Polimorfismo : a brief recap

In C++ il polimorfismo significa che una chiamata di un metodo (`Eval` nel nostro caso) da parte di un puntatore a classe madre (`FunzioneBase`) causerà l'esecuzione di una funzione diversa a seconda del tipo di oggetto a cui il puntatore punta.

```
class FunzioneBase {

 public:

  virtual double Eval(double x) const =0;

};

class Parabola : public FunzioneBase {

 public:

  virtual double Eval(double x) const override {return m_a*x*x+m_b*x+m_c;}

  ...

};

class Coseno : public FunzioneBase {

 public :

  double Eval(double x) const override { return cos(x);  } ;

  ...

};
```

- I metodi possono essere implementati nella classe base e sovrascritti dalla classe derivata
- La parola chiave `virtual` rende possibile il polimorfismo
- Caso estremo: i membri virtuali puri (`virtual … = 0`) non sono implementati nella classe base e devono essere implementati nella classe derivata
- Una classe con almeno un membro virtuale puro è chiamata classe astratta: non è possibile costruire istanze di una classe astratta!

Un potenziale errore con le classi astratte

```
class FunzioneBase{

public:

  virtual double Eval(double x) const = 0;

};

class Parabola: public FunzioneBase{

public:

  Parabola(double a, double b, double c);
  virtual double Eval(double x) {return m_a*x*x + m_b*x + m_c;};

private:

  double m_a, m_b, m_c;

};
```

```
int main() {

  Parabola p(1,2,3);

}
```

Qui manca una `const`! Non è lo stesso `Eval` nella classe base: la classe derivata è quindi considerata astratta (l'uso della keyword `override` avrebbe generato un chiaro messaggio di errore)

```
clang++-7 -pthread -std=c++17 -o main main.cpp
main.cpp:42:12: error: variable type 'Parabola' is an abstract class
  Parabola p(1,2,3);
           ^
main.cpp:10:18: note: unimplemented pure virtual method 'Eval' in 'Parabola'
  virtual double Eval(double x) const = 0;
                 ^
1 error generated.
compiler exit status 1
```

Polimorfismo : a brief recap

In C++ il polimorfismo significa che una chiamata di un metodo (`Eval`) da parte di un puntatore a classe madre (`FunzioneBase`) causerà l'esecuzione di una funzione diversa a seconda del tipo di oggetto a cui il puntatore punta.

```
class FunzioneBase {

 public:

  virtual double Eval(double x) const =0;

};

class Parabola : public FunzioneBase {

 public:

  virtual double Eval(double x) const override {return m_a*x*x+m_b*x+m_c;}

  ...

};

class Coseno : public FunzioneBase {

 public :

  double Eval(double x) const override { return cos(x);  } ;

  ...

};
```

La keyword `override` fa si che un errore venga segnalato se il metodo virtuale con lo stesso nome viene sovrascritto nella classe figlia con qualche parametro diverso

- I metodi possono essere implementati nella classe base e sovrascritti dalla classe derivata
- La parola chiave `virtual` rende possibile il polimorfismo
- Caso estremo: i membri virtuali puri (`virtual … = 0`) non sono implementati nella classe base e devono essere implementati nella classe derivata
- Una classe con almeno un membro virtuale puro è chiamata classe astratta: non è possibile costruire istanze di una classe astratta!

Alcune osservazioni su classi astratte e polimorfiche

```
int main () {

  FunzioneBase * myfbase = new FunzioneBase() ;

  Parabola     * mypara  = new Parabola (3,-5,2) ;

  FunzioneBase * myparapoly = new Parabola(3,-5,2) ;

  mypara->GetVertex();

  myparapoly->GetVertex() ;

  Bisezione bisettore ;

  double zero = bisettore.CercaZeri( 0.9, 1.1, mypara, 0.001 ) ;

  double zero = bisettore.CercaZeri( 0.9, 1.1, myparapoly, 0.001 ) ;

}
```

NO! Impossibile creare un oggetto di tipo `FunzioneBase`

NO! `myparapoly` è un puntatore a `FunzioneBase`, non conosce i metodi `Parabola`

Sono accettate sia una `Parabola*` sia una `FunzioneBase*` (e un puntatore a qualsiasi altra classe che eredita da `FunzioneBase`)

Il metodo accetta un puntatore a `FunzionaBase`

```
double Bisezione::CercaZeri(double xmin,
                            double xmax,
                            const FunzioneBase * f,
                            double prec=0.001) {

  double fa = f->Eval(a);
  double fb = f->Eval(b);

  // ... code for bisezione
```

Ecco la magia: `Eval()` è quello che viene passato alla funzione tramite il puntatore

Come impostare i parametri in un algoritmo ? Concentriamoci su `prec`

```
#ifndef __AlgoZeri_h__
#define __AlgoZeri_h__

#include "Funzioni.h"

const double prec_default = 0.001;
const unsigned int nmax_default = 100;

class Solutore {

public:

  Solutore() {
    m_a = 0;
    m_b = 0;
    m_prec = prec_default ; // Precisione di default;
    m_niterations = 0 ;
    m_nmax = nmax_default ;
  }

  Solutore( double prec ) {
    m_a = 0;
    m_b = 0;
    m_prec = prec ;
    m_niterations = 0 ;
    m_nmax = nmax_default ;
  }

  virtual ~Solutore() {;} ;

  // virtual double CercaZeri(double xmin , double xmax , const FunzioneBase* f, double prec = 0.001, unsigned int nmax = 100) = 0;
  // virtual double CercaZeri(double xmin , double xmax , const FunzioneBase& f, double prec = 0.001, unsigned int nmax = 100) = 0;

  virtual double CercaZeri(double xmin , double xmax , const FunzioneBase* f, double prec = prec_default, unsigned int nmax = nmax_default ) = 0;
  virtual double CercaZeri(double xmin , double xmax , const FunzioneBase& f, double prec = prec_default, unsigned int nmax = nmax_default ) = 0;

  void SetPrecisione(double epsilon) { m_prec = epsilon; }
  double GetPrecisione() { return m_prec;}

  void SetNMaxiterations(unsigned int n ) { m_nmax = n
  unsigned int GetNMaxiterations () { return m_nmax ; }

  unsigned int GetNiterations () { return m_niterations

  virtual bool Trovato() const =0;
  virtual double Incertezza() const =0;

protected:

  double m_a, m_b;
  double m_prec;
  unsigned int m_nmax, m_niterations;

};
```

Ci sono molte opzioni diverse per impostare i valori dei parametri dell'algoritmo

Costruttore senza argomenti: imposta la precisione su un numero predefinito

Costruttore con precisione come argomento: set `m_prec`

Si può modificare la precisione quando si chiama il metodo `CercaZeri`

Si può modificare la precisione quando con il metodo dedicato `SetPrecisione`

Come impostare i parametri in un algoritmo ? Opzione 1 : nella classe

```
#ifndef __AlgoZeri_h__
#define __AlgoZeri_h__

#include "Funzioni.h"

const double prec_default = 0.001;
const unsigned int nmax_default = 100;

class Solutore {

public:

  Solutore() {
    m_a = 0;
    m_b = 0;
    m_prec = prec_default ; // Precisione di default;
    m_niterations = 0 ;
    m_nmax = nmax_default ;
  }

  Solutore( double prec ) {
    m_a = 0;
    m_b = 0;
    m_prec = prec ;
    m_niterations = 0 ;
    m_nmax = nmax_default ;
  }

  virtual ~Solutore() {;} ;

  // virtual double CercaZeri(double xmin , double xmax , const FunzioneBase* f, double prec = 0.001, unsigned int nmax = 100) = 0;
  // virtual double CercaZeri(double xmin , double xmax , const FunzioneBase& f, double prec = 0.001, unsigned int nmax = 100) = 0;

  virtual double CercaZeri(double xmin , double xmax , const FunzioneBase* f, double prec = prec_default, unsigned int nmax = nmax_default ) = 0;
  virtual double CercaZeri(double xmin , double xmax , const FunzioneBase& f, double prec = prec_default, unsigned int nmax = nmax_default ) = 0;

  void SetPrecisione(double epsilon) { m_prec = epsilon; }
  double GetPrecisione() { return m_prec;}

  void SetNMaxiterations(unsigned int n ) { m_nmax = n ; }
  unsigned int GetNMaxiterations () { return m_nmax ; } ;

  unsigned int GetNiterations () { return m_niterations ; } ;

  virtual bool Trovato() const =0;
  virtual double Incertezza() const =0;

protected:

  double m_a, m_b;
  double m_prec;
  unsigned int m_nmax, m_niterations;

};
```

