigma_f (b-a)}{\sqrt{N}}$$

cioè presenta un andamento lento $\frac{1}{\sqrt{N}}$: più aumentano il numero di punti, migliore sarà la stima dell'integrale. Associare un errore così ad un integrale di tipo Montecarlo.

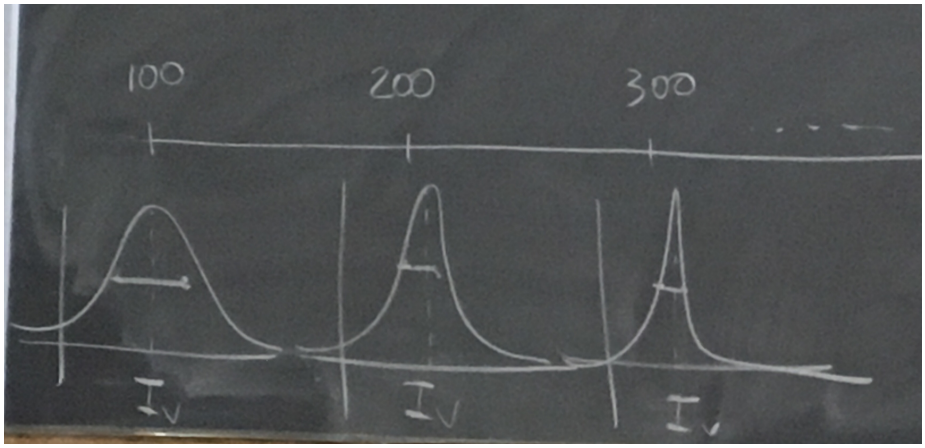

Figura 5.3: Senso dell'errore associato all'integrale Monte Carlo con il metodo della media

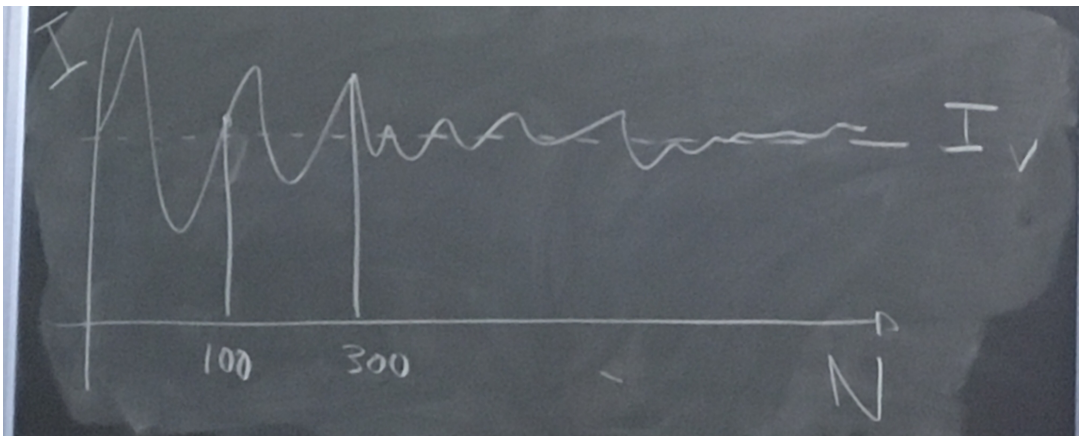

Figura 5.4: Grafico dell'andamento della precisione. Ovviamente i valori però tendono ad oscillare (in valore infatti viene sempre da una Gaussiana, però ha sempre un margine per cambiare)

Quello che si potrebbe fare sarebbe andare a stimare la costante dell'errore $\sigma = \frac{k}{\sqrt{N}}$. Per stimarla, potrei costruire la Gaussiana data dall'integrale N' calcolato, stimare la costante $k = \sigma \cdot \sqrt{N'}$.

