

Summary sulle formule di quadratura di Newton-Cotes

Grado Polinomio	Nr. Punti	Errore	Grado	Errore composta	Formula	Tipo
0	1	$-\frac{(b-a)^3 f''(\xi_1)}{24}$	1	$\sim h^2$	Midpoint	Aperta
1	2	$-\frac{(b-a)^3 f''(\xi_2)}{12}$	1	$\sim h^2$	Trapezoidi	Chiusa
2	3	$-\frac{(b-a)^5 f^{IV}(\xi_3)}{2880}$	3	$\sim h^4$	Simpson	Chiusa

Andamento dell'errore in funzione del passo di integrazione

```
#include "Integrale.h"
#include "Funzioni.h"

#include <cmath>
#include <iostream>
#include <iomanip>

#include "TGraph.h"
#include "TApplication.h"
#include "TAxist.h"
#include "TPad.h"
#include "TCanvas.h"
#include "TLegend.h"

using namespace std;

int main () {
    TApplication app("app",0 ,0);
    xsinx f ;

    TGraph gSimpson;
    TGraph gMidpoint;

    double a=0., b=M_PI/2., esatto=1. ;
    double h = 0;

    Midpoint MidpointIntegrator(a,b);
    Simpson SimpsonIntegrator(a,b) ;

    int nstep = 4 ;
    for ( int k = 0 ; k< 20 ; k++ ){
        h = fabs(b-a) / nstep;
        gSimpson.SetPoint(k, h, fabs(SimpsonIntegrator.Integra(nstep,f)-esatto));
        gMidpoint.SetPoint(k, h, fabs(MidpointIntegrator.Integra(nstep,f)-esatto));
        nstep*=2 ;
    }

    Deposito l'errore dentro al TGraph (possiamo
    scriverlo anche in un file per un utilizzo successivo)
}
```

- Codice per studiare l'andamento dell'errore in funzione del passo h per i diversi metodi studiati : vale nel caso si conosca il valore esatto dell'integrale !

```
TCanvas can;
can.cd();
can.SetLogy();
can.SetLogx();
can.SetGridx();
can.SetGridy();

gSimpsonSetTitle("Errore calcolo integrali");
gSimpson.GetXaxis()->SetTitle("Passo h");
gSimpson.GetYaxis()->SetTitle("Errore");
gSimpson.SetMinimum(1E-16);
gSimpson.SetMaximum(1.);
gSimpson.SetLineColor(2);
gSimpson.SetMarkerStyle(20);
gSimpson.Draw("ALP");

gMidpoint.SetLineColor(3);
gMidpoint.SetMarkerStyle(21);
gMidpoint.Draw("sameLP");

TLegend leg(0.1,0.7,0.3,0.9);
leg.AddEntry(&gSimpson,"Simpson","lp");
leg.AddEntry(&gMidpoint,"Midpoint","lp");
leg.Draw("same");

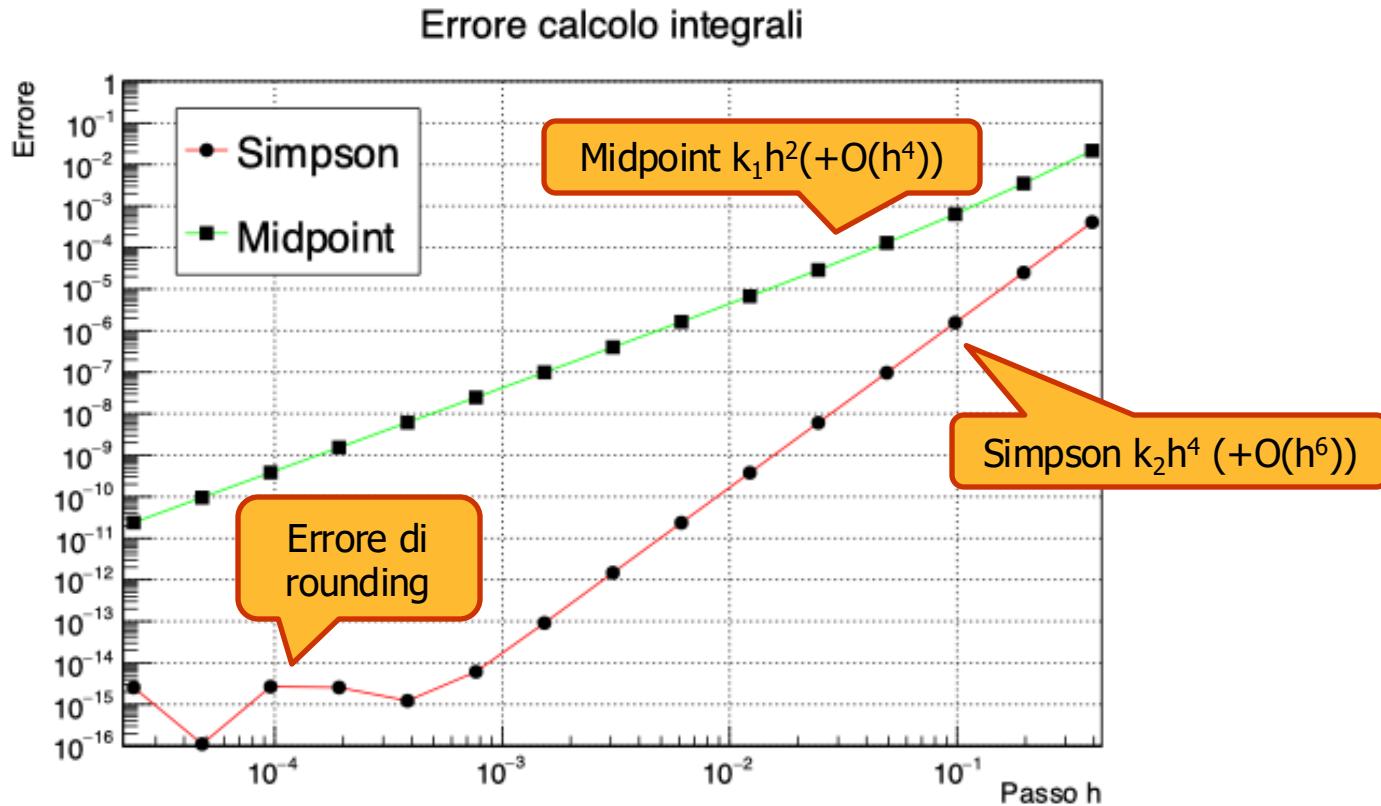
app.Run();

return 0;
```

- scala log-log e griglia
- Tutti I dettagli del primo grafico : l'unico con opzione A (axis) per Draw()
- Disegno l'altro grafico con opzione "same"
- Costruisco una legenda
- Disegno la legenda

Andamento dell'errore in funzione del passo di integrazione

- Prendo un integrale NOTO e calcolo l'errore commesso come differenza tra il valore calcolato e il valore noto in funzione del passo di integrazione utilizzato ($\int_0^{\pi/2} x \sin(x) dx$)



- Questo esercizio è molto utile per calibrare i nostri algoritmi, verifichiamo che siano corretti su un caso di integrale noto

Integrazione numerica

- Integrazione a **numero di passi fissato** : abbiamo scritto algoritmi che accettano in input un numero di intervalli e ci restituiscono un valore dell'integrale
 - Quanto è preciso questo valore ? L'abbiamo studiato su un integrale noto, ma quanto è affidabile su un integrale a priori non noto ?
- Integrazione a **precisione fissata** : in questo caso vogliamo algoritmi che accettino in input una precisione \bar{e} e ci restituiscano un valore dell'integrale che sia entro \bar{e} dal valore esatto dell'integrale
 - Come facciamo però a predire un errore se NON conosciamo il valore esatto dell'integrale che vogliamo calcolare ?

Stima runtime dell'errore

- È possibile avere una stima dell'errore che stiamo commettendo integrando con un metodo di quadratura di quelli indicati ?

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

Questo coefficiente vale per Midpoint e Trapezoidi (il metodo vale anche per Simpson ma il coefficiente sarà diverso)

$$I_h - I_{h/2} = \frac{3}{4} k_1 h^2 + o(h^4) \Rightarrow e(h) = k_1 h^2 = \frac{4}{3} (I_h - I_{h/2})$$

- È possibile stimare quale passo \bar{h} è necessario per ottenere una stima dell'integrale con una incertezza \bar{e} impostata dall'utente ?

$$e(h) = k_1 h^2 = \frac{4}{3} (I_h - I_{h/2}) \Rightarrow k_1 = \frac{4}{3} \frac{(I_h - I_{h/2})}{h^2}$$

$$\bar{e} = k_1 \bar{h}^2 \Rightarrow \bar{h} = \sqrt{\frac{\bar{e}}{k_1}} = \sqrt{\frac{3}{4} \frac{\bar{e} h^2}{(I_h - I_{h/2})}}$$

Algoritmi a precisione fissata : tre opzioni

1. Algoritmo che lavori a precisione fissata (raddoppia il numero di punti ad ogni ciclo finchè l'errore runtime è inferiore alla precisione richiesta)

- Si può utilizzare il corrispondente metodo a numero di passi fissato dentro un do { ... }while opportuno

```
double Integra( double precision , FunzioneBase& f ){

    unsigned int n = 2 ;
    double In = DBL_MAX;
    double I2n = Integra( n ,f);

    do {
        In = I2n ;
        n = n * 2 ;
        I2n = Integra( n ,f);

    } while ( ( 4./3. ) * fabs( I2n - In ) > precision );

    return I2n;
}
```

caveat : frammenti di codice non provati !

Ripeto il calcolo finchè la condizione su precision non viene raggiunta

Restituisco il più preciso tra i due che ho calcolato

Algoritmo a precisione fissata : tre opzioni

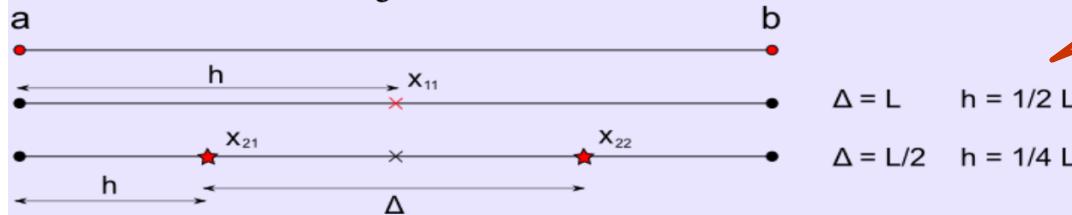
2. Il metodo dei trapezoidi si presta ad una versione ottimizzata per il calcolo di I_n e I_{2n} : ad ogni passaggio aggiungiamo punti al risultato del passo precedente

```
sum0 = (f[a] + f[b]) / 2; I0 = sum0 * (b-a)
```

Al primo passaggio dell'algoritmo suddividiamo l'intervallo in due:

```
sum1 = sum0 + f[x11]; I1 = sum1 * (b-a)/2
```

e così secondo lo schema in figura



```
sum2 = sum1 + f[x21] + f[x22]; I2 = sum2 * (b-a)/4
```

```
sum3 = sum2 + f[x31] + f[x32] + f[x33] + f[x34]; I3 = sum3 * (b-a)/8
```

Vedi lezione7 dal sito
di labTNDS

3. Oppure in generale dalle formule precedenti stimare il passo h necessario per avere un certo errore e fare il calcolo con il passo stimato

- Due valutazioni dell'integrale con passo n e $2n$ (n arbitrario ma di valore ragionevole)
- Stimare la costante k_1 e determinare il passo \bar{h} necessario per una precisione \bar{e}
- Calcolare l'integrale con quel passo e restituirlo

Nota sulla struttura del codice :

- Se vogliamo implementare sia l'integrazione a numero di passi fissato sia l'integrazione a precisione fissata possiamo usare il solito schema

```
class Integral {  
  
public:  
  
    virtual double Integra(unsigned int nstep, FunzioneBase &) = 0 ;  
  
    virtual double Integra(double prec, FunzioneBase &) = 0 ;  
};  
  
class Trapezoidi : public Integral {  
  
public :  
  
    double Integra (unsigned int nstep, FunzioneBase& f) override { ... } ;  
  
    double Integra (double prec , FunzioneBase& f) override { ... } ;  
};
```

□ **Devono** essere implementati **entrambi** in tutte le classi figlie !

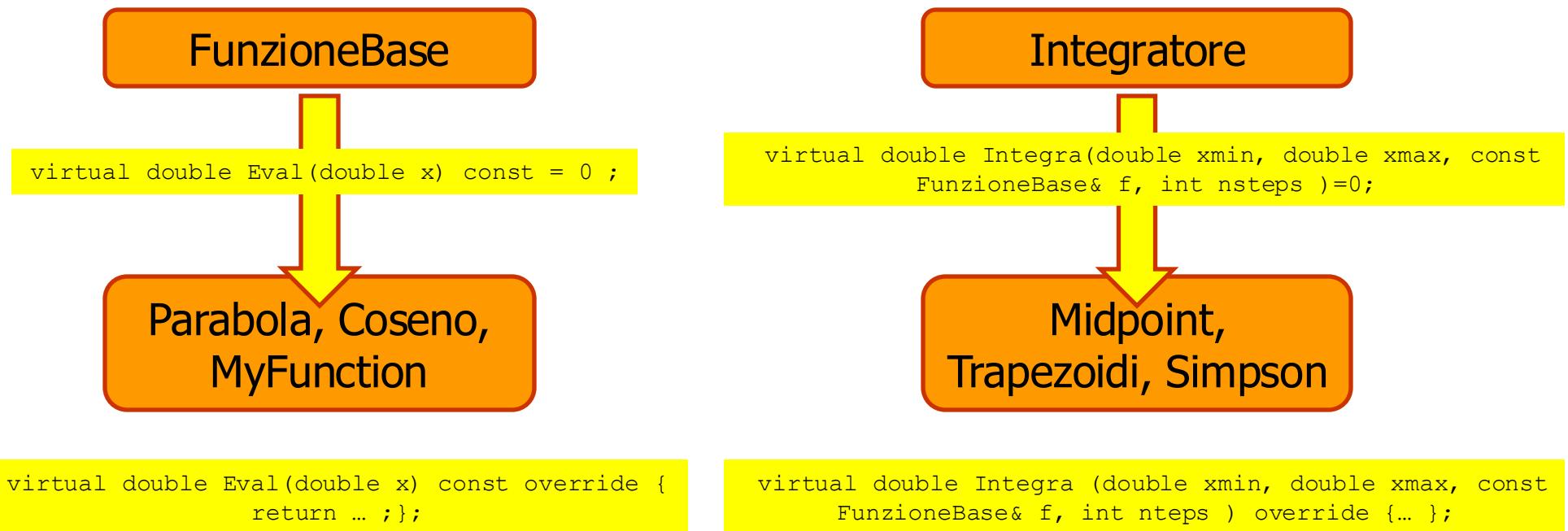
Nota sulla struttura del codice :