La precisione è un membro protetto della classe, l'unico modo per modificarla è tramite i costruttori e `SetPrecision()`

Come impostare i parametri in un algoritmo ? Option 2 : nel metodo

```
#ifndef __AlgoZeri_h__
#define __AlgoZeri_h__

#include "Funzioni.h"

const double prec_default = 0.001;
const unsigned int nmax_default = 100;

class Solutore {

public:

  Solutore() {
    m_a = 0;
    m_b = 0;
    m_prec = prec_default ; // Precisione di default;
    m_niterations = 0 ;
    m_nmax = nmax_default ;
  }

  Solutore( double prec ) {
    m_a = 0;
    m_b = 0;
    m_prec = prec ;
    m_niterations = 0 ;
    m_nmax = nmax_default ;
  }

  virtual ~Solutore() {;} ;

  // virtual double CercaZeri(double xmin , double xmax , const FunzioneBase* f, dou      0.001, unsigned int nmax = 100) = 0;
  // virtual double CercaZeri(double xmin , double xmax , const FunzioneBase& f, doubl ec = 0.001, unsigned int nmax = 100) = 0;

  virtual double CercaZeri(double xmin , double xmax , const FunzioneBase* f, double prec = prec_default, unsigned int nmax = nmax_default ) = 0;
  virtual double CercaZeri(double xmin , double xmax , const FunzioneBase& f, double prec = prec_default, unsigned int nmax = nmax_default ) = 0;

  void SetPrecisione(double epsilon) { m_prec = epsilon; }
  double GetPrecisione() { return m_prec;}

  void SetNMaxiterations(unsigned int n ) { m_nmax = n ; }
  unsigned int GetNMaxiterations () { return m_nmax ; } ;

  unsigned int GetNiterations () { return m_niterations ; } ;

  virtual bool Trovato() const =0;
  virtual double Incertezza() const =0;

protected:

  double m_a, m_b;
  double m_prec;
  unsigned int m_nmax, m_niterations;

};
```

La precisione NON è un membro della classe, il suo valore viene passato come argomento di `CercaZeri` e non memorizzato nella classe

Consistenza !

```
double Bisezione::CercaZeri(double xmin,
                            double xmax,
                            const FunzioneBase & f,
                            double prec ,
                            unsigned int nmax ) {

  Segno sign;

  m_niterations = 0;
  m_prec = prec ;
  m_nmax = nmax ;

  if ( xmin<xmax ) {
    m_a = xmin; m_b = xmax;
  } else {
    m_a = xmax; m_b = xmin;
  }
  double fa = f.Eval(m_a);
  double fb = f.Eval(m_b);
  while ( (m_b-m_a) > m_prec ) {
    double c = 0.5*(m_b+m_a);
    double fc = f.Eval(c);
    if ( m_niterations > m_nmax ) break;
    m_niterations++ ;
    if ( sign.Eval(fa)*sign.Eval(fc) <= 0 ) {
      m_b=c; fb=fc;
    } else if ( sign.Eval(fb)*sign.Eval(fc)<=0 ) {
      m_a=c; fa=fc;
    } else return 0.;
  }
  return 0.5*(m_b+m_a);

}
```

L'algoritmo utilizza `m_prec`, quindi deve essere impostato prima. Se non si vuole usare `m_prec` (membro della classe) si deve mettere `prec` in questo punto. Basta essere coerenti!

Un commento sulla funzione segno

```
double Bisezione::CercaZeri(double xmin,
                            double xmax,
                            const FunzioneBase & f,
                            double prec ,
                            unsigned int nmax ) {

  Segno sign;

  m_niterations = 0;
  m_prec = prec ;
  m_nmax = nmax ;

  if ( xmin<xmax ) {
    m_a = xmin; m_b = xmax;
  } else {
    m_a = xmax; m_b = xmin;
  }
  double fa = f.Eval(m_a);
  double fb = f.Eval(m_b);
  while ( (m_b-m_a) > m_prec ) {
    double c = 0.5*(m_b+m_a);
    double fc = f.Eval(c);
    if ( m_niterations > m_nmax ) break;
    m_niterations++ ;
    if ( sign.Eval(fa)*sign.Eval(fc) <= 0 ) {
      m_b=c; fb=fc;
    } else if ( sign.Eval(fb)*sign.Eval(fc)<=0 ) {
      m_a=c; fa=fc;
    } else return 0.;
  }
  return 0.5*(m_b+m_a);

}
```

La funzione `Segno` può essere costruita come una funzione che eredita da `FunzioneBase` esattamente come e.g. `Parabola`

```
class Segno : public FunzioneBase {
 public:
  Segno() {;} ;
  ~Segno() {;} ;
  virtual double Eval(double x) const override {
    return (x==0.?0.:(x>0?1.:-1));
  }
};
```

Più sicuro computazionalmente controllare il prodotto dei segni che il segno del prodotto !

Una nota sui distruttori delle classi virtuali

Ogni volta che la classe ha almeno una funzione virtuale, anche il distruttore dovrebbe essere dichiarato `virtual`

```
#include <iostream>
using namespace std;
class Base {
public :
  virtual ~Base() { std::cout << "Base destructor called" << endl ;  } ;
  virtual void method() = 0;
};
class Derived : public Base {
public :
  ~Derived() { std::cout << "Derived destructor called" << endl ; }
  void method() { std::cout << "Derived::method() called" << endl ; }
};

int main() {
  Base * myclass = new Derived() ;
  myclass->method();
  delete myclass;
}
```

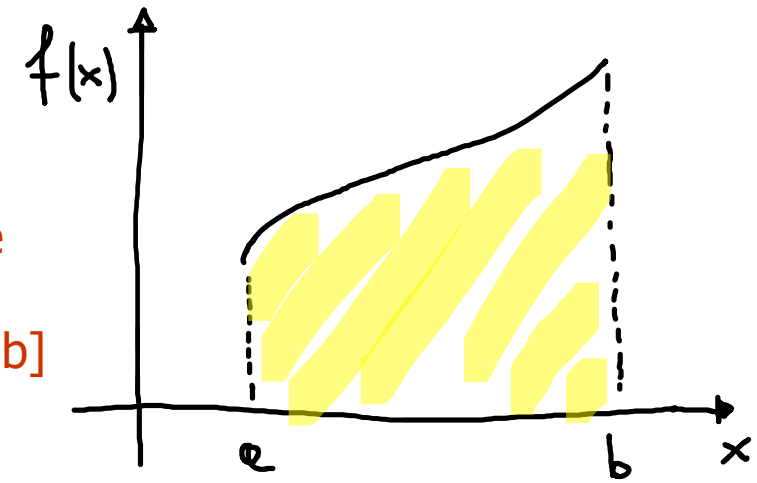
Se il distruttore della classe base non è dichiarato virtuale, `delete myclass` chiamerà il distruttore della classe base e non quello derivato. Un distruttore virtuale è più sicuro poiché tutto ciò che è posseduto dalla classe derivata verrà deallocato correttamente