(sto calcolando un integrale su N N' volte)

Si potrebbe anche fare un'operazione più semplice, cioè calcolare la deviazione standard delle funzione f

$$\sigma_f = \sqrt{\frac{1}{N-1}\sum_i (f_i - < f >)^2}$$

Calcolo del potenziale gravitazione di una sfera Il potenziale della sfera risulta:

$$V(\underline{r}) = \int dx'dy'dz' \frac{G \cdot \rho(\underline{r}')}{|\underline{r} - \underline{r}'|}$$

*** Vai a rubare i file che ha fatto vedere a lezione ***

Simulazione di un apparato sperimentale La simulazione che andremo a fare sarà quella dell'indice di rifrazione del prisma in funzione della lunghezza d'onda (lo abbiamo già fatto in laboratorio).

Capitolo 6

Equazioni Differenziali

6.1 15/12/2017

Il problema che si cerca di affrontare è quello di risolvere un **Problema di Cauchy**:

$$\begin{cases} y'(x) = f(x, y(x)) \\ y(x_0) = y_0 \end{cases}$$

Se la funzione f risulta *Lipschitziana* su un intervallo [a,b], con $f : \mathbb{R} \to \mathbb{R}$. In tali casi, vale infatti in **Teorema di Esistenza e Unicità**.

Teorema 6.1.1 (Di esistenza e Unicità) *Dato un problema di Cauchy per f. Se f è continua in un rettangolo aperto di $\mathbb{R}$ e Lipschitziana in y,*

$$\Rightarrow \forall (x_0, y_0) \in \mathbb{R},\ \exists! y(x) \quad soluzione \quad : y(x_0) = y_0$$

La soluzione di questo problema, a differenza dell'integrale, deve ridarmi una funzione ben definita. Il metodi più semplici da implementare con un calcolatore si basano sulla creazione della soluzione "per punti". Parto quindi da un punto certo, cioè la condizione iniziale del **Problema di Cauchy**, e inizio a "muovermi" in maniera iterativa intorno alla condizione iniziale.

Otterrò dunque come soluzione una collezione di punti che rappresentano la mia funzione.

La maniera più comoda e semplice è proseguire utilizzando *Taylor*:

$$y(x_0 + h) = y(x_0) + y'(x_0) \cdot h + \frac{1}{2} y''(x_0) \cdot h^2 + \dots$$

Si hanno a questo punto due strade da seguire, per rendere la soluzioni migliore:

- Diminuire il passo h
- Aumentare il troncamento

La maniera più semplice e intuitiva, che si basa su dati che si hanno già a disposizione, è calcolare solo fino al termine di grado 1. Questo è noto come **Metodo di Eulero**.

$$y_{n+1} = y_n + hy'(x_n) + O(h^2) \qquad \text{(Metodo di Eulero)}$$

6.1.1 Metodi one-step

Il metodo di Eulero, come altri, appartiene a una classe di metodi, noti come *one-step*, cioè del tipo:

$$y_{n+1} = y_n + h\Phi(x_n, y(x_n), h)$$

dove Φ è una funzione specifica.

Si definisce **l'errore locale di troncamento** t come segue:

$$\forall (x_0, y_0) \in \mathbb{R}, \quad t(x_0, y_0, h) = \frac{y(x_0 + h) - [y_0 + h\Phi(x_0, y_0, h)]}{h}$$

Difatti, si stima la differenza tra la soluzione esatta e la mia approssimazione.

Si definisce la **Consistenza** di un metodo come:

Φ è consistente di ordine P se $\forall (x_0, y_0) \in \mathbb{R}$ $t(x_0, y_0, h)$ è un $O(h^p)$ quando $h \to 0$.

Il metodo di Eulero, come si vede banalmente, è consistente di ordine 1.