- Se vogliamo implementare sia l'integrazione a numero di passi fissato sia l'integrazione a precisione fissata possiamo usare il solito schema

```
class Integral {  
public:  
    virtual double Integra(unsigned int nstep, FunzioneBase &) = 0 ;  
};  
  
class Trapezoidi : public Integral {  
public :  
    double Integra (unsigned int nstep, FunzioneBase& f) override { ... } ;  
    double Integra (double prec , FunzioneBase& f) { ... } ;  
};
```

- Alternativamente il metodo a precisione fissata può essere implementato solo nella classe Trapezoidi (non virtuale) e non nella classe base Integral

Possibile struttura delle classi : caso più generalizzabile



Possibile struttura delle classi : caso più generalizzabile

```
#ifndef __INTEGRAL_H__
#define __INTEGRAL_H__

#include "Funzioni.h"
#include <iostream>

using namespace std;

// Base class : generic integrator
class Integral {
public:
    virtual double Integra(unsigned int nstep, FunzioneBase &) = 0;
protected:
    unsigned int m_nstep;
    double m_a, m_b, m_sum, m_integral, m_h;
    int m_sign;
};

// derived class : Midpoint integrator
class Midpoint : public Integral {
public:
    Midpoint (double a, double b) : Integral(a,b) { ; } ;
    double Integra(unsigned int nstep, FunzioneBase &f) override { };
};

// derived class : Simpson integrator
class Simpson : public Integral {
public:
    Simpson ( double a, double b ) : Integral(a,b) {;};
    double Integra (unsigned int nstep, FunzioneBase &f) override{ };
};
#endif // __INTEGRAL_H__
```

Classe base
astratta

Classe derivata
Midpoint

Classe derivata
Simpson

Possibile struttura delle classi : perchè così complicata ?

```
#include <iostream>
#include <cmath>
#include <iomanip>

#include "Funzioni.h"
#include "Integrale.h"

using namespace std;

int main( int argc , const char ** argv) {

    if ( argc != 2 ) {
        cout << "Usage : ./calcolatore <nstep>" << endl;
        return -1;
    }

    double a = 0;
    double b = M_PI ;

    int nstep = atoi(argv[1]);
    Seno mysin ;
    Trapezoidi integrator_t (a,b,mysin);
    cout << integrator_t.Integra(nstep) << endl;

    Simpson integrator_s (a,b,mysin);
    cout << setprecision(12) << integrator_s->Integra(nstep) << endl;

    // some code here ...

    MyComplexCalculator calc;
    double myoutput = calc.ComplicatedFunction(double k1,
                                                double k2,
                                                const Integratore & inte,
                                                const FunzioneBase & f) {...};

    // some more code here .....

    return 0;
}
```

funzione

Integratori

Quale vantaggio a scrivere Simpson o Trapezoidi come derivati di Integral ?

- Immaginate una funzione molto complicata (ComplicatedFunction) che, tra le altre cose, debba svolgere degli integrali
- Codifichiamo ComplicatedFunction in termini di un generico integratore inte
- Attraverso il polimorfismo possiamo dinamicamente decidere quale integratore (Midpoint, Simpson, Trapezoidi) utilizzare

```
double MyComplexCalculator::ComplicatedFunction(double k1,
                                                double k2,
                                                const Integratore & inte,
                                                const FunzioneBase & f) {...};
```

Questo codice lavora con un integratore generico come input, l'integratore effettivo dipende dal puntatore che viene passato qui!

Implementazione alternativa : puntatori a funzione, funtori e lambda

```
// this is a function
double mysinfunction ( double x ) { return sin(x); } ;

// this is a functor : use it as a function when you need extra parameters. It works like a
// function as long as you specify the operator()

class mysinfunctor {

public :

    mysinfunctor() { m_para = 1 ;}
    mysinfunctor( double para ) { m_para = para ;}
    double operator() ( double x ) {return m_para * sin(x) ;}

private :
    double m_para;

} funct ;

// this is a lambda function
auto mylambda = [] (double x) -> double { return sin(x); };
```

Semplice funzione che verrà passata come puntatore, complicato passare dei parametri esterni però

Un funtore è essenzialmente una classe della quale vogliamo usare il metodo `operator()` per simulare una funzione. Eventuali parametri possono essere passata nei costruttori o con dei metodi del tipo `SetParams`

Una funzione lambda può essere vista come una funzione anonima utile per procedure ‘corte’ (compatto e pulito)

Implementazione alternativa : puntatori a funzione, funtori e lambda

```
// this is simple, but works only with function pointers

double TrapSimple ( double a, double b, int nsteps, double (*f) (double) ) {

    double integral = 0.5 * ( f(a) + f(b) ) ;
    double delta = (b-a) / nsteps ;

    for ( int i = 1 ; i < nsteps ; i++ ) integral+= f( a+i*delta ) ;

    return integral * delta ;
};

// this is a template function, works with function pointers, functors and lambdas

template<typename Func> double TrapTemplate ( double a, double b, int nsteps, Func f ) {

    double integral = 0.5 * ( f(a) + f(b) ) ;
    double delta = (b-a) / nsteps ;

    for ( int i = 1 ; i < nsteps ; i++ ) integral+= f( a+i*delta ) ;

    return integral * delta ;
};

// this uses std::function : works with function pointers, functors and lambdas

double TrapFunction ( double a, double b, int nsteps, std::function<double (double)> f ){

    double integral = 0.5 * ( f(a) + f(b) ) ;
    double delta = (b-a) / nsteps ;

    for ( int i = 1 ; i < nsteps ; i++ ) integral+= f( a+i*delta ) ;

    return integral * delta ;
}
```

Il codice è identico nei tre casi, solo le interface sono diverse

Implementazione alternativa : puntatori a funzione, funtori e lambda

```

int main() {

    // MidpointSimple only works with function pointers

    std::cout << TrapSimple(0,M_PI,10,mysinfunction) << std::endl ;

    // also with lambdas if there's no capture

    auto lambda = [](double x)->double { return sin(x); };

    std::cout << TrapSimple(0,M_PI,10, lambda ) << endl;
    std::cout << TrapSimple(0,M_PI,10, [](double x)->double { return sin(x); } ) << endl;
    std::cout << " " << TrapSimple(0,M_PI,10, sin ) << endl;

    // The other two methods work with function pointers, functors and lambdas

    mysinfunctor m1 ;

    std::cout << TrapTemplate(0,M_PI,10, mysinfunction ) << endl;
    std::cout << TrapTemplate(0,M_PI,10, m1 ) << endl;
    std::cout << TrapTemplate(0,M_PI,10, [&](double x)->double { return sin(x); } ) << endl;

    std::cout << TrapFunction(0,M_PI,10, mysinfunction ) << endl;
    std::cout << TrapFunction(0,M_PI,10, mysinfunctor() ) << endl;
    std::cout << TrapFunction(0,M_PI,10, [](double x)->double { return sin(x); } ) << endl;

    double para = 2;
    mysinfunctor m2(para) ;

    std::cout << TrapTemplate(0,M_PI,10, m2 ) << endl;
    std::cout << TrapTemplate(0,M_PI,10, [&](double x)->double { return para*sin(x); } ) << endl;

}

```

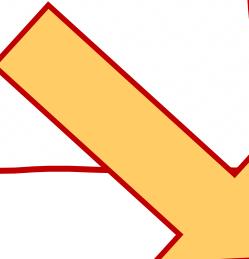
Una nota sulla divisione tra interi

```
#include <iostream>
using namespace std;

int main() {

    if ( 0.5 != 1/2 ){
        cout << "I've seen things you people" << endl;
        cout << "wouldn't believe ..." << endl;
    }
    return 0;
}
```

Immagine sfondo gruppo
WhatsApp docenti labTNDS



```
[Leonardos-MBP-3:Lezione8 lcarmina$ ./integers
I've seen things you people
wouldn't believe ...
```

Algoritmi di integrazione di equazioni differenziali

Laboratorio Trattamento Numerico dei Dati Sperimentali

Prof. L. Carminati
Università degli Studi di Milano

Equazioni differenziali : qualche definizione iniziale

□ DISCLAIMER: elenchiamo qui solo i punti principali della teoria che ci permettono di capire il ragionamento di fondo. Per una trattazione più completa e rigorosa si devono considerare i corsi di analisi matematica !

Definizione

Un'equazione in cui l'incognita è una funzione è detta equazione funzionale. Un'equazione funzionale in cui compare almeno una derivata della funzione incognita è detta equazione differenziale.

Definizione

Un'equazione differenziale si dice ordinaria (ODE) se la funzione incognita dipende da una sola variabile, si dice a derivate parziali (PDE) se dipende da più variabili.

Definizione

La generica equazione differenziale ordinaria di ordine n ha la seguente forma

$$F(x, y(x), \dots, y^n(x)) = 0$$

dove $y: I \subset \mathbb{R} \rightarrow \mathbb{R}$ è derivabile n volte in I e $F: A \subset \mathbb{R}^{n+2} \rightarrow \mathbb{R}$

Equazioni differenziali : qualche definizione iniziale

Definizione

Il massimo ordine di derivazione della funzione incognita è detto ordine dell'equazione differenziale.

Definizione

L'equazione differenziale è detta in forma normale se si può esplicitare la derivata di ordine massimo:

$$y^n(x) = f(x, y(x), \dots, y^{n-1}(x)) \\ \text{con } f : A \subset \mathbb{R}^{n+1} \rightarrow \mathbb{R}$$

Definizione

Si dice soluzione dell'equazione differenziale una funzione $y(x)$ definita su un intervallo $I \subset \mathbb{R}$, derivabile n volte in I e tale che

$$y^n(x) = f(x, y(x), \dots, y^{n-1}(x)), \quad \forall x \in I$$

Equazioni differenziali : qualche definizione iniziale

- In generale per determinare univocamente una soluzione sarà necessario impostare delle condizioni. Possibili condizioni per una equazione differenziale ordinaria di ordine n sono le condizioni iniziali

Definizione

Problema di Cauchy o problema ai valori iniziali:

$$\begin{cases} y(x_0) = \bar{y}_0 \\ y'(x_0) = \bar{y}_1 \\ \dots \\ y^{n-1}(x_0) = \bar{y}_{n-1} \\ y^n(x) = f(x, y(x), y'(x), \dots, y^{n-1}(x)) \end{cases}$$

Definizione

Si dice soluzione del problema di Cauchy una soluzione definita in un intervallo I avente x_0 come punto interno e che soddisfi le condizioni iniziali.

Equazioni differenziali : esistenza e unicità della soluzione locale

□ Esistenza e unicità della soluzione "locale"

- $f: A \subset \mathbb{R}^{n+1} \rightarrow \mathbb{R}$
- f è continua in A
- f localmente Lipschitziana rispetto a y , uniformemente rispetto a x
 $(\forall (x_0, \vec{y_0}) \in A, \exists \text{ un intorno } K_{(x_0, \vec{y_0})} \subset A \text{ in cui } \exists L > 0 \text{ tc } \forall (x, \vec{y}), (x, \vec{z}) \in K \text{ risulta } \|f(x, \vec{y}) - f(x, \vec{z})\| < L \|\vec{y} - \vec{z}\|)$

allora

$\forall (x_0, \vec{y_0}) \in A, \exists \delta > 0$ per cui esiste una sola funzione
 $y: [x_0 - \delta, x_0 + \delta] \rightarrow \mathbb{R}$ soluzione del problema di Cauchy

Equazioni differenziali : prolungamento della soluzione

□ Esistenza e unicità della soluzione “globale”

- $f: A = [(\alpha, \beta) \times \mathbb{R}^n] \subset \mathbb{R}^{n+1} \rightarrow \mathbb{R}$ con $A(\alpha, \beta)$ aperto in \mathbb{R}
- f è continua in $(\alpha, \beta) \times \mathbb{R}^n$
- localmente Lipschitziana rispetto a y , uniformemente rispetto a x
 $(\forall (x_0, \vec{y}_0) \in A, \exists$ un intorno $K_{(x_0, \vec{y}_0)} \subset A$ in cui $\exists L > 0$ tc $\forall (x, \vec{y}), (x, \vec{z}) \in K$ risulta $\|f(x, \vec{y}) - f(x, \vec{z})\| < L \|\vec{y} - \vec{z}\|$)
- sublineare : $\|f(x, \vec{y})\| \leq A(x) + B(x)\|\vec{y}\|$ con $A(x)$ e $B(x)$ continue
allora

$\forall x_0 \in (\alpha, \beta), \forall \vec{y}_0 \in \mathbb{R}^n$ esiste un'unica soluzione globale del problema di Cauchy

- In altre parole: sotto opportune ipotesi sulla funzione f (continuità , lipschitzianità e sublinearità) possiamo garantire esistenza, unicità e estendibilità della soluzione
- nelle nostre applicazioni utilizzeremo funzioni f ‘buone’ in modo da non incorrere in particolari problemi di esistenza o unicità.

Equazioni differenziali : qualche definizione iniziale

- In generale una equazione differenziale di ordine n si può riscrivere come un sistema di equazioni differenziali del primo ordine

Data l'equazione $y^n = f(x, y, \dots, y^{n-1})$ introduciamo le funzioni ausiliarie

$$y_0(x) = y(x), y_1(x) = y'(x), y''(x) = y_2(x) \dots y^{n-1}(x) = y_{n-1}$$

Il problema di Cauchy può essere riscritto in questo modo

$$\begin{cases} y'_0(x) = y_1(x) \\ y'_1(x) = y_2(x) \\ \dots \\ y'_{n-1}(x) = f(x, y_0, y_1(x), \dots, y_{n-1}(x)) \end{cases} \quad \text{con} \quad \begin{cases} y_0(x_0) = \bar{y}_0 \\ y_1(x_0) = \bar{y}_1 \\ \dots \\ y_{n-1}(x_0) = \bar{y}_{n-1} \end{cases}$$

$$\text{Con } f: A \subset \mathbb{R}^{n+1} \rightarrow \mathbb{R}$$

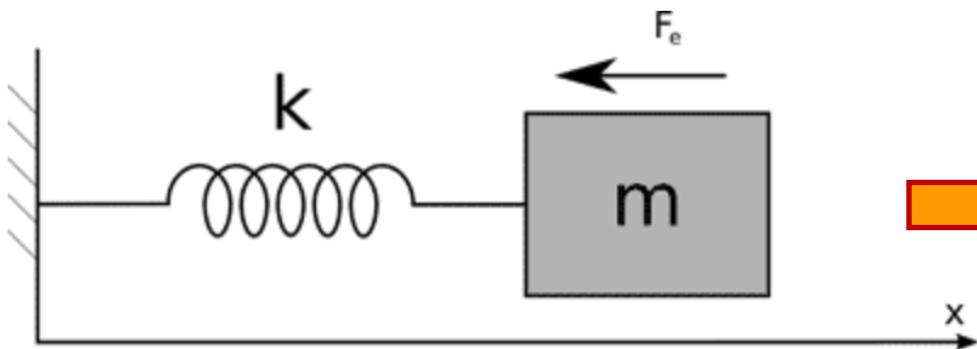
- Ci possiamo pertanto concentrare sulle tecniche di integrazione di equazioni differenziali di ordine 1