Algoritmi di quadratura numerica

Laboratorio Trattamento Numerico dei Dati Sperimentali

Prof. L. Carminati
Università degli Studi di Milano

Integrazione numerica



- Proviamo a scrivere un algoritmo numerico per affrontare un problema potenzialmente molto utile ovvero calcolare l'integrale di una funzione "regolare" f(x) in un intervallo chiuso e limitato [a,b]

- L'idea alla base dell'approccio numerico che utilizzeremo in questa sessione di laboratorio è quello di provare a sostituire la funzione con una sua approssimazione (quanto buona?) che sia più facilmente integrabile e che ci permetta di ricavare delle formulette semplici da implementare al calcolatore
  - Scegliamo delle funzioni approssimanti semplici (polinomi !)
  - Studiamo quanto buona è l'approssimazione che compiamo (i.e. di quanto ci sbagliamo nella stima del nostro integrale ? L'errore è accettabile ?)

$$\int_a^b f(x)dx \Rightarrow \int_a^b p_n(x)dx$$

Approssimazione polinomiale: richiamo teoremi fondamentali

- Soluzione più semplice: approssimare la funzione utilizzando dei polinomi di grado n

Teorema di esistenza e unicità del polinomio interpolante

Dati $n+1$ punti di interpolazione (x_i, f_i) per $i = 0 \dots n$ con $x_i \neq x_j$ per $i \neq j$

$\Longrightarrow \exists!$ polinomio $p \in P_n$ tale che $p(x_i) = f_i$ per $i = 0 \dots . n$

Teorema del resto dell'interpolazione polinomiale

Se $f \in C^{n+1}[a,b] \Longrightarrow \exists\, un\ punto\ \xi(x) \in (a,b)\ \ tale\ che$

$$r(x) = f(x) - p_n(x) = \prod_{i=0}^{n}(x - x_i)\frac{f^{n+1}(\xi(x))}{(n+1)!}$$

Costruzione del polinomio interpolante

- Per costruire il polinomio interpolante utilizziamo i polinomi di Lagrange : dato un set di n+1 punti (nodi) $x_0 \ldots x_n \in [a,b]$ il polinomio j-esimo di grado n è definito come

$$l_j^n(x) = \prod_{k=0, k\neq j}^{n} \frac{x - x_k}{x_j - x_k}$$

- È possible dimostrare che gli n+1 polinomi $\{l_0^n(x), \ldots l_n^n(x)\}$ costituiscono una base nello spazio vettoriale P_n dei polinomi di grado n

- Un generico polinomio p_n in P_n può quindi essere scritto come combinazione lineare dei polinomi della base. La particolarità dei polinomi di Lagrange è che i coefficienti c_i della combinazione lineare sono proprio i valori della funzione nei punti x_i !

$$p_n(x) = \sum_{i=0}^{n} c_i l_i^n(x) = \sum_{i=0}^{n} f(x_i) l_i^n(x)$$

Costruzione del polinomio interpolante

- Proviamo ad approssimare il nostro integrale sostituendo la funzione integranda con una sua approssimazione costruita con i polinomi di Lagrange :

$$\int_a^b f(x)dx \cong \int_a^b p_n(x)dx = \int_a^b \left[\sum_{i=0}^{n} f(x_i) l_i^n(x)\right] dx = \sum_{i=0}^{n} f(x_i) \int_a^b l_i^n(x)dx$$

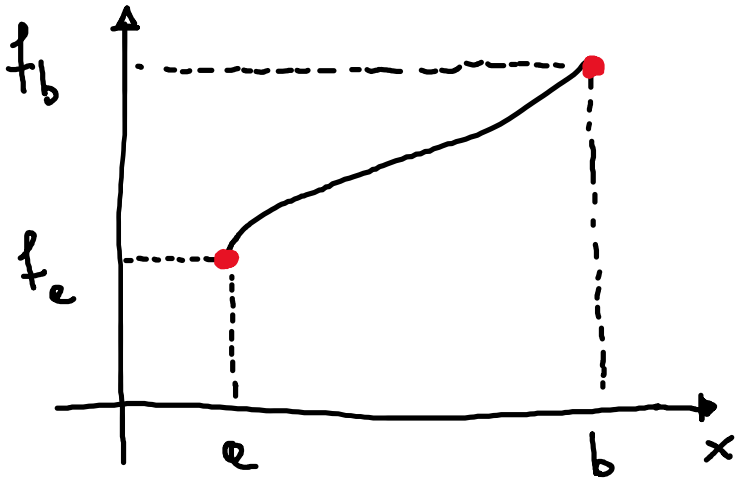
Non dipende dalla funzione ma solo dai punti scelti x_i

- Questo approccio (interpolazione polinomiale su nodi equidistanti) porta alla nascita di una serie di formule di integrazione note come **formule di Newton-Cotes** :
  - Chiuse : se anche i punti $x_0 = a$ e $x_n = b$ sono utilizzati
  - Aperte : se tutti i nodi sono interni all'intervallo

Formule di Newton-Cotes chiuse : caso n = 1 (Trapezoidi)

Proviamo ad applicare la formula generale dei polinomi di Lagrange al caso in cui n=1. Se n=1 significa che ci servono n+1 = 2 valutazioni della funzione.

- Se scegliamo di valutare la funzione negli estremi dell'intervallo a e b $\Rightarrow$ **Metodo dei Trapezoidi** (formula chiusa)



$$(a, f_a = f(a)) \text{ e } \left(b, f_b = f(b)\right)$$

$$\begin{cases} l_0^1(x) = \dfrac{x-b}{a-b} \\ l_1^1(x) = \dfrac{x-a}{b-a} \end{cases} \Rightarrow p_1(x) = f_a \frac{x-b}{a-b} + f_b \frac{x-a}{b-a}$$

$$\int_a^b f(x)dx \cong \tilde{I} = \int_a^b p_1(x)dx =$$

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

Formule di Newton-Cotes chiuse : caso n = 1 (Trapezoidi)

$$\tilde{I} = \int_a^b \left(f_a \frac{x-b}{a-b} + f_b \frac{x-a}{b-a} \right) dx =$$

$$\int_a^b f_a \frac{x}{a-b} dx - \int_a^b f_a \frac{b}{a-b} dx + \int_a^b f_b \frac{x}{b-a} dx - \int_a^b f_b \frac{a}{b-a} dx =$$

$$\frac{f_a}{a-b} \frac{x^2}{2}\Big|_a^b - \frac{f_a b}{a-b}(b-a) + \frac{f_b}{b-a} \frac{x^2}{2}\Big|_a^b - \frac{f_b a}{b-a}(b-a) =$$

$$\frac{f_a}{a-b} \frac{(b^2-a^2)}{2} + f_a b + \frac{f_b}{b-a} \frac{(b^2-a^2)}{2} - f_b a =$$

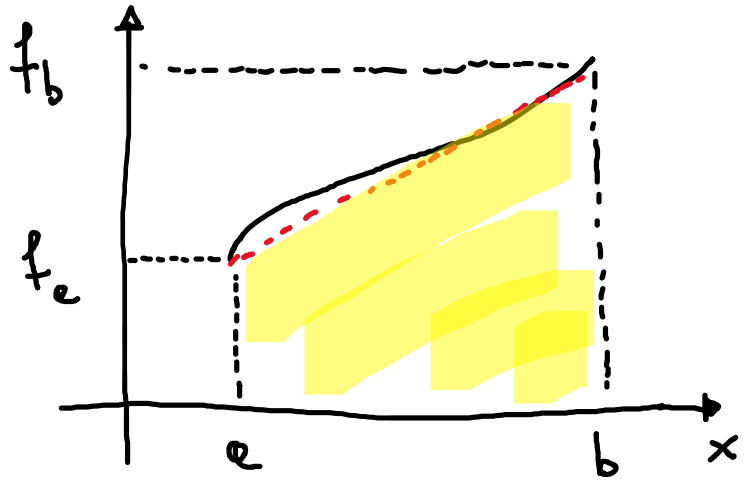
$$-\frac{f_a}{2}(b+a) + f_a b + \frac{f_b}{2}(b+a) - f_b a$$

$$\boxed{\tilde{I} = \frac{f_a + f_b}{2}(b-a)}$$

Formule di Newton-Cotes chiuse : errore per il caso n = 1 (Trapezoidi)