Definisco **Convergenza** di un metodo come:

Φ è covergente in $[a, b]$ se

$$\forall x \in [a, b], \quad y_N := y_0 + \sum_{j=0}^{N-1} h\Phi(x_j, y_j, h), \quad h = \frac{x - a}{N}$$

e risulta che:

$$\lim_{N \to +\infty} y_N = y(x)$$

Teorema 6.1.2 *Se Φ è continua in $\mathbb{R} \times [0, h_0]$, **ALLORA**:*

- *Se Φ è Lipschitziana in y si ha che consisteza $\Leftrightarrow$ convergenza.*
- *Se l'ordine è P, l'errore sarà limitato come $|y(x_n) - y_N| < kh^p$.*

Ordine 2, Metodo di Runge

Cerchiamo ora di determinare un metodo di ordine 2.

$$y(x_0 + h) = y(x_0 + \frac{h}{2}) + \frac{h}{2}y'(x_0 + \frac{h}{2}) + \frac{h^2}{4}\frac{1}{2!}y''(x_0 + \frac{h}{2}) + O(h^3)$$

$$y(x_0) = y(x_0 + \frac{h}{2}) - \frac{h}{2}y'(x_0 + \frac{h}{2}) + \frac{h^2}{4}\frac{1}{2!}y''(x_0 + \frac{h}{2}) + O(h^3)$$

Ho difatti eseguito Taylor a destra e sinistra del punto a metà tra x_0 e $x_0 + h$. In questo caso, poiché il calcolo diretto della derivata seconda risulterebbe troppo oneroso da un punto di vista computazionale, si mostra di seguito un trucco per ottenere un errore dell'ordine $O(h^3)$ ovviando al problema della derivata seconda.

Eseguendo la differenza:

$$y(x_0 + h) = y(x_0) + hy'(x_0 + \frac{h}{2}) + O(h^3)$$

Riferendosi al problema di Cauchy, ottengo che posso scrivere il metodo come:

$$y(x_0 + h) = y_0 + hf\left(x_0 + \frac{h}{2}, y_0 + \frac{h}{2}f(x_0, y_0)\right) \qquad \text{(Runge)}$$

Questo metodo risulta consistente, convergente, di ordine 2 e con un errore dell'ordine kh^2.

Formalizzando come algoritmo per il compilatore, si ha:

$$\begin{aligned} y_{n+1} &= y_n + hk_2 \\ k_1 &= f(x_n, y_n) \\ k_2 &= f(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1) \end{aligned}$$

Ordine 4, Metodo Runge-Kutta

Vediamo ora un algoritmo per il calcolo di ordine 4:

$$\begin{aligned} k_1 &= f(x_n, y_n) \\ k_2 &= f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_1\right) \\ k_3 &= f\left(x_n + \frac{h}{2}, y_n + \frac{h}{2}k_2\right) \\ k_4 &= f\left(x_n + h, y_n + hk_3\right) \\ y_{n+1} &= y_n + h \cdot \left(\frac{k_1}{6} + \frac{k_2}{3} + \frac{k_3}{3} + \frac{k_4}{6}\right) \end{aligned}$$

L'errore, secondo quanto dice la teoria, va come $err = kh^4$. Si mostra come si possa ottenere.

Si prende quindi la stima della soluzione a passo h e $\frac{h}{2}$.

$$\begin{aligned} x(t = \bar{t})_h - x_v &= kh^4 \\ x(t = \bar{t})_{\frac{h}{2}} - x_v &= k\left(\frac{h}{2}\right)^4 \end{aligned}$$

Faccio la differenza e stimo l'errore:

$$x(t=\bar{t})_h - x(t=\bar{t})_{\frac{h}{2}} = \frac{15}{16}kh^4$$

Purtroppo, spesso, scrivere tutto questo in un sistema di classi potrebbe risultare parecchio complicato.

```
// FunzioneLineareBase.hh
class FunzioneLineareBase{
public:
    virtual VettoreLineare Eval(double t, VettoreLineare)
        =0;

};

//OscillatoreArmonico.hh
class OscillatoreArmonico: public FunzioneLineareBase{
public:
    VettoreLineare Eval(double t, VettoreLineare);

};

//EquazioneDifferenzialeBase.hh

class EquazioneDifferenzialeBase {

public:
    virtual VattoreLineare PASSO(double t, VettoreLineare
        vl, FunzioneLineareBase * f, double h);
};
```

Listing 6.1: Esempio di costruzione delle classi.