Equazioni differenziali

- Perché siamo interessati a studiare problemi di questo tipo ? Cambiamo notazione : se sostituiamo y con x e x con t abbiamo magicamente le equazioni della dinamica !

$$F = ma = m \frac{d^2}{dt^2} x \Rightarrow \frac{d^2}{dt^2} x = \frac{F}{m}$$

Equazione
differenziale
di ordine 2



$$\frac{d^2}{dt^2} x = \frac{F}{m} = \frac{k}{m} x$$

Due eqdiff di
ordine 1

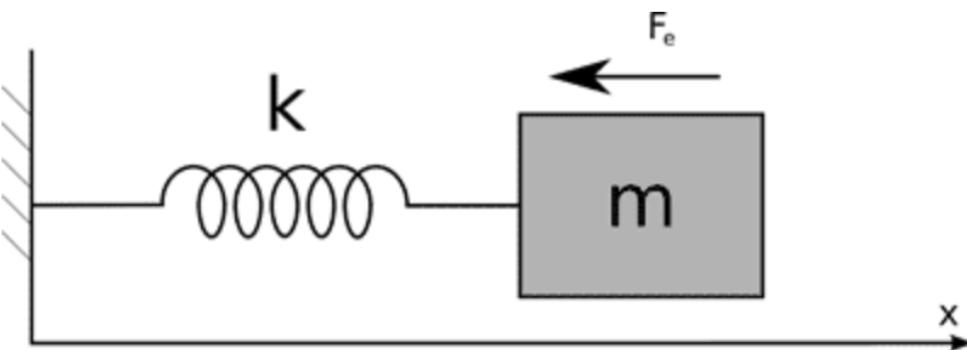
$$\begin{cases} \frac{dx}{dt} = v \\ \frac{dv}{dt} = -\frac{k}{m} x \end{cases} \quad \begin{cases} x(t_0) = \bar{x}_0 \\ v(t_0) = \bar{v}_0 \end{cases} \quad \Rightarrow \quad x(t) = A \cdot \cos\left(\sqrt{\frac{k}{m}} t + \varphi\right)$$

Oscillatore armonico semplice

Equazioni differenziali

- Perchè siamo interessati a studiare problemi di questo tipo ? Se sostituiamo y con x e x con t abbiamo magicamente le equazioni della dinamica !!

$$F = ma = m \frac{d^2}{dt^2} x \Rightarrow \frac{d^2}{dt^2} x = \frac{F}{m}$$



$$\frac{d^2}{dt^2} x = \frac{F}{m} = \frac{k}{m} x - \alpha \frac{dx}{dt} + \sin(\omega t)$$

Posso aggiungere un termine di smorzamento e/o una forzante

Oscillatore armonico smorzato con forzante

$$\begin{cases} \frac{dx}{dt} = v \\ \frac{dv}{dt} = -\omega_0^2 x - \alpha v + \sin(\omega t) \end{cases}$$

$$\begin{cases} x(t_0) = \bar{x}_0 \\ v(t_0) = \bar{v}_0 \end{cases}$$

Equazioni differenziali : approccio numerico

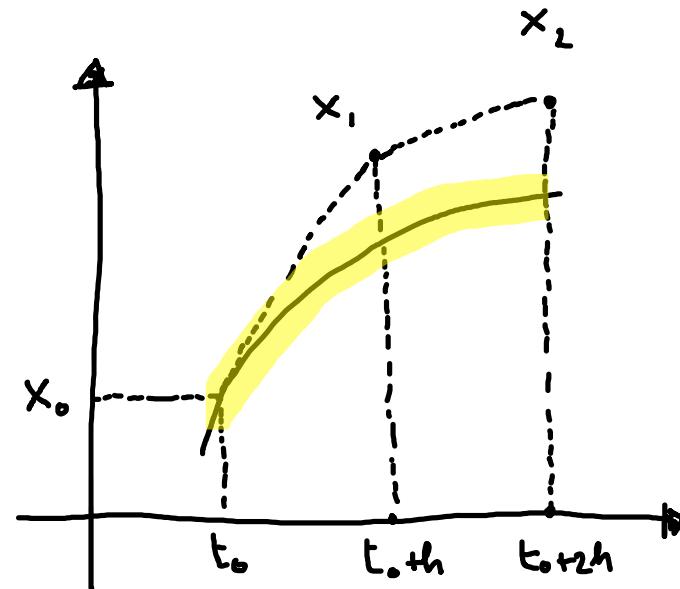
Algoritmi che affrontino il problema dal punto di vista numerico: procediamo per punti !

- Dobbiamo trovare una tecnica di approssimazione della soluzione implementabile algoritmicamente: partiamo da una approssimazione di Taylor della funzione

$$x_1 = x(t_0 + h) = x(t_0) + x'(t_0)h + \frac{1}{2}x''(t_0)h^2 + \dots$$

- Risolviamo iterativamente il problema applicando ogni volta l'approssimazione in un intervallo 'piccolo' : troviamo un set di punti consecutivi $p_i=(t_i,x_i)$ che approssimano la soluzione cercata.

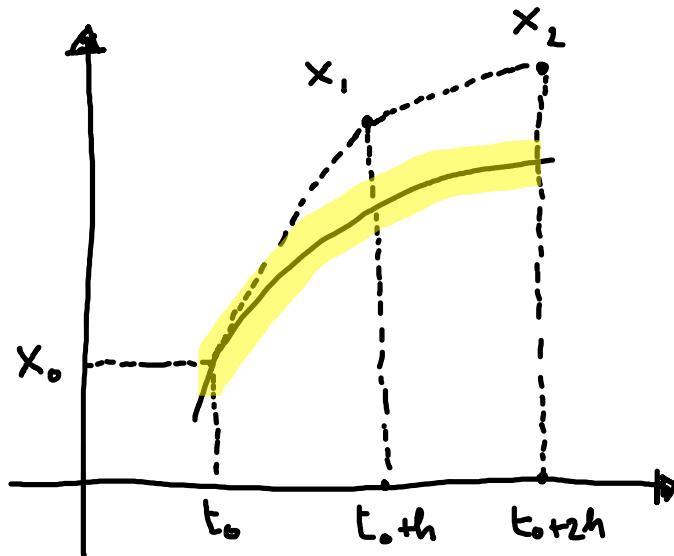
- La domanda a cui rispondiamo è : se il sistema per $t=t_0$ si trova nella posizione y_0 dove si troverà all'istante generico t ? A partire da t_0 raggiungiamo l'istante t in un numero N di passi di lunghezza $\Delta t=h$
- La soluzione numerica dell'equazione differenziale è costituita da un set di punti che approssimano la soluzione esatta



Metodo di Eulero

- Facciamo la cosa più semplice che ci viene in mente : tronchiamo lo sviluppo di Taylor al primo ordine. La derivata prima ce la fornisce il problema !

$$x_1 = x(t_0 + h) = x(t_0) + x'(t_0)h + \frac{1}{2}x''(t_0)h^2 + \dots$$



$$\begin{cases} x'(t) = f(t, x(t)) \\ x(t_0) = x_0 \end{cases}$$

$$x_1 = x(t_0) + h x'(t_0) + o(h)$$

$$x_{n+1} = x(t_n) + h x'(t_n) + o(h)$$

- L'algoritmo è molto semplice : a partire dal punto noto della soluzione (t_0, x_0) approssiamo la soluzione con la retta tangente al punto ed estrapoliamo fino a t_0+h . Ripartiamo con lo stesso procedimento a partire dal nuovo punto (t_0+h, x_1)

Implementazione : caso di sistema fisico

- Il caso di un tipico: sistema fisico nel caso monodimensionale (pensiamo al solito oscillatore armonico) è rappresentato da una equazione di ordine 2 \Rightarrow due equazioni del primo ordine

$$\begin{cases} \frac{dx}{dt} = v \\ \frac{dv_x}{dt} = -\frac{k}{m}x \end{cases} \quad \begin{cases} x(t_0) = \bar{x}_0 \\ v(t_0) = \bar{v}_0 \end{cases}$$

- Dobbiamo quindi pensare di costruire un codice che implementi il semplice algoritmo (Eulero) della slide precedente per le due componenti (posizione x e velocità v_x):
- Ma se il problema fosse più complicato (esempio : moto di una particella carica in una regione di spazio con presenza di un campo elettrico e/o magnetico) ? Come scrivere codice che sia indipendente dal tipo e dalla dimensione del problema ?

Implementazione : caso di sistema fisico

- Il moto di una particella carica in una regione di spazio con presenza di un campo elettrico e magnetico è rappresentato da 6 equazioni di ordine 1, 3 per la posizione e 3 per la velocità

$$\frac{d}{dt} \begin{pmatrix} x \\ y \\ z \\ v_x \\ v_y \\ v_z \end{pmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & \frac{q}{m} B_z & -\frac{q}{m} B_y \\ 0 & 0 & 0 & -\frac{q}{m} B_z & 0 & \frac{q}{m} B_x \\ 0 & 0 & 0 & \frac{q}{m} B_y & -\frac{q}{m} B_x & 0 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ v_x \\ v_y \\ v_z \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ \frac{q}{m} E_x \\ \frac{q}{m} E_y \\ \frac{q}{m} E_z \end{pmatrix}$$

- Meglio pensare al problema in termini vettoriali !

Implementazione : caso di sistema fisico

□ Proviamo ad esplicitare il metodo di Eulero in tutte le sue componenti:

Usiamo dei vector per le colonne

$$\begin{pmatrix} x(t+h) \\ y(t+h) \\ z(t+h) \\ v_x(t+h) \\ v_y(t+h) \\ v_z(t+h) \end{pmatrix} = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \\ v_x(t) \\ v_y(t) \\ v_z(t) \end{pmatrix} + h \begin{pmatrix} \frac{d}{dt}x(t) \\ \frac{d}{dt}y(t) \\ \frac{d}{dt}z(t) \\ \frac{d}{dt}v_x(t) \\ \frac{d}{dt}v_y(t) \\ \frac{d}{dt}v_z(t) \end{pmatrix}$$

Qui dobbiamo introdurre una funzione che accetti in input il vettore di stato (x, y, z, v_x, v_y, v_z) e un istante di tempo t e ci restituisca il vettore delle derivate a quell'istante di tempo (una funzione vettoriale con un metodo Eval opportuno)

Metodo passo : ci permette di fare evolvere il vettore di stato del sistema dall'istante t a quello $t+h$

In notazione vettoriale sarebbe
 $\overrightarrow{x(t+h)} = \overrightarrow{x(t)} + h\overrightarrow{F(t, \overrightarrow{x(t)})}$
Diamo un'algebra ai vector !

Algebra dei vettori : facciamo una prova

- Sarebbe molto utile poter implementare le operazioni di somma tra `vector`, prodotto scalare tra `vector`, prodotto tra un `vector` e uno scalare
- Sappiamo implementare `operator+` per esempio nelle classi, ma come fare con una classe non nostra, il `vector` ?
- Possiamo implementarli come funzioni libere come nella slide successiva



```
#include <iostream>
#include "VectorOperations.h"

using namespace std;

int main() {

    vector<double> v1 {2,3,4} ;
    vector<double> v2 {22,33,44};

    cout << "This is v1 " << endl;
    Print (v1) ;
    cout << "This is v2 " << endl;
    Print (v2) ;

    double k = 4.;
    vector<double> v3 = k*v1;
    cout << "====>> Multiply v1 by a value k = " << k << endl;
    Print ( v3 ) ;

    cout << "====>> Summing v1 and v2 (operator+)" << endl;
    vector<double> v4 = v1+v2 ;
    cout << "This is v1 " << endl;
    Print (v1) ;
    cout << "This is v2 " << endl;
    Print (v2) ;
    cout << "This is v4 " << endl;
    Print (v4) ;

    cout << "====>> Summing v1 and v2 (operator+=)" << endl;
    vector<double> v5 = v1+=v2 ;
    cout << "This is v1 (modified !)" << endl;
    Print (v1) ;
    cout << "This is v2 " << endl;
    Print (v2) ;
    cout << "This is v5 " << endl;
    Print (v5) ;
}
```

Algebra dei vettori : definiamo somma e differenza

VectorOperations.h

```
// =====
// somma di due vettori : somma componente per componente
// =====

template <typename T> inline std::vector<T> operator+(const std::vector<T> &a, const std::vector<T> &b) {

    // if ( a.size() != b.size() ) throw "Trying to sum vectors with different size" ;

    if ( a.size() != b.size() ) {
        cout << "Trying to sum vectors with different size, exiting" << endl ;
        exit(-1);
    }

    std::vector<T> result(a.size());

    for (int i = 0; i < static_cast<int>(a.size()); i++) result[i] = a[i] + b[i];

    // Alternativamente si puo' usare l'algoritmo transform della STL
    //
    // std::transform(a.begin(), a.end(), b.begin(), result.begin() , std::plus<T>());

    return result;
}
```

Different ways to check that the two vectors have the same size (see next slide for "throw")

Pesca gli elementi di due vector ed esegue l'operazione specificata
std::transform(
"punto di partenza nel primo vettore",
"punto di termine nel primo vettore",
"punto di partenza nel secondo vettore",
"dove parto per inserire i risultati,
"quale operazione con elementi vettori di input)

<http://www.cplusplus.com/reference/algorithm/transform/>

Intermezzo : exceptions, come passare una eccezione al main

```

#include <iostream>

// =====
// somma componente per componente
// =====

template <typename T> inline std::vector<T> operator+(const std::vector<T> &a, const std::vector<T> &b) {
    if ( a.size() != b.size() ) throw "Trying to sum vectors with different size" ;
    std::vector<T> result(a.size());
    std::transform(a.begin(), a.end(), b.begin(), result.begin(), std::plus<T>());
    return result;
}

using namespace std;

int main() {
    vector<double> v1 {2,3,4,5} ;
    vector<double> v2 {5,6,4} ;

    vector<double> v3 ;

    try{
        v3 = v1+v2 ;
    } catch ( const char* msg ) {
        cout << msg << endl;
        cout << "Exiting" << endl;
        exit(-1);
    }

    Print(v3);
}

```

An exception of type `char*` is thrown in case vectors have different size

The exception is caught in the main : you can decide what to do here instead of inside the function

Algebra dei vettori : prodotto scalare tra due vettori

VectorOperations.h

```
// =====
// prodotto scalare tra due vettori
// =====

template <typename T> inline T operator*(const std::vector<T> &a, const std::vector<T> &b) {

    // if ( a.size() != b.size() ) throw "Trying to sum vectors with different size" ;

    if ( a.size() != b.size() ) {
        cout << "Trying to sum vectors with different size, exiting" << endl ;
        exit(-1);
    }

    T sum = 0 ;
    for (int i = 0; i < static_cast<int>(a.size()); i++) sum += a[i] * b[i];

    // Alternativamente si puo' usare l'algoritmo inner_product della STL
    //
    // sum = std::inner_product(std::begin(a), std::end(a), std::begin(b), 1.0);

    return sum;
}
```