- Nel caso n=1 abbiamo semplicemente approssimato l'area sotto la funzione con un trapezio come in figura e considerato l'area del trapezio come stima dell'integrale

$$\int_a^b f(x)dx \cong \frac{f_a + f_b}{2}(b-a)$$



- Quanto vale l'errore R che commettiamo nella stima del valore dell'integrale quando utilizziamo questa approssimazione polinomiale ? Utilizziamo il teorema del resto

Ipotesi f'' continua in [a,b]

$$r(x) = f(x) - p_n(x) = (x-x_0)(x-x_1)\frac{f''(\xi(x))}{2}$$

$$R = \int_a^b r(x)dx = \frac{f''(\xi)}{2}\int_a^b (x-x_0)(x-x_1)dx$$

Teorema della media di Bonnet

Formule di Newton-Cotes chiuse : caso n = 1 (Trapezoidi)

$$R = \frac{f''(\xi)}{2}\int_a^b (x^2 - xx_1 - xx_0 + x_0x_1)dx$$

$$\frac{f''(\xi)}{2}\left[\frac{x^3}{3} - x_1\frac{x^2}{2} - x_0\frac{x^2}{2} + x_0x_1x\right]_a^b =$$

$$\frac{f''(\xi)}{2}\left[\frac{b^3 - a^3}{3} - b\frac{b^2 - a^2}{2} - a\frac{b^2 - a^2}{2} + ba(b - a)\right] =$$

$$\frac{f''(\xi)}{2}\left[\frac{2b^3 - 2a^3 - 3b^3 + 3ba^2 - 3ab^2 + 3a^3 + 6b^2a + 6ba^2}{6}\right] =$$

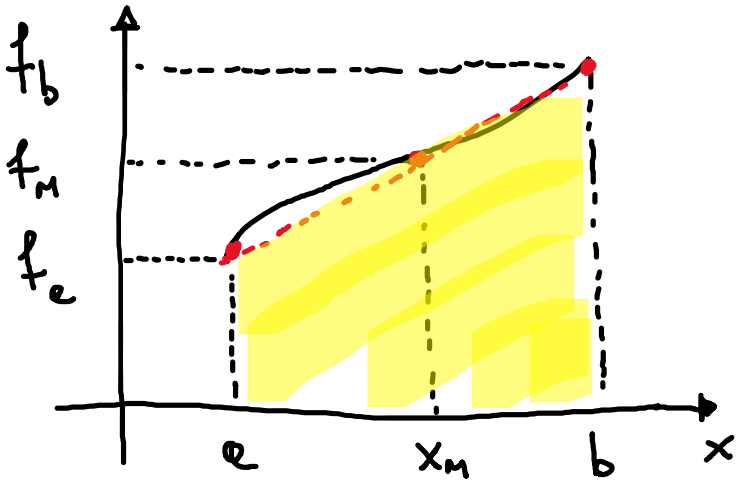
$$-\frac{f''(\xi)}{12}[b^3 - a^3 + 3ba^2 - 3b^2a]$$

$$R = -\frac{f''(\xi)}{12}(b - a)^3 \qquad \xi \in]a, b[$$

Formule di Newton-Cotes chiuse : caso n = 2 (Simpson)

Proviamo ad applicare la formula generale dei polinomi di Lagrange al caso in cui n=2. Se n=2 significa che ci servono n+1 = 3 valutazioni della funzione.

- Se scegliamo di valutare la funzione negli estremi dell'intervallo a e b e nel punto medio $x_M \Longrightarrow$ **Metodo di Simpson** (formula chiusa)



$$\big(a, f_a = f(a)\big), (x_M, f_M = f(x_M), \big(b, f_b = f(b)\big)$$

$$\begin{cases} l_0^2(x) = \dfrac{x - x_M}{a - x_M} \cdot \dfrac{x - b}{a - b} \\ l_1^2(x) = \dfrac{x - a}{x_M - a} \cdot \dfrac{x - b}{x_M - b} \\ l_2^2(x) = \dfrac{x - a}{b - a} \cdot \dfrac{x - x_M}{b - x_M} \end{cases}$$

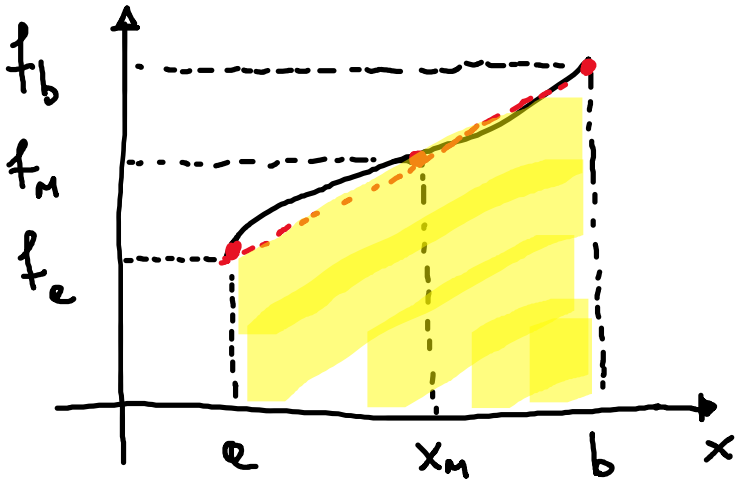
$$p_2(x) = f_a l_0^2(x) + f_M l_1^2(x) + f_b l_2^2(x)$$

$$\int_a^b f(x)dx \cong \int_a^b p_2(x)dx$$

Formule di Newton-Cotes chiuse : caso n = 2 (Simpson)

Proviamo ad applicare la formula generale dei polinomi di Lagrange al caso in cui n=2. Se n=2 significa che ci servono n+1 = 3 valutazioni della funzione.

- Se scegliamo di valutare la funzione negli estremi dell'intervallo a e b e nel punto medio $x_M \Longrightarrow$ **Metodo di Simpson** (formula chiusa)



$$I = \int_a^b f(x)dx = \int_a^b p_2(x)dx + R = \tilde{I} + R$$

Con qualche calcolo si può ricavare

$$\tilde{I} = \frac{b-a}{3}[f(a) + 4f(x_M) + f(b)]$$

$$R = -\frac{(b-a)^5}{2880} f^{IV}(\xi) \quad \xi \in]a,b[$$

Ipotesi f^{IV} continua in [a,b]

Grado di precisione di una formula di quadratura

- Definizione : *una formula di quadratura ha un grado di precisione k se è esatta quando la funzione integranda è un polinomio di grado k ed esiste un polinomio di grado k+1 per cui l'errore è non nullo*

- Data una formula di Newton-Cotes sui nodi $x_i=a+ih$, con $h=(b-a)/n$, $i = 0,\ldots,n$ (chiuse)

$$\begin{cases} E = \dfrac{f^{n+2}(\xi)h^{n+3}}{(n+2)!}\displaystyle\int_0^n \prod_{j=0}^{n} t(t-j)dt \quad \text{per n pari e } f \in C^{n+2}[a,b] \\ \qquad\qquad \Longrightarrow \text{grado di precisione } n+1 \\ E = \dfrac{f^{n+1}(\xi)h^{n+2}}{(n+1)!}\displaystyle\int_0^n \prod_{j=0}^{n} (t-j)dt \quad \text{per n dispari e } f \in C^{n+1}[a,b] \\ \qquad\qquad \Longrightarrow \text{grado di precisione } n \end{cases}$$

- Trapezoidi (n=1,dispari) ha grado di precisione 1
- Simpson (n=2, pari) ha grado di precisione 3 : in generale sempre meglio formule con n pari !
- Formule di quadratura di Newton-Cotes con n+1 nodi hanno grado di pecisione almeno n (derivata si annulla nella formula dell'errore)

Formule di Newton-Cotes aperte : metodo Midpoint

Formula di quadratura con 1 nodo: prima formula *aperta* di Newton-Cotes, "*midpoint*"

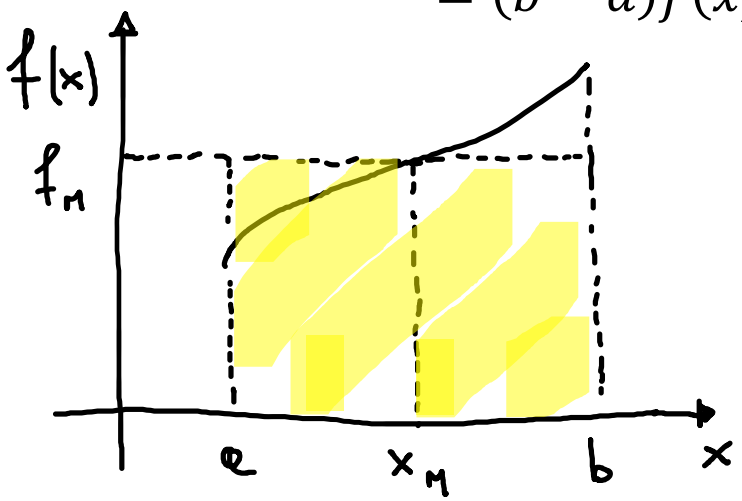
- Per ricavare l'espressione dell'errore relativa alla regola del punto medio, scriviamo la formula di Taylor con resto di Lagrange per f(x) attorno al punto medio $x_M = (a + b)/2$ fino all'ordine 1.

$$f(x) = f(x_M) + (x - x_M)f'(x_M) + \frac{(x - x_M)^2}{2} f''(\xi(x))$$

$$\int_a^b f(x)dx = \int_a^b \left[f(x_M) + (x - x_M)f'(x_M) + \frac{(x - x_M)^2}{2} f''(\xi(x)) \right] dx$$

$$= (b - a)f(x_M) + f'(x_M)\frac{(x - x_M)^2}{2}\Big|_b^a + f''(\xi)\frac{(x - x_M)^3}{2}\Big|_b^a =$$

$$= f(x_M)(b - a) + \frac{f''(\xi)(b - a)^3}{24}$$

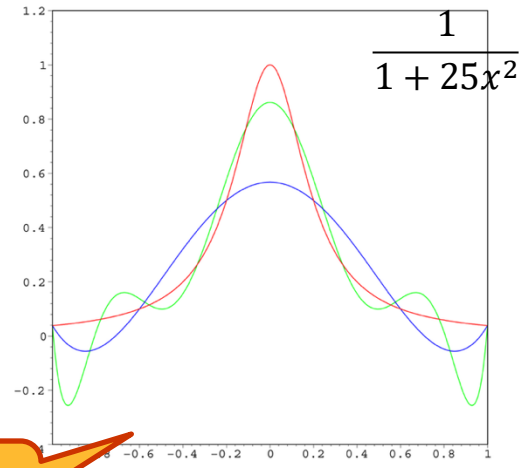


- Ho approssimato l'area sotto la curva con l'area del rettangolo di altezza $f(x_M)$
- Non si usano f(a) e f(b), potenzialmente utilizzabile anche con funzioni (integrabili) che divergono agli estremi

Migliorare la precisione delle formule di quadratura

Come possiamo cercare di migliorare la precision sulla stima dell'integrale utilizzando le formule di quadratura di Newton-Cotes ?

- Scegliere un polinomio di grado elevato : più alta è il grado del polinomio migliore è l'approssimazione (in teoria). Questo non è sempre vero almeno nel caso di nodi equispaziati (controesempio di Runge): in pratica non si eccede mai l'ordine 5-6. Si può ovviare al problema con una scelta opportune dei nodi (nodi di Chebycev)

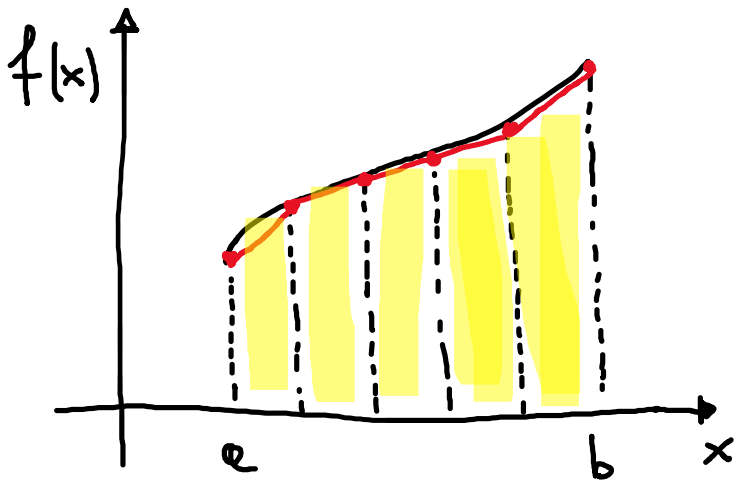


Blu : polinomio di 5° grado
Verde : polinomio di 9° grado

- Suddividere l'intervallo di integrazione in un gran numero di sotto-intervalli e utilizzare le formule di Newton-Cotes in ciascuno di essi : **formule di Newton-Cotes composte**. Importante studiare l'andamento dell'errore nei vari casi.

Formule di Newton-Cotes composte: metodo trapezoidi composto

Dividiamo l'intervallo di integrazione in N sotto-intervalli uguali e in ciascuno di essi applichiamo la formula dei trapezi che abbiamo ricavato in precedenza:



$$h = \frac{b-a}{N}$$

$$I_k = \frac{f(x_k) + f(x_{k+1})}{2}(x_{k+1} - x_k)$$

$$I = \sum_{k=0}^{N-1} I_k = \sum_{k=0}^{N-1} \frac{f(x_k) + f(x_{k+1})}{2} h$$

$$I = h\left(\frac{1}{2}f(x_0) + \frac{1}{2}f(x_1) + \frac{1}{2}f(x_1) + \cdots \frac{1}{2}f(x_{N-1}) + \frac{1}{2}f(x_{N-1}) + \frac{1}{2}f(x_N)\right)$$

$$= \left(\frac{1}{2}f(a) + \sum_{k=1}^{N-1} f(a+kh) + \frac{1}{2}f(b)\right) h$$

Molto facile da implementare in un codice numerico !