Prodotto interno tra gli elementi
std::inner_product(
"punto di partenza nel primo vettore",
"punto di termine nel primo vettore",
"punto di partenza nel secondo vettore",
"valore iniziale della variabile di accumulazione"

http://www.cplusplus.com/reference/numeric/inner_product/

Algebra dei vettori : definiamo prodotto per uno scalare

VectorOperations.h

```
// =====
// prodotto tra uno scalare e un vettore
// =====

template <typename T> inline std::vector<T> operator*( T c , const std::vector<T> &a) {

    std::vector<T> result(a.size());

    for (int i = 0; i < static_cast<int>(a.size()); i++) result[i] = c * a[i];

    // Alternativamente si puo' usare l'algoritmo inner product
    //
    // std::transform(std::begin(a), std::end(a), std::begin(result), [&c](T x){ return x * c; } );

    return result;
}

// =====
// prodotto tra un vettore e uno scalare
// =====

template <typename T> inline std::vector<T> operator*( const std::vector<T> &a , T c) {

    std::vector<T> result(a.size());

    for (int i = 0; i < static_cast<int>(a.size()); i++) result[i] = c * a[i];

    // oppure il ciclo for puo' essere sostituito da ( ~ stesso numero di operazioni con il
    // move constructor del vector altrimenti sarebbe meno efficiente )
    //
    // result = c * a ;

    // Alternativamente si puo' usare l'algoritmo transform della STL con una lambda function
    //
    // std::transform(std::begin(a), std::end(a), std::begin(result), [&c](T x){ return x * c; } );

    return result;
}
```

Devo definire sia $c*a$ sia $a*c$

Possiamo usare transform
passando una lambda function

Algebra dei vettori : facciamo una prova

```
#include <iostream>
#include "VectorOperations.h"

using namespace std;

int main() {

    vector<double> v1 {2,3,4} ;
    vector<double> v2 {22,33,44};

    cout << "This is v1 " << endl;
    Print (v1) ;
    cout << "This is v2 " << endl;
    Print (v2) ;

    double k = 4.;
    vector<double> v3 = k*v1;
    cout << "==>> Multiply v1 by a value k = " << k << endl;
    Print (v3) ;

    cout << "==>> Summing v1 and v2 (operator+)" << endl;
    vector<double> v4 = v1+v2 ;
    cout << "This is v1 " << endl;
    Print (v1) ;
    cout << "This is v2 " << endl;
    Print (v2) ;
    cout << "This is v4 " << endl;
    Print (v4) ;

    cout << "==>> Summing v1 and v2 (operator+=)" << endl;
    vector<double> v5 = v1+=v2 ;
    cout << "This is v1 (modified !)" << endl;
    Print (v1) ;
    cout << "This is v2 " << endl;
    Print (v2) ;
    cout << "This is v5 " << endl;
    Print (v5) ;

}
```

ProvaAlgebra.cxx

```
This is v1
Printing vector
2 3 4
End of printing vector
This is v2
Printing vector
22 33 44
End of printing vector
==>> Multiply v1 by a value k = 4
Printing vector
8 12 16
End of printing vector
==>> Summing v1 and v2 (operator+)
This is v1
Printing vector
2 3 4
End of printing vector
This is v2
Printing vector
22 33 44
End of printing vector
This is v4
Printing vector
24 36 48
End of printing vector
==>> Summing v1 and v2 (operator+=)
This is v1 (modified !)
Printing vector
24 36 48
End of printing vector
This is v2
Printing vector
22 33 44
End of printing vector
This is v5
Printing vector
24 36 48
End of printing vector
```

Implementazione : caso di sistema fisico

□ Proviamo ad esplicitare tutte le componenti della soluzione con

Usiamo dei vector per le colonne

$$\begin{pmatrix} x(t+h) \\ y(t+h) \\ z(t+h) \\ v_x(t+h) \\ v_y(t+h) \\ v_z(t+h) \end{pmatrix} = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \\ v_x(t) \\ v_y(t) \\ v_z(t) \end{pmatrix} + h \begin{pmatrix} \frac{d}{dt}x(t) \\ \frac{d}{dt}y(t) \\ \frac{d}{dt}z(t) \\ \frac{d}{dt}v_x(t) \\ \frac{d}{dt}v_y(t) \\ \frac{d}{dt}v_z(t) \end{pmatrix}$$

Metodo 'passo' : ci permette di fare evolvere il vettore di stato del sistema dall'istante t a quello $t+h$

Qui dobbiamo introdurre una funzione che accetti in input il vettore di stato (x, y, z, v_x, v_y, v_z) e un istante di tempo t e ci restituisca il vettore delle derivate a quell'istante di tempo (una funzione vettoriale con un metodo Eval opportuno)

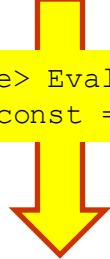
In notazione vettoriale sarebbe
 $\overrightarrow{x(t+h)} = \overrightarrow{x(t)} + h\overrightarrow{F(t, \overrightarrow{x(t)})}$
Diamo un'algebra ai vector !

Possibile struttura delle classi

Sistemi fisici

FunzioneVettorialeBase

```
virtual vector<double> Eval(double t, const  
vector<double> & x) const = 0 ;
```



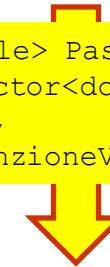
Oscillatore, pendolo, ...

```
vector<double> Eval(double t, const  
vector<double> & x) const override{ ... } ;
```

Algoritmi

EquazioneDifferenzialeBase

```
virtual vector<double> Passo (double t,  
const vector<double> & x,  
double h,  
const FunzioneVettorialeBase & f )=0;
```



Eulero, Runge-Kutta, ..

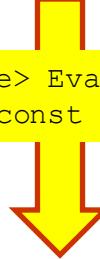
```
virtual vector<double> Passo (double t,  
const vector<double> & x,  
double h,  
const FunzioneVettorialeBase & f )  
override { ... } ;
```

Possibile struttura delle classi

Sistemi fisici

FunzioneVettorialeBase

```
virtual vector<double> Eval(double t, const  
vector<double> & x) const = 0 ;
```



Oscillatore, pendolo, ...

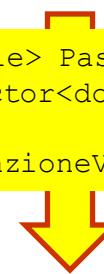
```
vector<double> Eval(double t, const  
vector<double> & x) const override{ ... } ;
```

Perchè una dipendenza esplicita dal tempo ?
Vedi caso dell'oscillatore armonico con
forzante ($\sin(t)$)

Algoritmi

EquazioneDifferenzialeBase

```
virtual vector<double> Passo (double t,  
const vector<double> & x,  
double h,  
const FunzioneVettorialeBase & f )=0;
```



Eulero, Runge-Kutta, ..

```
virtual vector<double> Passo (double t,  
const vector<double> & x,  
double h,  
const FunzioneVettorialeBase & f )  
override { ... } ;
```

Inline functions

- In VectorOperations.h tutte le funzioni sono dichiarate inline

```
#include "functions.h"
#include "myclass.h"

int main() {
    myclass foo ;
    double x ;
    double bar = mysin( x ) ;
}
```

```
myclass.h
```

```
class myclass{
public :
    double method ( double x ) ;
private :
    double num ;
};
```

```
myclass.cpp
```

```
#include "myclass.h"
#include "functions.h"

double myclass::method ( double x ) {
    return mysin ( x ) ;
};
```

```
functions.h
```

```
#include <cmath>

inline double mysin ( double x ) { return sin(x) ; } ;
```

Inline functions

- In `VectorOperations.h` tutte le funzioni sono dichiarate inline

```
#include "functions.h"
#include "myclass.h"

int main() {
    myclass foo ;
    double x ;
    double bar = mysin( x ) ;
}
```

```
myclass.h
class myclass{
public :
    double method ( double x ) ;
private :
    double num ;
};
```

```
myclass.cpp
#include "myclass.h"
#include "functions.h"

double myclass::method ( double x ) {
    return mysin ( x ) ;
};
```

```
main : main.o myclass.o
g++ -o main main.o myclass.o

main.o : main.cpp functions.h
g++ -c -o main.o main.cpp

myclass.o : myclass.cpp myclass.h
g++ -c -o myclass.o myclass.cpp
```

```
functions.h
#include <cmath>

inline double mysin ( double x ) { return sin(x) ; } ;
```

- Se la funzione `mysin` non fosse dichiarata inline il compilatore darebbe un errore “duplicate symbols”

Rappresentiamo il problema fisico:

- Un problema fisico reale è definito dalle sue derivate : codifichiamo un generico problema come una `FunzioneVettorialeBase` (astratta). Ogni sistema concreto erediterà da `FunzioneVettorialeBase`

```
// =====
// classe astratta, restituisce la derivata da valutata nel punto x
// =====

class FunzioneVettorialeBase {

public:
    virtual vector<double> Eval(double t, const vector<double> & x) const = 0;
};

// caso fisico concreto, oscillatore armonico

class OscillatoreArmonico : public FunzioneVettorialeBase {
public:
    OscillatoreArmonico(double omega0) { m_omega0 = omega0; }

    virtual vector<double> Eval(double t, const vector<double> & x) const {

    };

private:
    double m_omega0;
};
```

Metodo virtuale puro : restituisce il vettore delle derivate calcolate rispetto ad un vettore di stato x (ed eventualmente un tempo t)

Dobbiamo implementare qui la derivata: restituire un `vector` le cui componenti rappresentano le derivate all'istante t . Qui codifichiamo il contenuto fisico del problema

Rappresentiamo il problema fisico:

- Un problema fisico reale è definito dalle sue derivate : codifichiamo un generico problema come una `FunzioneVettorialeBase` (astratta). Ogni sistema concreto erediterà da `FunzioneVettorialeBase`

```
// =====  
// classe astratta, restituisce la derivata da valutata nel punto x  
// =====  
  
class FunzioneVettorialeBase {  
public:  
    virtual vector<double> Eval(double t, const vector<double> & x) const {  
        ...  
    }  
};
```

Metodo virtuale puro : restituisce il vettore delle derivate calcolate rispetto ad un vettore di stato x (

```
virtual vector<double> Eval(double t, const vector<double> & x) const {  
  
    vector<double> dxdt { x[1] , -(m_omega0*m_omega0*x[0]) } ;  
  
    return dxdt;  
};
```

```
private:  
    double m_omega0;  
};
```

Dobbiamo implementare qui la derivata: restituire un `vector` le cui componenti rappresentano le derivate all'istante t. Qui codifichiamo il contenuto fisico del problema



$$\frac{d}{dt} \begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} v \\ -\omega_0^2 x \end{pmatrix}$$

Rappresentiamo la parte algoritmica:

- Un algoritmo specifico (e.g. Eulero) eredita da una classe astratta generale

EquazioneDifferenzialeBase :

```
// =====
// classe astratta per un integratore di equazioni differenziali
// =====

class EquazioneDifferenzialeBase {

public:
    virtual vector<double> Passo(double t, const vector<double>& x, double h, const FunzioneVettorialeBase &f) const =0;
};

// integratore concreto, metodo di Eulero

class Eulero : public EquazioneDifferenzialeBase {

public:
    virtual vector<double> Passo(double t, const vector<double> & x, double h, const FunzioneVettorialeBase &f) const override {

        return x+(f.Eval(t,x))*h;
    }
};
```

Sfruttiamo di nuovo il polimorfismo

Dobbiamo codificare qui come funziona il metodo di Eulero : utilizzando l'algebra dei vettori il metodo diventa indipendente dalle dimensioni del problema !

Studiamo l'oscillatore armonico :

```
#include "EquazioniDifferenziali.h"
#include <iostream>
#include <string>

#include "TApplication.h"
#include "TAxis.h"
#include "TCanvas.h"
#include "TGraph.h"

int main (int argc, char** argv ) {

    if ( argc!=2 ) {
        std::cerr << "Usage: " << argv[0] << " <stepsize>" << std::endl;
        return -1;
    }

    TApplication myApp("myApp",0,0);
    Eulero myEuler;
    OscillatoreArmonico osc(1.);

    double tmax = 70.;
    double h = atof(argv[1]);
    vector<double> x {0.,1.};
    double t = 0.;

    TGraph myGraph ;
    int nstep = int(tmax/h+0.5);

    for (int step=0; step<nstep; step++) {
        myGraph.SetPoint(step,t,x[0]);
        x = myEuler.Passo(t,x,h,osc);
        t = t+h;
    }

    TCanvas c ;
    c.cd();
    string title = "Oscillatore armonico (Eulero h = " + std::to_string(h) + ")";
    myGraphSetTitle(title.c_str());
    myGraph.GetXaxis()->SetTitle("Tempo [s]");
    myGraph.GetYaxis()->SetTitle("Posizione x [m]");
    myGraph.Draw("AL");

    myApp.Run();
}
```

Algoritmo di integrazione

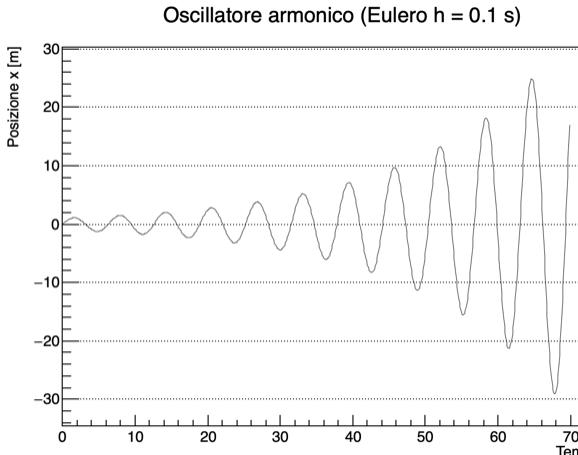
Problema fisico : eredita da
FunzioneVettorialeBase

Ciclo base del codice : la
costruzione della soluzione
avviene per chiamate
successive del metodo Passo

Determino quanti passi fare

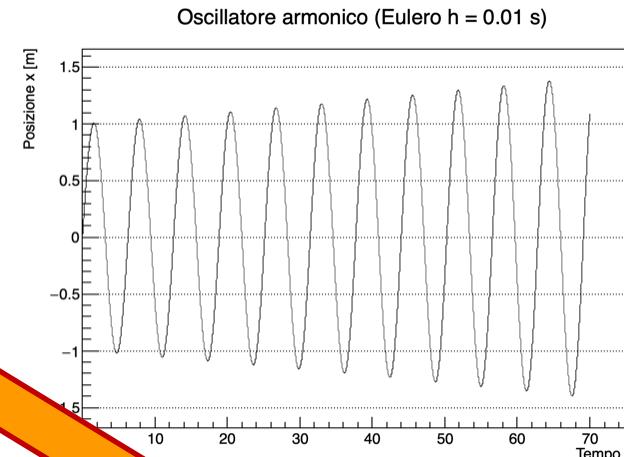
Polimorfismo : entra un
OscillatoreArmonico

Integrazione di equazioni differenziali con il metodo di Eulero



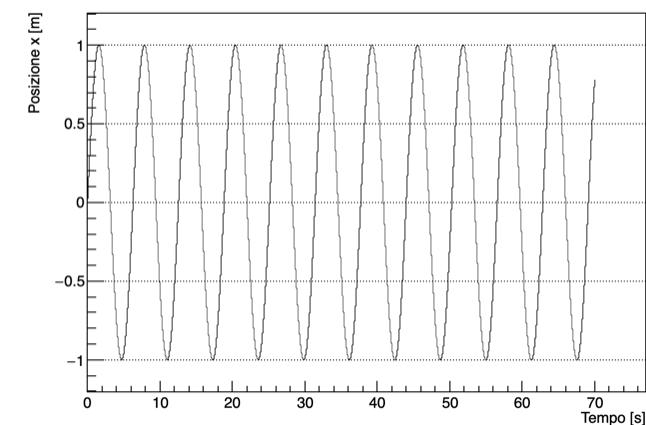
□ Integriamo l'oscillatore armonico da 0 a 70 s con diversi passi

Passo di integrazione
 $h=0.1$: 700 integrazioni



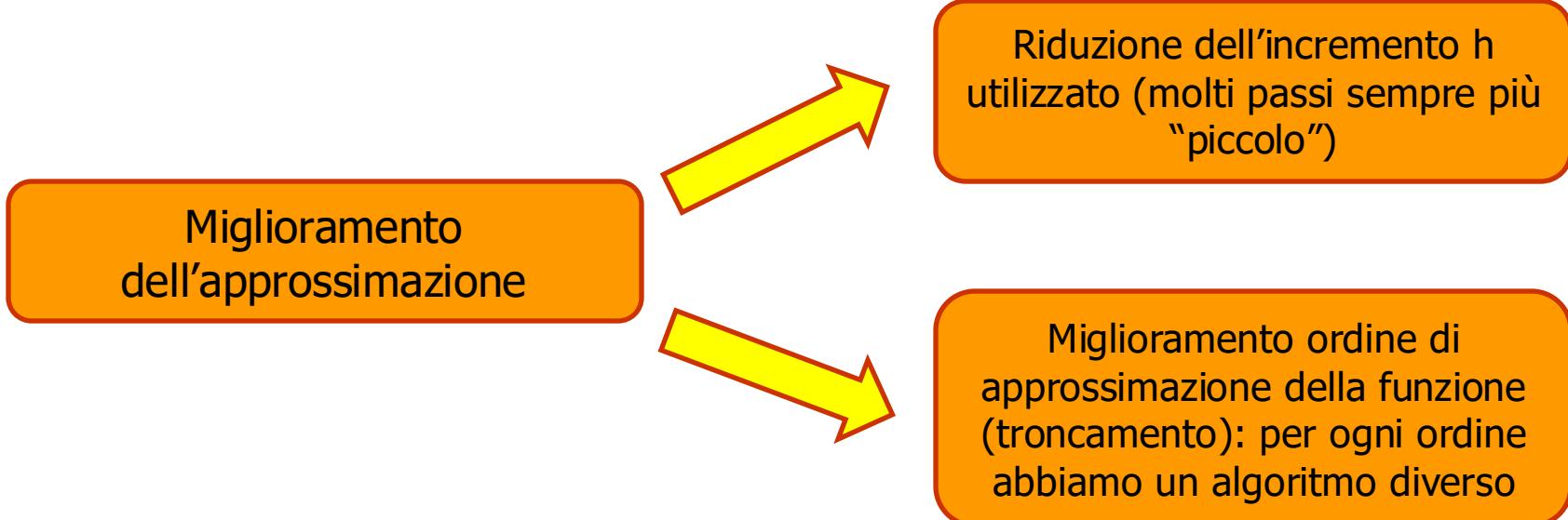
Passo di integrazione
 $h=0.01$: 7000 integrazioni

Passo di integrazione $h=0.0001$: 700000 integrazioni



Integrazione dell'equazione differenziale con passo di integrazione che diminuisce: interessante studiare come va l'errore tra la nostra soluzione e la soluzione esatta al variare del passo h per un tempo fissato (diciamo $t=70$ s)

Equazioni differenziali : approccio numerico



Metodo di Runge-Kutta di ordine 2

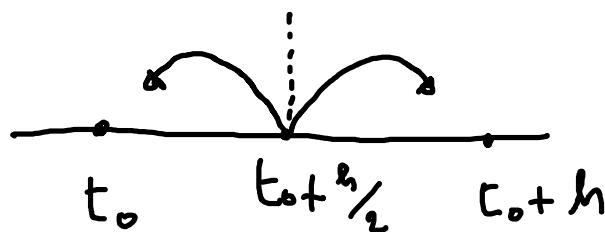
- Possiamo migliorare il metodo di integrazione cercando di migliorare l'ordine di troncamento utilizzato per il calcolo ?

$$\begin{cases} x'(t) = f(t, x(t)) \\ x(t_0) = t_0 \end{cases}$$

↳ $x_1 = x(t_0) + h x'(t_0) + \frac{1}{2} h^2 x''(t_0) + o(h^2)$

La derivate seconda purtroppo non è nota.
Dobbiamo calcolarla o trovare un modo di cancellare questo termine di errore

- Proviamo a questo punto a scrivere lo sviluppo di Taylor della ipotetica soluzione a partire dal punto di mezzo, sviluppando in avanti e all'indietro



Metodo di Runge-Kutta di ordine 2

$$x(t_0 + h) = x\left(t_0 + \frac{h}{2}\right) + \frac{h}{2} x'\left(t_0 + \frac{h}{2}\right) + \frac{h^2}{8} x''\left(t_0 + \frac{h}{2}\right) + o(h^2)$$

$$x(t_0) = x\left(t_0 + \frac{h}{2}\right) - \frac{h}{2} x'\left(t_0 + \frac{h}{2}\right) + \frac{h^2}{8} x''\left(t_0 + \frac{h}{2}\right) + o(h^2)$$



$$x(t_0 + h) - x(t_0) = h x'\left(t_0 + \frac{h}{2}\right) + o(h^2)$$

Questa non è nota
dal problema

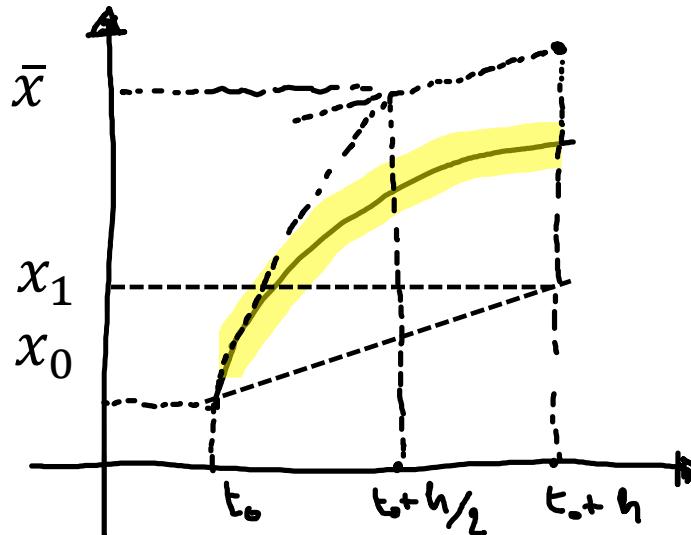
Possiamo applicare Eulero
(errore $\sim h^2$), questo termine
viene poi moltiplicato per h

$$x'\left(t_0 + \frac{h}{2}\right) = f\left(t_0 + \frac{h}{2}, x\left(t_0 + \frac{h}{2}\right)\right) \cong f\left(t_0 + \frac{h}{2}, x_0 + \frac{h}{2} f(t_0, x_0)\right)$$

$$x(t_0 + h) = x(t_0) + h f\left(t_0 + \frac{h}{2}, x_0 + \frac{h}{2} f(t_0, x_0)\right) + o(h^2)$$

Metodo di Runge-Kutta di ordine 2

- Formalizziamo il metodo che abbiamo appena realizzato (metodo di Runge di ordine 2) con errore locale di ordine h^3



$$\boxed{\begin{aligned}K_1 &= f(t_n, x_n) \\K_2 &= f\left(t_n + \frac{h}{2}, x_n + \frac{h}{2} K_1\right) \\x_{n+1} &= x_n + h K_2\end{aligned}}$$

- Gli steps dell'algoritmo sono i seguenti :

- A partire dal punto noto della soluzione (t_0, x_0) approssiamo la soluzione con la retta tangente al punto ed estrapoliamo fino a $t_0 + h/2$.
- Calcoliamo la derivata in $(t_0 + h/2, \bar{x})$
- A partire da (t_0, x_0) approssiamo la soluzione fino a $t_0 + h$ con la retta con coefficiente angolare calcolato nel punto 2.

Metodo di Runge-Kutta di ordine 4

"The numerical analysis of ordinary differential equations", J.C. Butcher

- Utilizzando sviluppi opportuni della funzione in punti vicini è possibile costruire un algoritmo di integrazione di equazioni differenziali con errore locale di ordine h^5

$$K_1 = f(t_n, x_n)$$

$$K_2 = f\left(t_n + \frac{h}{2}, x_n + \frac{h}{2} K_1\right)$$

$$K_3 = f\left(t_n + \frac{h}{2}, x_n + \frac{h}{2} K_2\right)$$

$$K_4 = f(t_n + h, x_n + hK_3)$$

$$x_{n+1} = x_n + \frac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4) + o(h^4)$$

Esercizio 8.2 : studio dell'errore con il metodo di Runge-Kutta di ordine 4

```
#include "EquazioniDifferenziali.h"
// ...
#include "TApplication.h"
#include "TGraph.h"
// ...
int main(int argc, char** argv) {
    TApplication myApp("myApp",0,0);
    RK myRK;
    OscillatoreArmonico osc(1.,0.,0.,0.);
    double tmax = 70.;
    TGraph errore ;
    for (int i=0; i<10; i++) {
        double h=0.1*pow(0.5,i);
        vector<double> x { 0. , 1. } ;
        double t = 0. ;
        int nstep = int(tmax/h+0.5);
        for (int step=0; step<nstep; step++) {
            x = myRK.Passo(t,x,h,osc);
            t = t+h;
        }
        double eps = fabs(x[0] - sin(t));
        errore.SetPoint(i,h,eps);
    }
    TCanvas myCanvas("errore","Errore in funzione del passo",600,600);
    errore.Draw("ALP");
    // ...
    myApp.Run();
}
```

Calcolo il numero di passi necessari

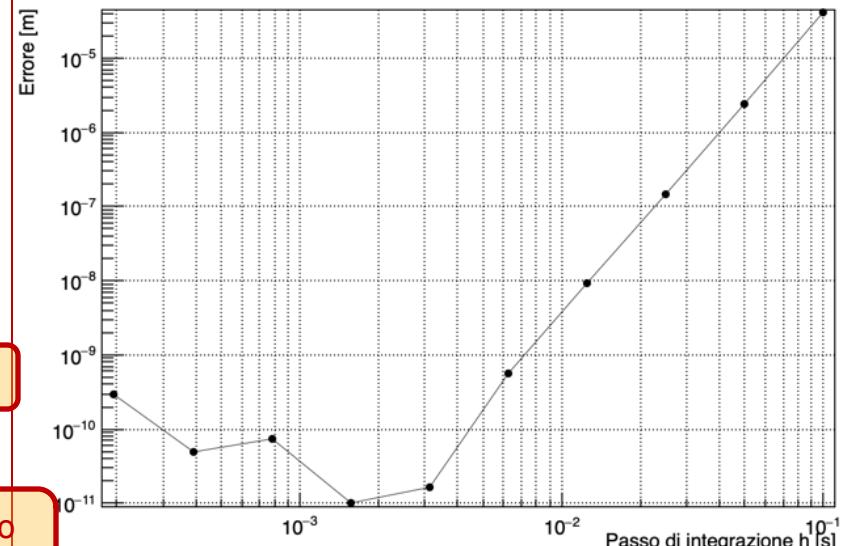
Ciclo esterno : ad ogni giro diminuiamo il passo h

Azzeriamo le condizioni iniziali

Evolvo il sistema fino a tmax

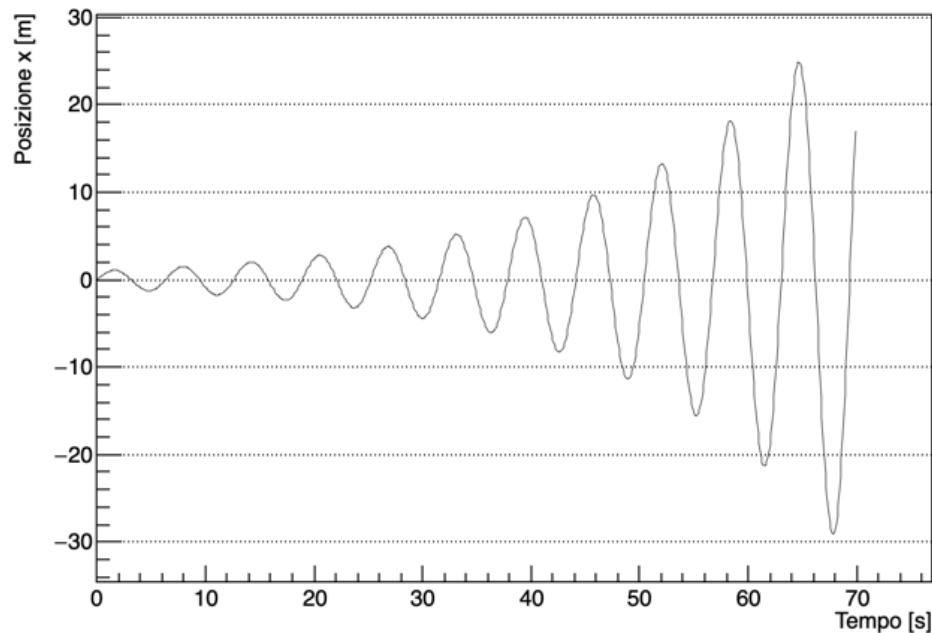
Calcolo l'errore commesso a tmax rispetto alla soluzione esatta