Formule di Newton-Cotes composte : errore per il metodo trapezoidi

Stimiamo l'errore che si commette integrando con il metodo dei trapezoidi composto: esplicitiamo l'errore che si commette in ogni singolo intervallo e sommiamolo

$$\int_a^b f(x)dx = \sum_{k=0}^{N-1} \left[\int_{x_k}^{x_{k+1}} \left(\frac{f(x_k) + f(x_{k+1})}{2} \right) h\, dx - \frac{h^3}{12} f''(\xi_k) \right]$$

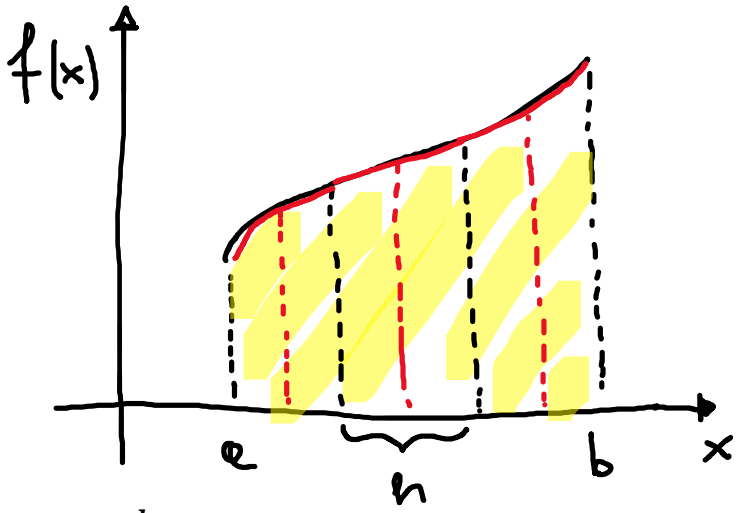
Teorema dei valori intermedi

$$R = -\sum_{k=0}^{N-1} \frac{h^3}{12} f''(\xi_k) = N \frac{h^3}{12} \overline{f''} = N \frac{h^3}{12} f''(\xi) = \frac{(b-a)}{h} \frac{h^3}{12} f''(\xi)$$

$$R = \frac{(b-a)}{12} f''(\xi) h^2$$

Formule di Newton-Cotes composte : metodo di Simpson

Proviamo a calcolare l'integrale componendo il metodo di Newton-Cotes con n=2



$$h = \frac{b-a}{N}$$

$$I_k = \left[f(x_k) + 4f\left(x_{M,k}\right) + f(x_{k+1}) \right] \frac{h}{3}$$

$$I = \sum_{k=0}^{N-1} I_k = \sum_{k=0}^{N-1} \left[f(x_k) + 4f\left(x_{M,k}\right) + f(x_{k+1}) \right] \frac{h}{3}$$

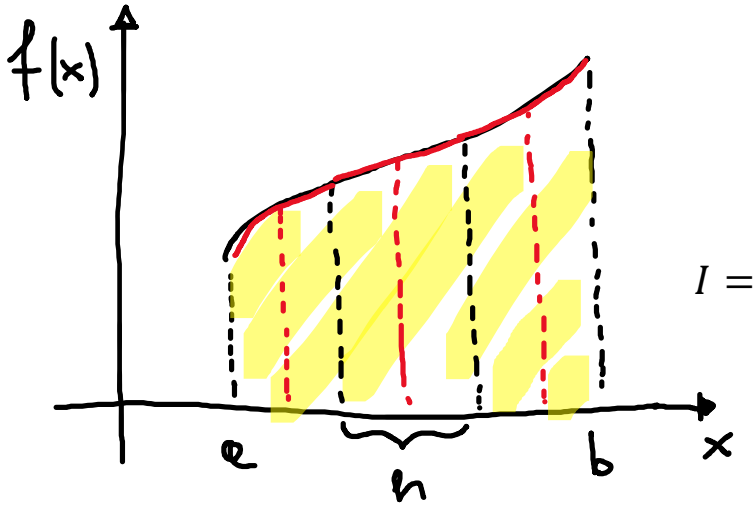
$$I = \frac{h}{3}\left(f(x_0) + 4f\left(x_{M,0}\right) + f(x_1) + f(x_1) + 4f\left(x_{M,1}\right) + f(x_2) + \cdots + f(x_{N-1}) + 4f\left(x_{M,N-1}\right) + f(x_N) \right)$$

$$= \frac{h}{3}\left(f(x_0) + 4f\left(x_{M,1}\right) + 2f(x_1) + 4f\left(x_{M,2}\right) + \cdots + 2f(x_{N-1}) + 4f\left(x_{M,N-1}\right) + f(x_N) \right)$$

$$I = \frac{h}{3}\left(f(x_0) + \left(\sum_{k=1}^{2N-1} c_k f\left(a + k\frac{h}{2} \right) \right) + f(x_N) \right) \quad dove\ c_k = \begin{cases} 4\ se\ k\ dispari \\ 2\ se\ k\ pari \end{cases}$$

Nota sulla formula di Simpson composta

- Nella formula di Simpson a N intervalli abbiamo dovuto calcolare la funzione in 2N punti (Simpson richiede la valutazione nel punto intermedio) :



Assicuriamoci sempre che N sia pari altrimenti la formula si rompe !

$$h = \frac{b-a}{N}$$

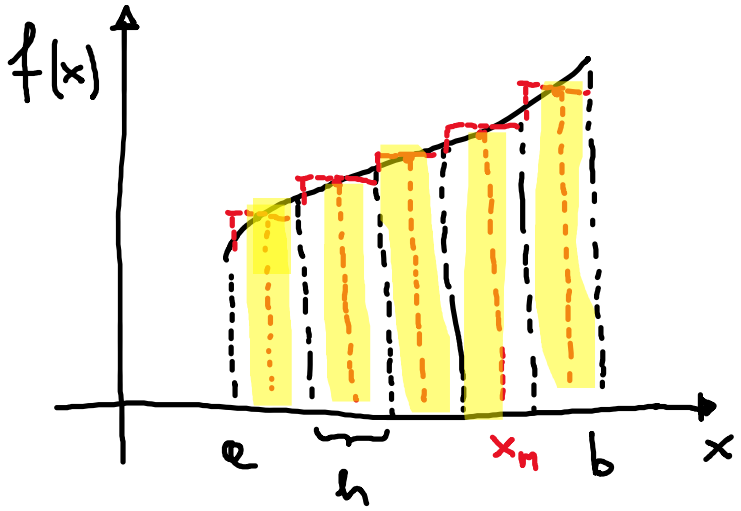
$$I = \frac{h}{3}\left(f(x_0) + \sum_{k=1}^{2N-1} c_k f\left(a + k\frac{h}{2}\right) + f(x_N)\right) \quad c_k = \begin{cases} 4 \; se \; k \; dispari \\ 2 \; se \; k \; pari \end{cases}$$

- questo non è molto onesto verso il metodo dei trapezoidi perché il peso computazionale dipende dal numero di operazioni che facciamo (numero di volte in cui valutiamo la funzione). Possiamo riformulare l'espressione per numero di valutazioni N fissato (quindi di fatto per un intervallo pari a 2h nella formula precedente):

$$I = \frac{h}{3}\left(f(x_0) + \sum_{k=1}^{N-1} c_k f(a + kh) + f(x_N)\right) \quad c_k = \begin{cases} 4 \; se \; k \; dispari \\ 2 \; se \; k \; pari \end{cases}$$

Formule di Newton-Cotes composte : metodo midpoint (formula aperta)

Dividiamo l'intervallo di integrazione in N sotto-intervalli uguali e in ciascuno di essi applichiamo il metodo del punto medio (Midpoint) :



$$h = \frac{b-a}{N}$$

$$I_k = f\left(x_{M,k}\right)\left(x_{k+1} - x_k\right) = f\left(x_{M,k}\right)h$$

$$I = \sum_{k=0}^{N-1} I_k = \sum_{k=0}^{N-1} f\left(x_M^k\right)h$$

$$\boxed{I = \sum_{k=0}^{N-1} f\left(a + \left(k + \frac{1}{2}\right)h\right)h}$$

Formula triviale da implementare : si sceglie un passo h (o equivalentemente un numero di intervalli N), si valuta la funzione nei punti medi, si somma e si moltiplica il tutto per h

Summary sulle formule di quadratura di Newton-Cotes

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

Migliorare la precisione del calcolo : estrapolazione di Richardson

Proviamo a combinare valutazioni dell'integrale con passi diversi per cancellare ordini superiori dell'errore

Trapezoidi

$$\begin{cases} I_h - I = k_1 h^2 + k_2 h^4 + \cdots \\ I_{h/2} - I = k_1 \left(\frac{h}{2}\right)^2 + k_2 \left(\frac{h}{2}\right)^4 + \cdots \end{cases} \quad \begin{cases} I_h - I = k_1 h^2 + k_2 h^4 + \cdots \\ 4I_{h/2} - 4I = k_1 h^2 + 4\frac{1}{16} k_2 h^4 + \cdots \end{cases}$$

$$I_h - 4I_{h/2} + 3I = \frac{3}{4} k_2 h^4 + \cdots \Rightarrow \boxed{I = \frac{1}{3}\left(4I_{h/2} - I_h\right) + o(h^4)}$$