Errore con metodo di Runge-Kutta 4

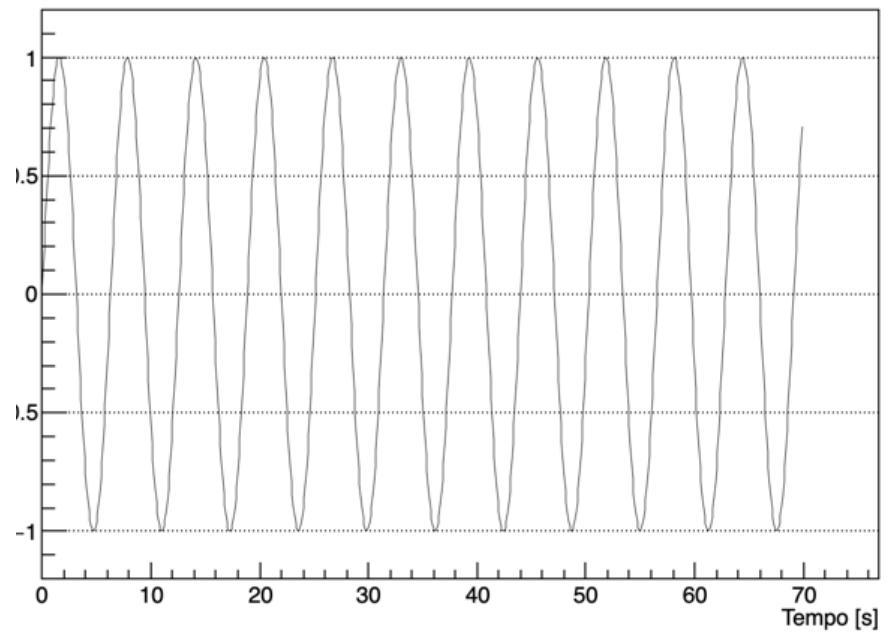


Errore con Eulero vs errore con RK4

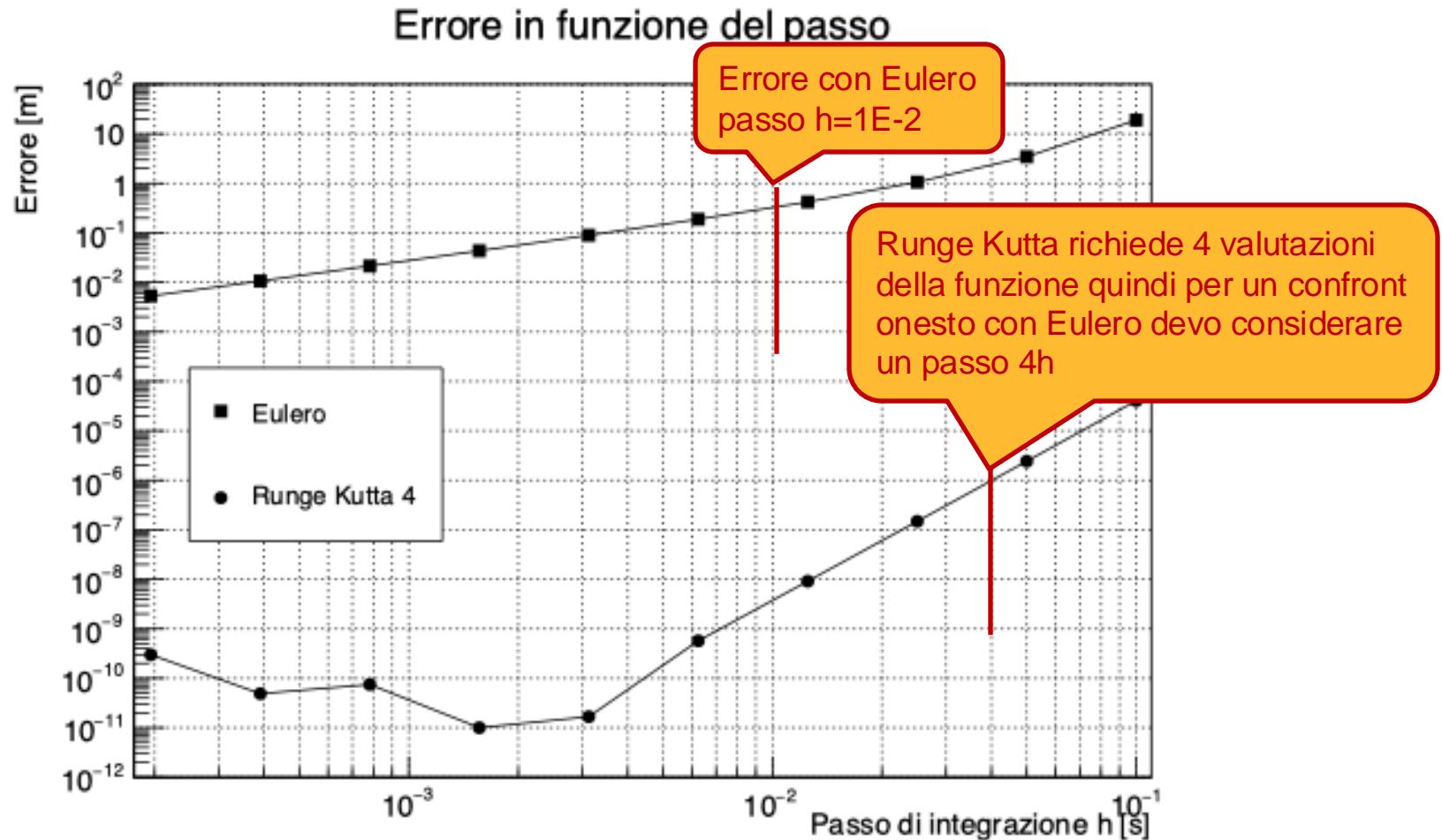
Oscillatore armonico (Eulero $h = 0.1$ s)



Oscillatore armonico (Runge-Kutta $h = 0.1$ s)

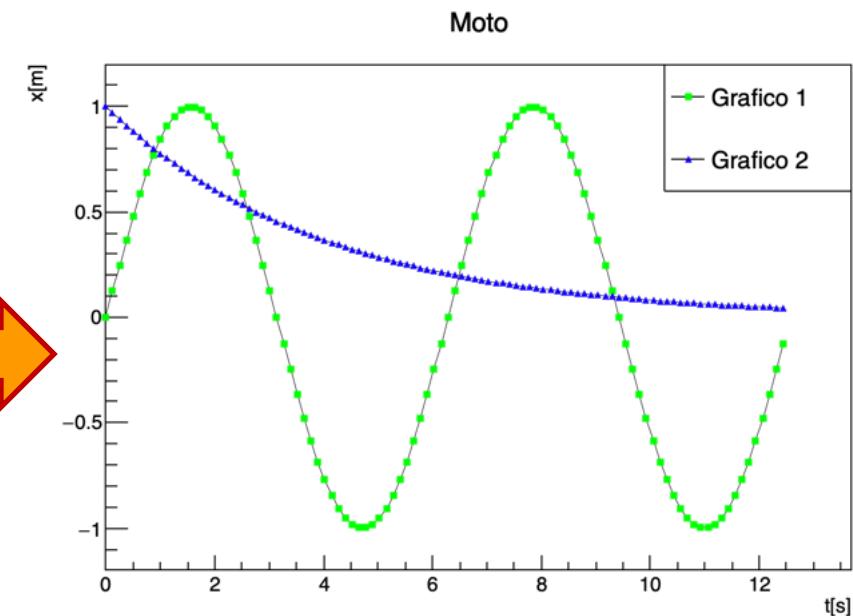


Errore con Eulero vs errore con RK4



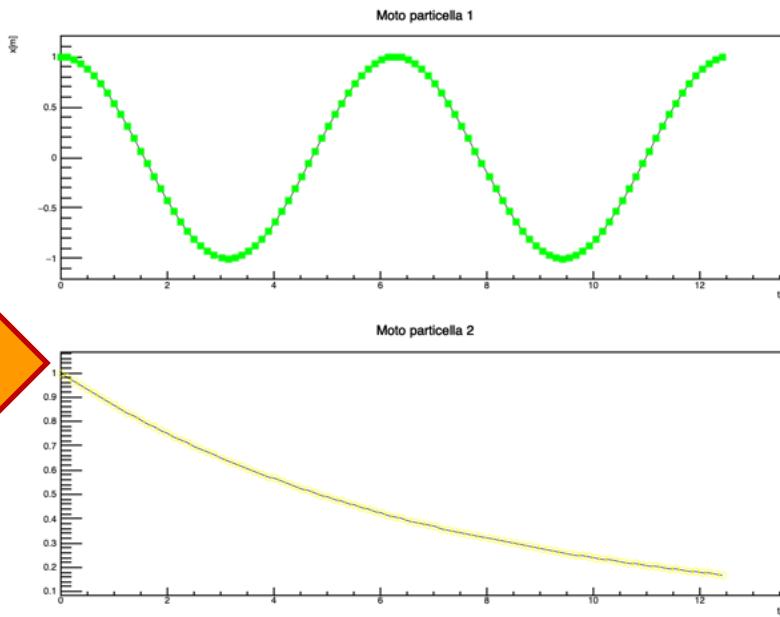
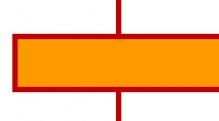
Intermezzo : un TCanvas per due TGraph sovrapposti (+ legenda)

```
TCanvas c1 ;  
  
c1.cd();  
  
g1.SetMarkerStyle(21);  
g1.SetMarkerSize(0.6);  
g1.SetMarkerColor(3);  
g1.GetXaxis()->SetTitle("t[s]");  
g1.GetYaxis()->SetTitle("x[m]");  
g1.SetTitle("Moto");  
  
g1.Draw("ALP");  
  
g2.SetMarkerStyle(22);  
g2.SetMarkerSize(0.6);  
g2.SetMarkerColor(4);  
  
g2.Draw("LPsame");      Opzione "same"  
  
TLegend leg (0.7,0.7,0.9,0.9);  
leg.AddEntry(&g1,"Grafico 1","PL") ;  
leg.AddEntry(&g2,"Grafico 2","PL") ;  
leg.Draw("same");
```



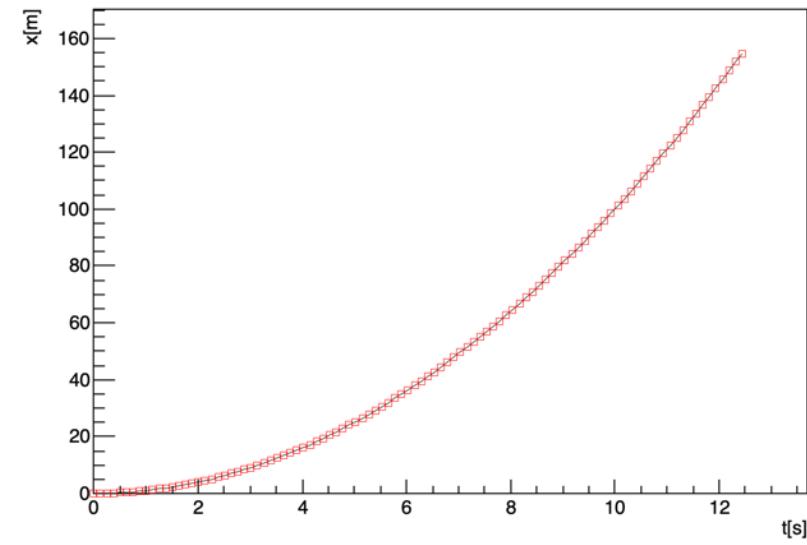
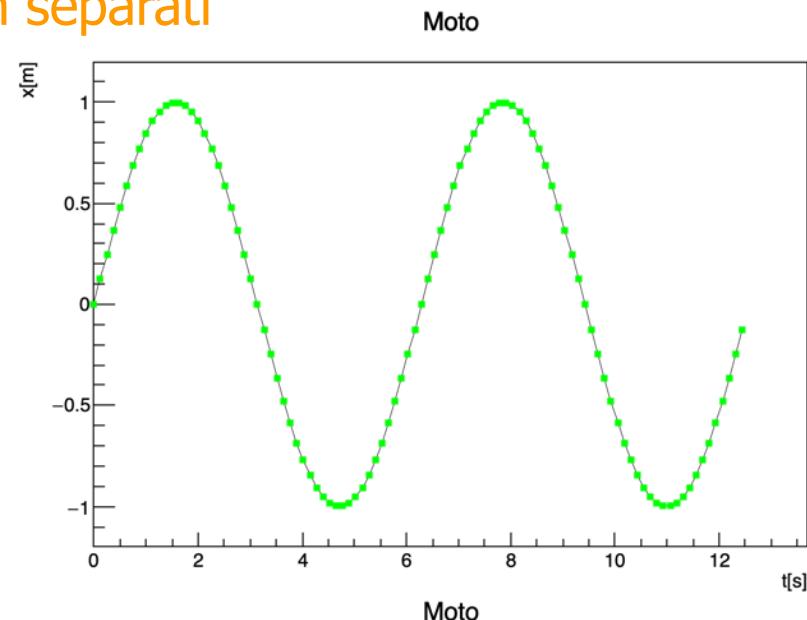
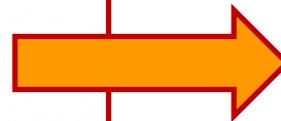
Intermezzo : un TCanvas diviso per due TGraph separati

```
TCanvas c2;  
c2.Divide(1,2); Divide il TCanvas  
c2.cd(1);  
  
g3.SetMarkerStyle(21);  
g3.SetMarkerSize(0.6);  
g3.SetMarkerColor(3);  
g3.GetXaxis()->SetTitle("t[s]");  
g3.GetYaxis()->SetTitle("x[m]");  
g3.setTitle("Moto particella 1");  
  
g3.Draw("ALP");  
  
c2.cd(2);  
g4.SetMarkerStyle(24);  
g4.SetMarkerSize(0.6);  
g4.SetMarkerColor(5);  
g4.GetXaxis()->SetTitle("t[s]");  
g4.GetYaxis()->SetTitle("x[m]");  
g4.setTitle("Moto particella 2");  
g4.Draw("ALP");
```



Intermezzo : due TCanvas per due TGraph separati

```
TCanvas c3;  
c3.cd();  
  
g5.SetMarkerStyle(26);  
g5.SetMarkerSize(0.6);  
g5.SetMarkerColor(6);  
g5.GetXaxis()->SetTitle("t[s]");  
g5.GetYaxis()->SetTitle("x[m]");  
g5.SetTitle("Moto");  
  
g1.Draw("ALP");  
  
TCanvas c4;  
c4.cd();  
g6.SetMarkerStyle(25);  
g6.SetMarkerSize(0.6);  
g6.SetMarkerColor(2);  
g6.GetXaxis()->SetTitle("t[s]");  
g6.GetYaxis()->SetTitle("x[m]");  
g6.SetTitle("Moto");  
g6.Draw("ALP");
```



Esercizio 8.3 : periodo del pendolo

```
#include "EquazioniDifferenziali.h"
#include <iostream>

// ...

#include "TApplication.h"
#include "TGraph.h"

// ...

int main(int argc, char** argv) {
    TApplication myApp("myApp",0,0);

    RK4 myRK4;
    PendoloReale osc(1.);
    double h=1.E-3;
    int nsteps=30;

    TGraph periodo ;
    TGraph differenza ;

    for (int i=0; i<nsteps; i++) {
        double A=0.1*(i+1);
        double v=0.;
        double t = 0.;

        vector<double> x { -A , v };

        while ( ... ) {

            ...
            x = myRK4.Passo(t,x,h,osc);
            t = t+h;
        }

        t = ... ;
        periodo.SetPoint(i,A,2.*t);
    }

    TCanvas myCanvas("periodo","Periodo in funzione del passo",600,600);
    periodo.Draw("ALP");

    // ...

    myApp.Run();
}
```