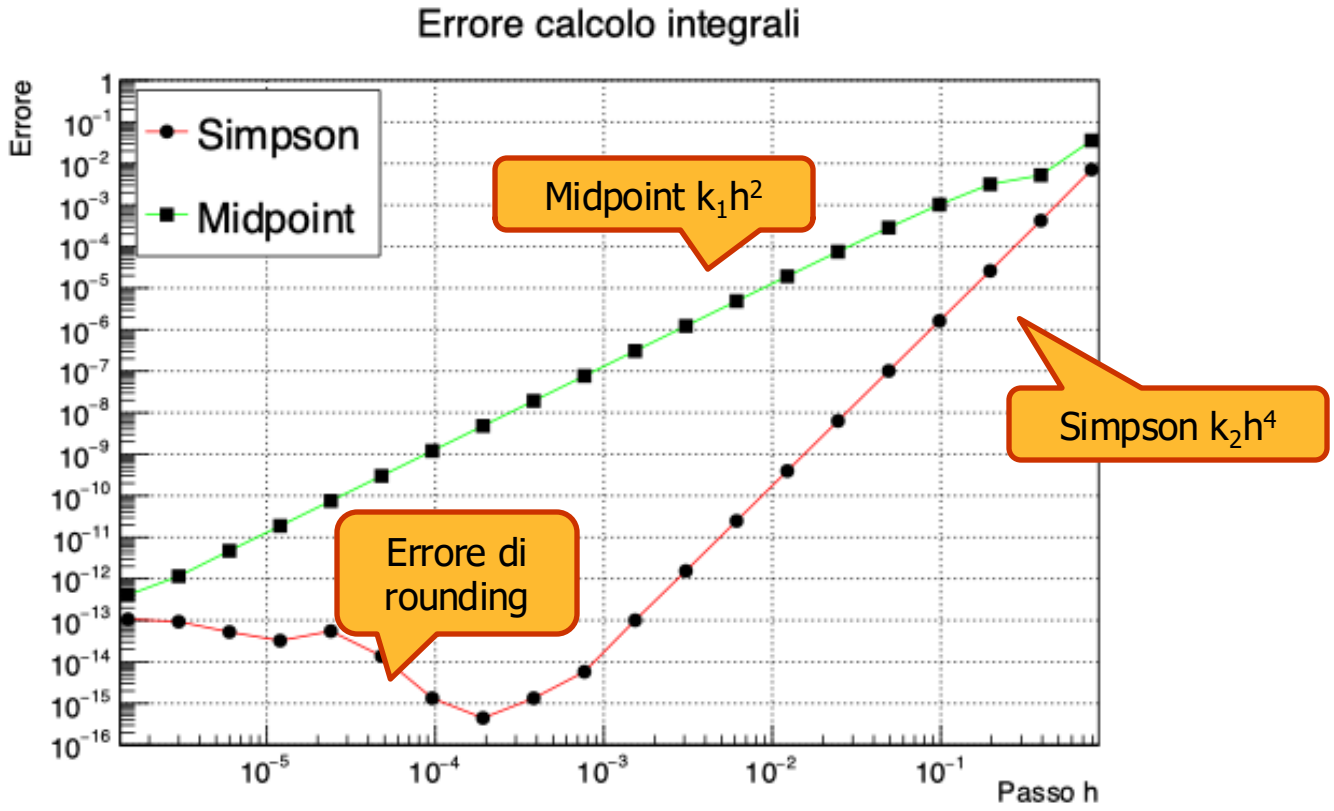
Simpson

$$\begin{cases} I_h - I = k_1 h^4 + k_2 h^6 + \cdots \\ I_{h/2} - I = k_1 \left(\frac{h}{2}\right)^4 + k_2 \left(\frac{h}{2}\right)^6 + \cdots \end{cases} \quad \begin{cases} I_h - I = k_1 h^4 + k_2 h^6 + \cdots \\ 16\, I_{h/2} - 16I = k_1 h^2 + 16\frac{1}{64} k_2 h^4 + \cdots \end{cases}$$

$$I_h - 16I_{h/2} + 15I = \frac{3}{4} k_2 h^6 + \cdots \Rightarrow \boxed{I = \frac{1}{15}\left(16I_{h/2} - I_h\right) + o(h^6)}$$

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 ($\int_0^{\pi/2} \mathrm{x}\,\sin(x)\,dx$)



- Nei casi tipici in cui l'integrale NON è NOTO si può solo stimare l'errore con tecniche run-time (vedi slides successive)

Quanti passi di integrazione ?

- Integrare a passo fissato (cioè decidendo a priori il numero di intervalli N o la dimensione dell'intervallo di integrazione h=(b-a)/N) non è particolarmente utile: se calcolo un integrale con N=10 intervalli usando il metodo dei trapezoidi cosa posso dire sull'errore che sto commettendo ? È grande, piccolo, accettabile? Come lo stimo?

- Normalmente sarebbe molto più utile lavorare con un algoritmo a precisione fissata : decide lui (iterativamente) quanti intervalli utilizzare per raggiungere la precisione desiderata (**stima runtime dell'errore**).

Stima runtime dell'errore: caso del metodo dei trapezoidi composto

- È possible avere una stima runtime dell'errore che stiamo commettendo integrando con un metodo di quadratura di quelli indicate ? Riutilizziamo il trucchetto usato per l'estrapolazione di Richardson

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

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

- È possible stimare quale passo h è necessario per ottenere una stima dell'integrale con un'incertezza fissata ?

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

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

Algoritmo a precisione fissata

In generale però vorremmo avere un algoritmo che funzioni a precisione fissata non a numero di punti (o passo) fissato.