Calcolo periodo

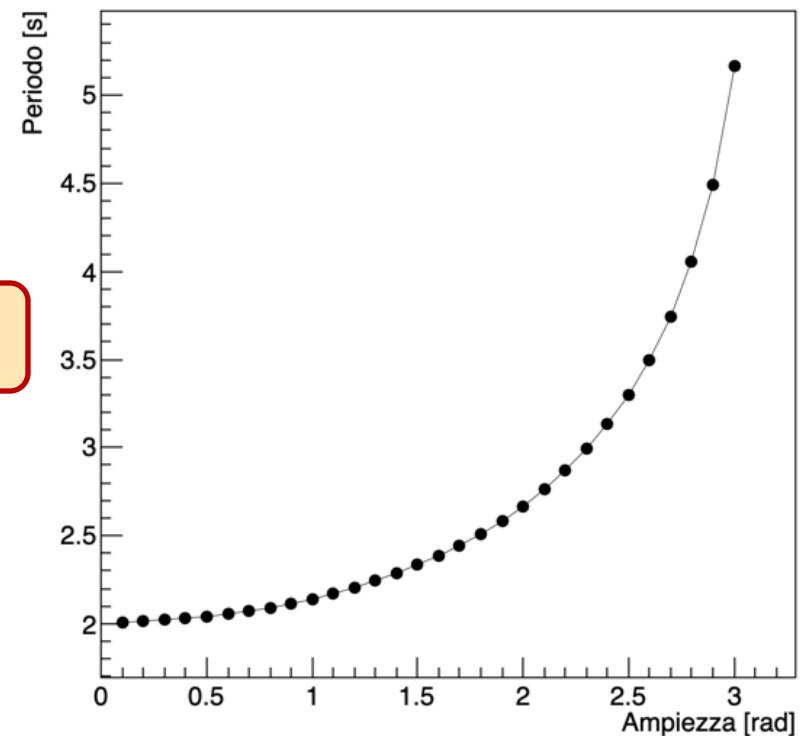
Aumento ogni volta la posizione iniziale

Impongo le condizioni iniziali (ampiezza A e velocità = 0)

Evolvo il sistema fino ad ampiezza massima (quando velocità cambia di segno ?)

$$\frac{d^2\theta}{dt^2} = - \left(\frac{g}{l} \right) \sin\theta$$

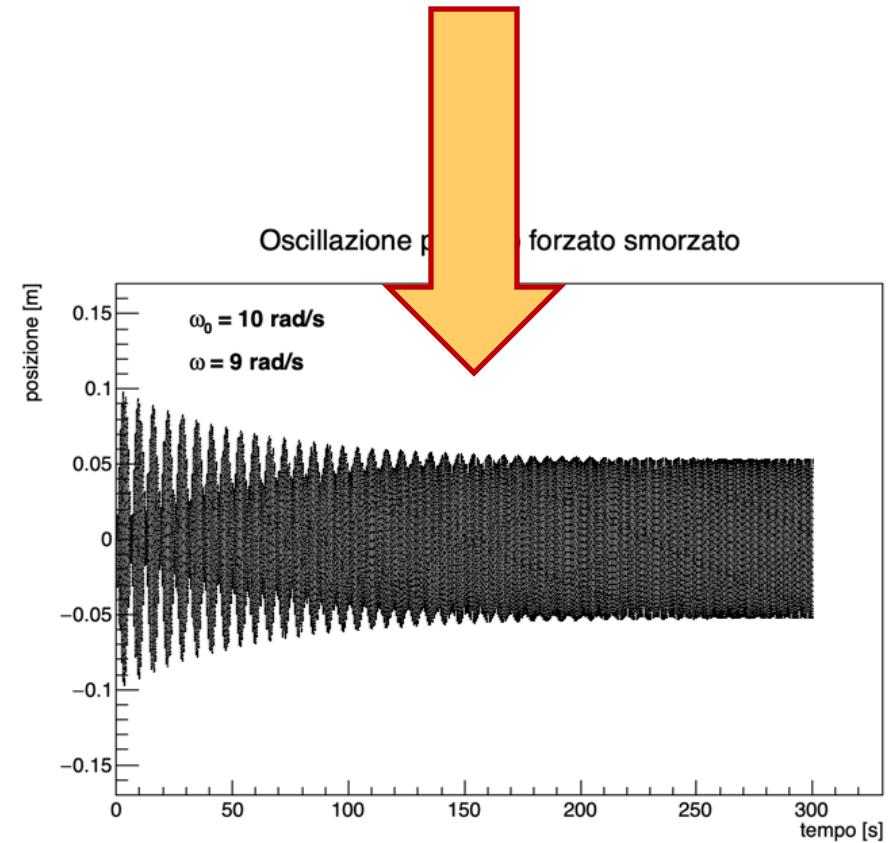
Periodo di oscillazione



Esercizio 8.4 : curva di risonanza

```
#include "EquazioniDifferenziali.h"
// ...
int main(int argc, char** argv) {
    TApplication myApp("myApp",0,0);
    RK4 myRK4;
    double omega0 = 10.;
    double omega = 9.;
    double alpha = 1./30.;
    double dumptime = 10./alpha;
    double h=1.E-2;
    double tmax = 300.;
    int nstep = int(tmax/h+0.5);
    TGraph moto;
    OscillatoreArmonico osc(omega0,alpha,omega);
    double t = 0.;
    vector<double> x { 0. , 0. } ;
    for (int step=0; step<nstep; step++) {
        x = myRK4.Passo(t,x,h,osc);
        moto.SetPoint(step, t , x[0] ) ;
        t = t+h;
    }
    TCanvas myCanvas("Moto","Oscillatore armonico smorzato con forzante",600,600);
    // draw options ....
    moto.Draw("ALP");
    myApp.Run();
}
```

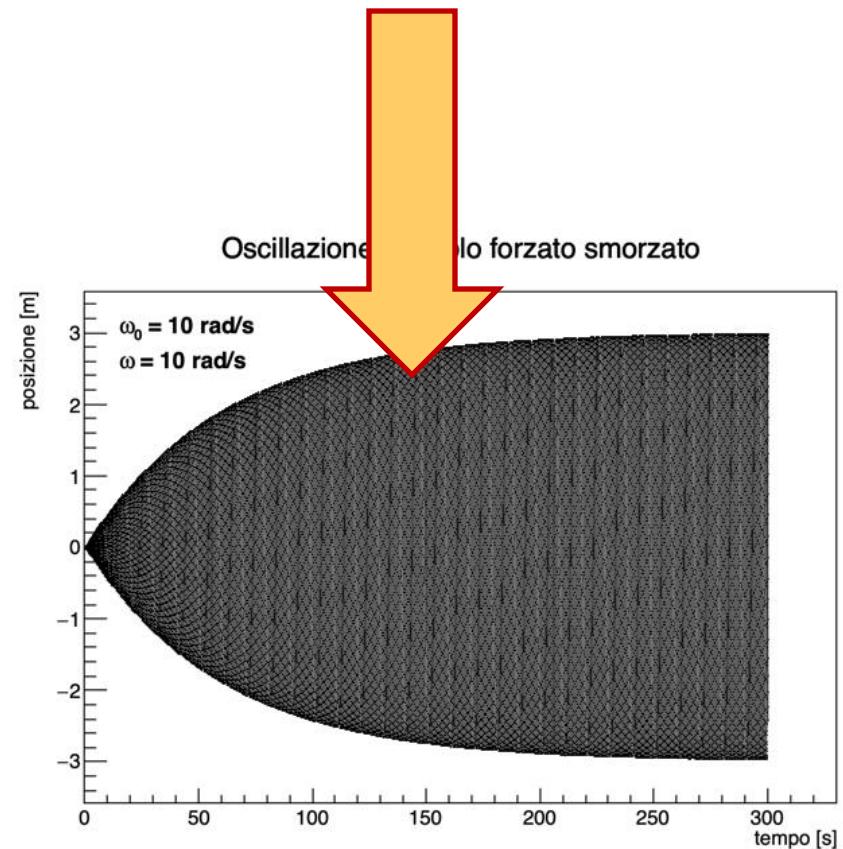
$$\frac{d^2x}{dt^2} = -\omega_0^2 x - \alpha \frac{dx}{dt} + \sin(\omega t)$$



Esercizio 8.4 : curva di risonanza

```
#include "EquazioniDifferenziali.h"
// ...
int main(int argc, char** argv) {
    TApplication myApp("myApp",0,0);
    RK4 myRK4;
    double omega0 = 10.;
    double omega = 9.; double omega= 10;
    double alpha = 1./30.;
    double dumptime = 10./alpha;
    double h=1.E-2;
    double tmax = 300.;
    int nstep = int(tmax/h+0.5);
    TGraph moto;
    OscillatoreArmonico osc(omega0,alpha,omega);
    double t = 0.;
    vector<double> x { 0. , 0. } ;
    for (int step=0; step<nstep; step++) {
        x = myRK4.Passo(t,x,h,osc);
        moto.SetPoint(step, t , x[0] ) ;
        t = t+h;
    }
    TCanvas myCanvas("Moto","Oscillatore armonico smorzato con forzante",600,600);
    // draw options ....
    moto.Draw("ALP");
    myApp.Run();
}
```

$$\frac{d^2x}{dt^2} = -\omega_0^2 x - \alpha \frac{dx}{dt} + \sin(\omega t)$$



Esercizio 8.4 : curva di risonanza

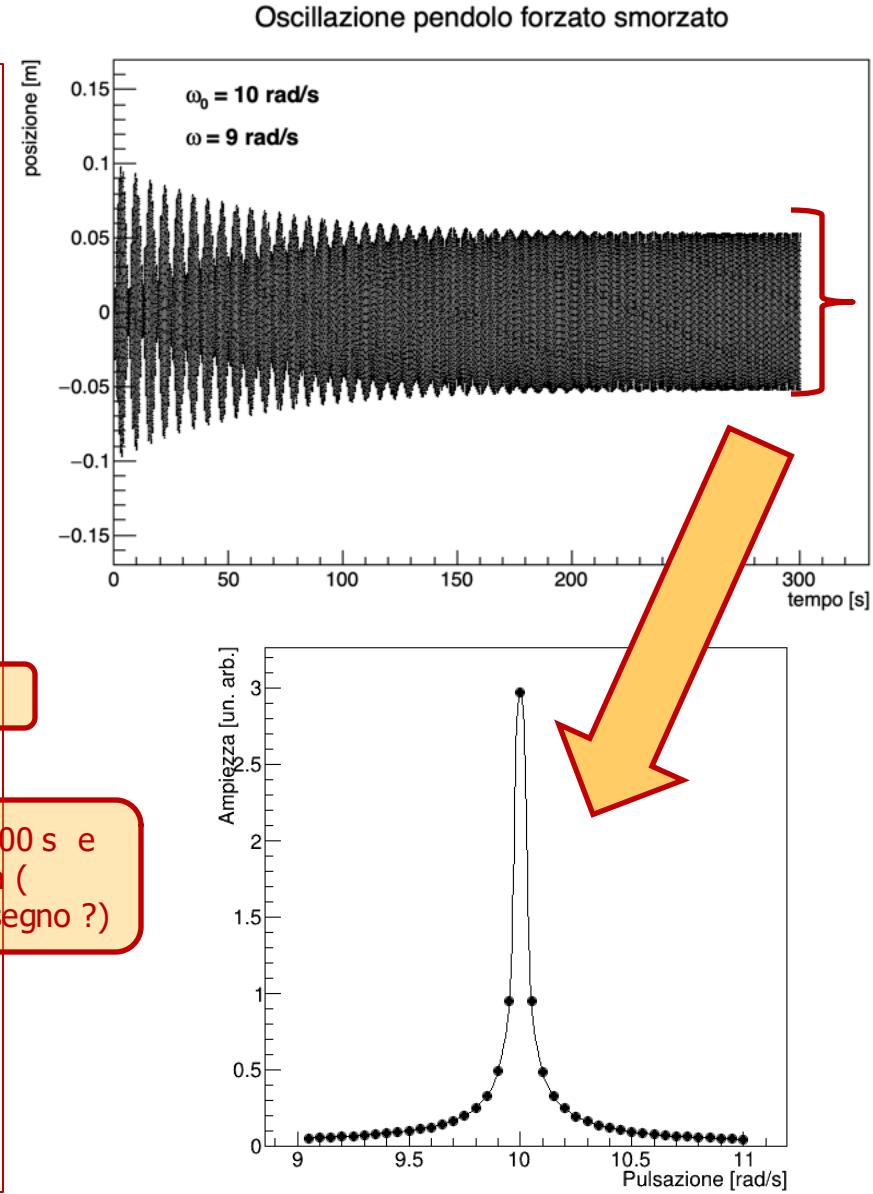
```
#include "EquazioniDifferenziali.h"
// ...
#include "TApplication.h"
#include "TGraph.h"
// ...
int main(int argc, char** argv) {
    TApplication myApp("myApp",0,0);
    RK4 myRK4;
    double omega0 = 10.;
    double alpha = 1./30.;
    double dumptime = 10./alpha;
    double h=1.E-2;
    int nsteps=40;
    TGraph ampiezza;
    for (int i=0; i<nsteps; i++) {
        double omega = 0.9*omega0+(0.2*omega0/nsteps)*(i+1);
        OscillatoreArmonico osc(omega0,alpha,omega);
        double t = 0.;
        vector<double> x { 0., 0. };
        double v = 0;
        unsigned int index = 0;
        while ( .... ) {
            x = myRK4.Passo(t,x,h,osc);
            t = t+h;
        }
        ampiezza.SetPoint(i,omega,fabs(x[0]));
    }
    TCanvas myCanvas("ampiezza","Ampiezza in funzione della frequenza forzante",600,600);
    ampiezza.Draw("ALP");
    myApp.Run();
}
```

Aumento ogni volta la frequenza della forzante

Impongo le condizioni iniziali

Evolvo il sistema fino a $t > 300$ s e poi cerco ampiezza massima (quando velocità cambia di segno ?)