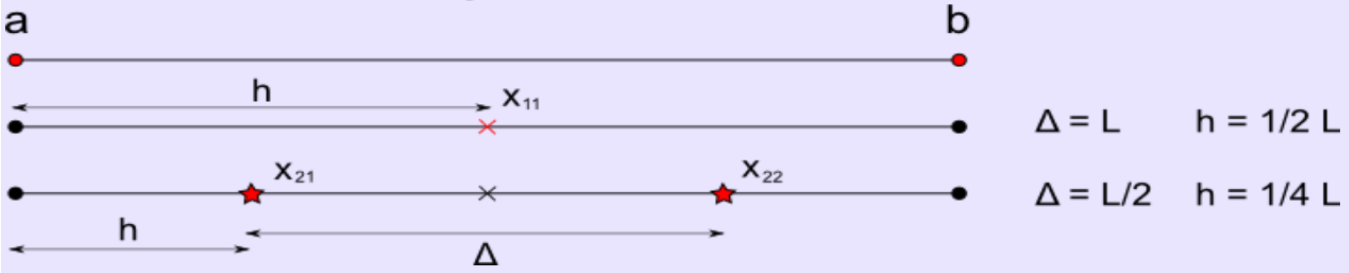
- Scrivere un algoritmo che stimi l'errore ad ogni passo (vedi slide "stima runtime dell'errore") e raddoppi il numero di intervalli ad ogni passaggio finchè non si raggiunge la precisione richiesta
- Il metodo dei trapezoidi si presta ad una versione ottimizzata di questo approccio : ad ogni passaggio 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
```

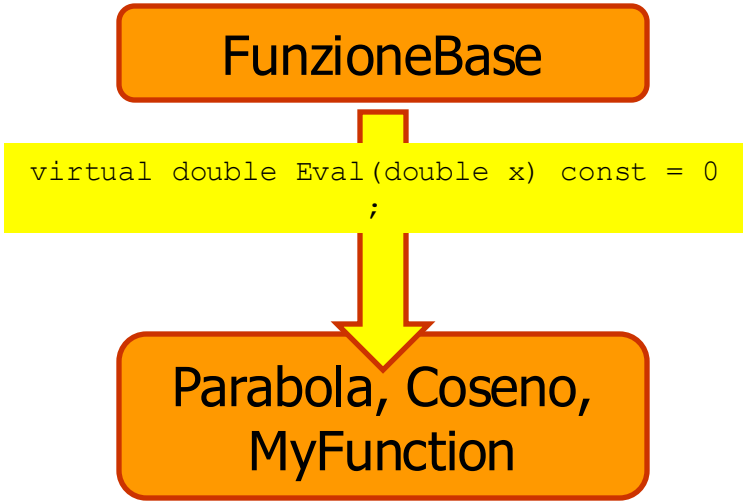
e così secondo lo schema in figura

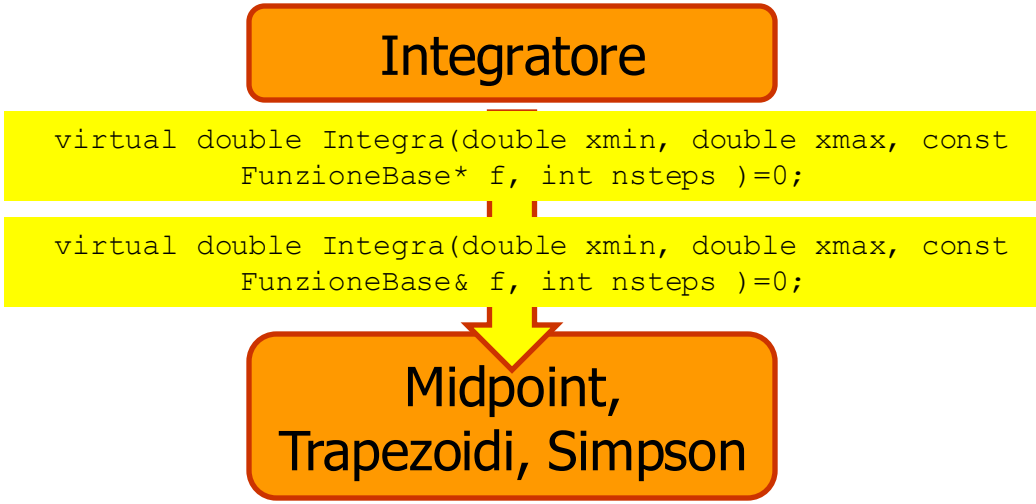



```
sum2 = sum1 + f[x21] + f[x22] ; I2 = sum2 * (b-a)/4

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

Vedi lezione7 dal sito di labTNDS

Possibile struttura delle classi : caso più generalizzabile






```
virtual double Eval(double x) const override {
                return … ;};
```

```
virtual double Integra (double xmin, double xmax, const
     FunzioneBase* f, int nteps ) override {… };
```

```
virtual double Integra (double xmin, double xmax, const
     FunzioneBase& f, int nteps ) override {… };
```

Possibile struttura delle classi : caso più generalizzabile

```
#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(a

  Seno mysin ;

  Trapezoidi integrator_t (a,b,mysin);
  cout << integrator_t.Integra(ntep) << endl;

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

  // some code here ...

  MyComplexCalculator calc;
  double muyoutput = calc.ComplicatedFunction(double k1,
                                              double k2,
                                              integrator_s ,
                                              mysin ) ;


  // some more code here ....

  return 0;

}
```

funzione

Integratori

Quale vantaggio a scrivere la classe `Midpoint` come derivata di `Integratore` ?

- Immaginate un algoritmo complesso che, tra le altre cose, debba svolgere degli integrali (`ComplicatedFunction`)
- 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 ) {...};
```

Questa funzione lavora con un generico `Integratore`, il metodo `Integra` che verrà invocato dipende dal puntatore passato in input !

Implementazione alternativa : puntatori a funzione, funtori e lambda

Function pointer, not possible to pass external parameters

```
// 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 speficy 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 ;
```

A functor is basically a class, and we use the operator() to mimic a function. Parameters can be passed through constructors of SetParams method

Implementazione alternativa : puntatori a funzione, funtori e lambda

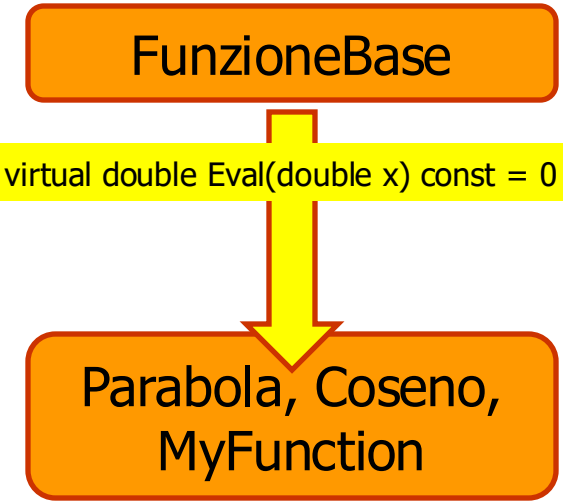
```
// this is simple, but works only with function pointers

double MidpointSimple ( 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 MidpointTemplate ( 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 MidpointFunction ( 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 ;

}
```

Implementazione alternativa : puntatori a funzione, funtori e lambda

```
int main() {

  // MidpointSimple only works with function pointers

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

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

  mysinfunctor m1 ;

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

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

  double para = 2;
  mysinfunctor m2(para) ;

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

}
```

Backup

Possibile struttura delle classi : caso semplice



virtual double Eval(double x) const override { return … ;};

Integratore

```
double Midpoint(double xmin, double xmax, const FunzioneBase* f, int nteps );
double Trapezoidi(double xmin, double xmax, const FunzioneBase* f, int nteps );
double Simpson(double xmin, double xmax, const FunzioneBase* f, int nsteps);
```

```
double Midpoint(double xmin, double xmax, const FunzioneBase& f, int nteps );
double Trapezoidi(double xmin, double xmax, const FunzioneBase& f, int nteps );
double Simpson(double xmin, double xmax, const FunzioneBase& f, int nsteps);
```

Calcolo integrale definito di una funzione

Using pointers

```
#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 = new Seno();
  Integral * integrator = new Integral(a,b,mysin);

  cout << integrator->Midpoint(nstep) << endl;
  cout << integrator->Simpson(nstep) << endl;

  return 0;

}
```

funzione

Integratore

Calcolo degli integrali

Using references

```
#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 ;
  Integral integrator(a,b,mysin);

  cout << integrator.Midpoint(nstep) << endl;
  cout << integrator.Simpson(nstep) << endl;

  return 0;

}
```

funzione

Integratore

Calcolo degli integrali

Derivazione errore metodo trapezoidi

Infatti:

$$E_T = \int_a^b R_1(x)\,dx = \int_{x_0}^{x_1} \frac{(x-x_0)(x-x_1)}{2!} f''(\xi_x)\,dx =$$

$$(x = x_0 + sh, \quad dx = hds) = \int_0^1 \frac{h^3 s(s-1)}{2!} f''(\xi_s) ds =$$

(in virtù del teorema[2] della media integrale, essendo $s(s-1) \leq 0$ in I, con $\xi \in]a,b[$)

$$= h^3 f''(\xi) \int_0^1 \frac{s(s-1)}{2} ds = -\frac{h^3}{12} f''(\xi) = O(h^3).$$

[2]Siano $f(x)$ e $g(x)$ continue in $[a,b]$ con $g(x)$ ivi di segno costante; allora esiste un punto $c \in [a,b]$ tale che $\int_a^b f(x)g(x)\,dx = f(c)\int_a^b g(x)\,dx$ (teorema di Bonnet).