Calcolo ampiezza



Breve dimostrazione su esercizi facoltativi

Qualche riflessione sulla convergenza dei metodi di integrazione

Definizione : metodo one-step

Algoritmo di integrazione di equazioni differenziali che dipende solo dalla posizione del punto precedente

$$\begin{cases} x'(t) = f(t, x(t)) \\ x(t_0) = x_0 \end{cases} \Rightarrow x_{n+1} = x_n + h\Phi(t_n, x_n, f)$$

Definizione : errore locale di troncamento

Dice di quanto mi sbaglio in un singolo passo
(assumendo che il precedente sia 'esatto')

$$\forall (t_n, x_n) \in R \text{ con } R = \{(t, x) : t_0 < t < t_f \text{ e } x \in \mathbb{R}\}$$

$$E_{loc}(t_n, x_n, f, h) = (x(t_{n+1}) - x_{n+1}) = x(t_{n+1}) - [x_n + h\Phi(t_n, x_n, f)]$$

Definizione : errore locale di troncamento unitario

$$\varepsilon_{loc}(t_n, x_n, f, h) = \frac{E_{loc}(t_n, x_n, f, h)}{h} = \frac{x(t_{n+1}) - [x_n + h\Phi(t_n, x_n, f)]}{h}$$

Qualche riflessione sulla convergenza dei metodi di integrazione

Definizione : consistenza

Il metodo Φ è consistente di ordine p se $\forall (t_n, x_n) \in R$ e $f \in C^p(\mathbb{R}) \Rightarrow \varepsilon_{loc}(t_n, x_n, f, h) = O(h^p) \quad h \rightarrow 0$

Dice di quanto mi sbaglio quando accumulo n errori locali per ottenere la stima della soluzione dopo n passi

Definizione : errore globale

Data una successione di valori $t_0 < t_1 < t_2 < \dots < t_N = t_f$ in $[t_0, t_f]$ (dove $t_{i+1} = t_i + h$ con $h = (t_f - t_0)/N$) definiamo errore globale $\varepsilon_{glob}(t_n, x_n, f, h) = x(t_n) - x_n$

Se il metodo è convergente l'errore che commetto per raggiungere t_N partendo da t_0 tende a zero per h che tende a zero (e N che tende a infinito)

Definizione : ordine di convergenza

Il metodo di integrazione di equazioni differenziali è convergente di ordine p in $[t_0, t_f]$ se $\max_{0 < n < N} |\varepsilon_{glob}(t_n, x_n, f, h)| = O(h^p)$ con $f \in C^p(\mathbb{R})$

Qualche riflessione sulla convergenza dei metodi di integrazione

Teorema di convergenza :

Se un metodo $\Phi(t, x, h)$ è consistente di ordine p in $[t_0, t_f]$ e $f(t, x)$ è Lipschtziana in x
 $\forall t \in [t_0, t_f] \Rightarrow \Phi(t, x, h)$ è convergente di ordine p in $[t_0, t_f]$

- Metodo di Eulero : ad ogni step h l'errore (locale) commesso è di ordine h^2 e quindi è consistente di ordine $p=1$. Se la funzione f è "buona" (eg. Lipschitziana in un intervallo) allora il metodo ha un buon comportamento anche globale nell'intervallo con un errore che scala come h
- Metodo di Runge-Kutta : ad ogni step h l'errore (locale) commesso è di ordine h^5 e quindi è consistente di ordine 4. Se la funzione f è "buona" (eg. Lipschitziana in un intervallo) allora il metodo ha un buon comportamento anche globale nell'intervallo con un errore che scala come h^4

Stima runtime dell'errore

- Integrando una equazione differenziale di cui non conosciamo la soluzione esatta come possiamo stimare l'errore che stiamo commettendo nel calcolo numerico della soluzione ad un istante t_N (dopo N passi) ? Possiamo utilizzare un trucchetto come quello già usato per le formule di quadrature numerica (qui illustrato per il metodo di RK4)

$$\begin{cases} |x_N - x| \cong kh_N^4 \quad (= \varepsilon) & h_N = \frac{t_N - t_0}{N} \\ |x_{2N} - x| \cong k \left(\frac{h_N}{2} \right)^4 \end{cases}$$

$$\Delta_N = |x_N - x_{2N}| = \frac{15}{16} kh_N^4 \Rightarrow \boxed{\varepsilon = kh_N^4 = \Delta_N \frac{16}{15}}$$

- Come stimare il passo h necessario per ottenere una precisione $\bar{\varepsilon}$?

Stima runtime dell'errore

- Cerchiamo ora di stimare quale passo h sarebbe necessario per ottenere la precisione $\bar{\varepsilon}$ voluta dall'utente ? Dalle equazioni della slide precedente (per RK4)

$$\varepsilon = kh_N^4 = \Delta_N \frac{16}{15} \Rightarrow k = \frac{\Delta_N}{h_N^4} \frac{16}{15}$$

$$\bar{\varepsilon} = (\bar{x} - x) = k\bar{h}^4 \Rightarrow \bar{h} = \sqrt[4]{\bar{\varepsilon} \frac{15}{16} \frac{h_N^4}{\Delta_N}}$$

- Fisso un istante di tempo al quale voglio calcolare la precisione della mia integrazione. Evolo il sistema con passo h_N e $h_N/2$, stimo il valore di k e infine il valore di \bar{h} necessario per ottenere la precisione $\bar{\varepsilon}$

Breve recap sulla compilazione

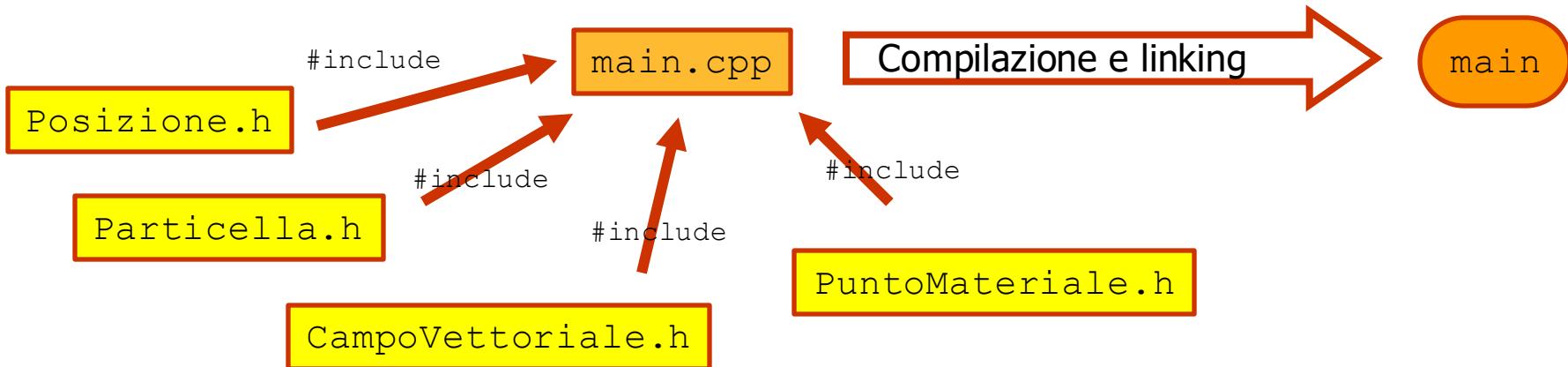
Compilazione : option 1

- Codice composto da : main.cpp, Particella.h, PuntoMateriale.h, Posizione.h, CampoVettoriale.h

NOME del target
= nome del file
di output !

```
LIBS = $(shell root-config --libs)
INCS = $(shell root-config --cflags)

main: main.cpp Posizione.h Particella.h CampoVettoriale.h PuntoMateriale.h
      g++ -o main main.cpp ${INCS} ${LIBS}
```



- Vantaggi : un solo file (.h), una sola istruzione di compilazione
- Svantaggi : una modifica ad una qualsiasi delle classi o funzioni causa una ricompilazione totale.
Leggibilità del codice .h può diventare complicata

Compilazione : option 2

- Codice composto da : main.cpp, Particella.cpp, PuntoMateriale.cpp, Posizione.cpp, CampoVettoriale.cpp (e i relativi relative files .h)

```
INCS=`root-config --cflags`  
LIBS=`root-config --libs`  
  
main: main.cpp Posizione.cpp Partcella.cpp CampoVettoriale.cpp PuntoMateriale.cpp Posizione.h Partcella.h CampoVettoriale.h PuntoMateriale.h  
g++ -o main main.cpp Posizione.cpp Partcella.cpp CampoVettoriale.cpp PuntoMateriale.cpp ${INCS} ${LIBS}
```



- Vantaggi : una sola istruzione di compilazione. Leggibilita' del codice migliore (?)
- Svantaggi : una modifica ad una qualsiasi delle classi o funzioni causa una ricompilazione totale.

Compilazione : option 3

- Codice composto da : main.cpp, Particella.cpp, PuntoMateriale.cpp, Posizione.cpp, CampoVettoriale.cpp (e relativi files .h)

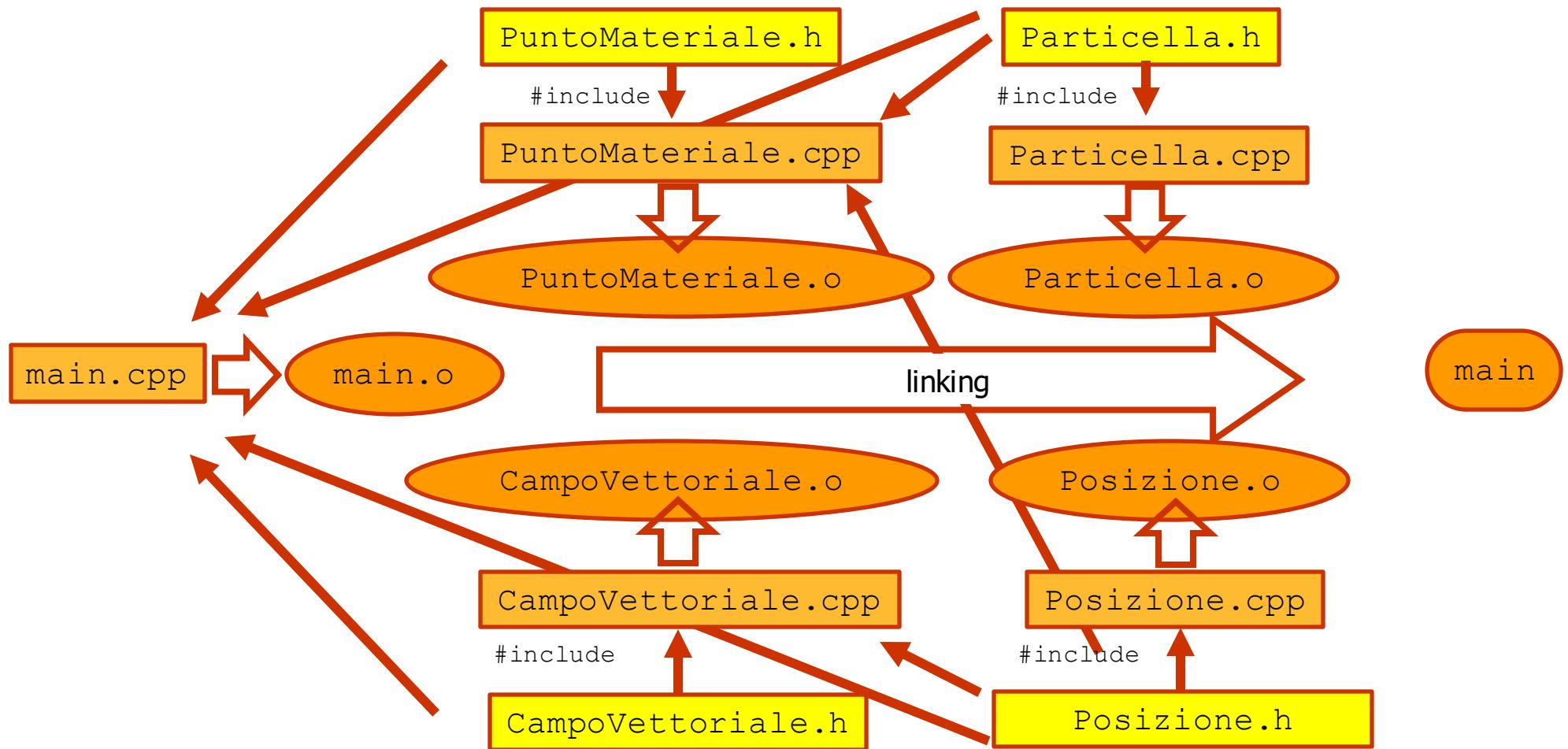
```
INCS='root-config --cflags'  
LIBS='root-config --libs'  
  
main: main.o Posizione.o CampoVettoriale.o Particella.o  
        g++ -o main main.o Posizione.o CampoVettoriale.o Particella.o ${LIBS}  
  
main.o: main.cpp  
        g++ -c main.cpp -o main.o ${INCS}  
  
Particella.o: Particella.cpp Particella.h  
        g++ -c Particella.cpp -o Particella.o  
  
Posizione.o: Posizione.cpp Posizione.h  
        g++ -c Posizione.cpp -o Posizione.o  
  
PuntoMateriale.o: PuntoMateriale.cpp PuntoMateriale.h Posizione.h Particella.h  
        g++ -c PuntoMateriale.cpp -o PuntoMateriale.o  
  
CampoVettoriale.o: CampoVettoriale.cpp CampoVettoriale.h Posizione.h  
        g++ -c CampoVettoriale.cpp -o CampoVettoriale.o  
  
clean:  
        rm *.o
```

`\${LIBS}` is needed in the linking phase (link root pre-compiled libraries)

`\${INCS}` is needed when compiling user code which makes use of ROOT objects (specify where the .h are)

- Vantaggi : una modifica ad una qualsiasi delle classi o funzioni causa una ricompilazione parziale
- Svantaggi : makefile piu' complicato

Compilazione : option 3 (schematic view)



Stanchi di makefiles che diventano sempre piu' lunghi ? Provate questo

Definire una sorta di regola generale per compilare i .o necessari

```
myLIBS=Posizione.o PuntoMateriale.o CampoVettoriale.o Particella.o  
RFLAGS:=`root-config --cflags` `root-config --libs`  
  
main: main.cpp ${myLIBS}  
      g++ -Wall -o $@ ${RFLAGS}  
  
.o : %.cpp %.h  
      g++ -Wall -c $< ${RFLAGS}
```

\$^ indica la lista delle dipendenze

\$@ indica il nome del target

Questo indica la prima dipendenza
(ie the cpp file)

In questo modo pero' una dipendenza da diversi .h non e' considerata

Stanchi di makefiles che diventano sempre piu' lunghi ? Provate questo

Definire una sorta di regola generale per compilare i .o necessari

```
myDEPS=Posizione.h PuntoMateriale.h CampoVettoriale.h Particella.h  
myLIBS=Posizione.o PuntoMateriale.o CampoVettoriale.o Particella.o  
  
RFLAGS:='root-config --cflags` `root-config --libs`  
  
main: main.cpp ${myLIBS}  
      g++ -Wall -o $@ $^ ${RFLAGS}  
  
.o : %.cpp %.h ${myDEPS}  
      g++ -Wall -c $< ${RFLAGS}
```

Specifica la lista delle dipendenze : non molto efficiente perche' un file .cpp sara' ricompilato ogni volta che uno dei .h viene modificato

Non molto elegante ma dovrebbe funzionare al primo ordine

Professional code

Usare CMAKE : <https://cmake.org/>

Backup

Algoritmi a precisione fissata : improved

- Algoritmo che lavori a precisione fissata (raddoppia il numero di punti ad ogni ciclo finché la condizione sulla precision viene raggiunta)

```
double Integra (double precision, FunzioneBase& f) {  
  
    unsigned int nstep=2;  
    double In = DBL_MAX; ;  
    double I2n = Integra(nstep , f);  
    m_h = (m_b-m_a) /nstep ;  
  
    do {  
        In = I2n;  
        nstep*=2 ;  
        m_h/=2.;  
        for (unsigned int i=1; i<nstep; i+=2 ) m_sum+= f.Eval ( /* .... */ );  
        I2n = m_sign * m_sum * m_h;  
    } while ( fabs( In - I2n) > precision ) ;  
  
    return I2n ;  
  
}
```

caveat : frammenti di codice non provati !

Aggiungo solo i punti necessari