17/03/2010

Introduzione alla Programmazione

Corso di Sperimentazioni di Fisica

Ingegneria Energetica

Università degli Studi di Pisa

Giulia De Bonis
giulia.debonis@pi.infn.it
www.pi.infn.it/~debonis

Edificio C – stanza 72
Polo Fibonacci, Largo B. Pontecorvo 3, Pisa

sono il computer, risolvo problemi

- Schematizzare il problema
  → definire l'algoritmo di calcolo
- Tradurre l'algoritmo nel linguaggio di programmazione scelto → C++
  → scrivere il codice (codice sorgente)
- Compilare il codice
  → produrre un eseguibile (programma)
- Eseguire il programma

Ogni linguaggio di programmazione ha le sue **regole** (parole riservate, istruzioni, operatori, sintassi).
Il mancato rispetto delle regole produce **errori di compilazione**.

Il **compilatore** è un "traduttore", cioè un programma che traduce le istruzioni del codice da un linguaggio ad alto livello (il programmatore) ad uno di livello più basso (il calcolatore).

Metodo di Newton
(o metodo delle tangenti)

Il metodo di Newton permette di trovare, per via numerica, la soluzione dell'equazione $f(x) = 0$

Si tratta di un metodo iterativo, che cioè calcola la soluzione ripetendo (*iterando*) il procedimento, fino a raggiungere il grado di approssimazione desiderato.

Metodo di Newton

L'algoritmo di calcolo consiste nell'approssimare la curva di funzione $y = f(x)$ con la retta tangente alla curva in un punto generico $\left(x_0, y_0 = f(x_0)\right)$. Questa retta ha equazione:

$$y - f(x_0) = f'(x_0) \cdot (x - x_0)$$

[Ricordiamo che $y - y_0 = m \cdot (x - x_0)$ è l'equazione di una retta di coefficiente angolare *m* passante per il punto (x_0, y_0), e che il coefficiente angolare della retta tangente è pari alla derivata prima della funzione]

La retta tangente intercetta l'asse delle *x* nel punto $(x = x_1, y = 0) \Rightarrow x_1 = x_0 - \dfrac{f(x_0)}{f'(x_0)}$

A questo punto, si ripete il procedimento, considerando la retta tangente alla curva nel punto $\left(x_1, y_1 = f(x_1)\right)$ e calcolando nuovamente l'intercetta della retta con l'asse delle x

$\Rightarrow x_2 = x_1 - \dfrac{f(x_1)}{f'(x_1)}$

Si ottiene, cioè, la relazione di ricorrenza

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Il processo di iterazione si interrompe quando $x_{n+1}=x_n$ entro la precisione richiesta dal calcolo. Il valore trovato è la soluzione dell'equazione cercata $f(x) = 0$

Metodo Iterativo → ciclo

I metodi iterativi, come il metodo di Newton, sono particolarmente indicati per essere implementati in un programma, attraverso l'impiego di un **CICLO**.

Un CICLO è la ripetizione di una sequenza di istruzioni. Il numero di ripetizioni è controllato da un indice, che viene incrementato ad ogni ciclo; la ripetizione del ciclo è condizionata dal verificarsi di una condizione di test.

Esempio - **ciclo for**

```
for(indice_valore-iniziale, condizione_di_test, incremento)
{
    ...
    <istruzioni da eseguire all'interno del ciclo>
    ...
}
```

Applicazione del metodo di Newton al calcolo della radice quadrata dei numeri reali

(sottoinsieme delle radici positive)

Supponiamo di voler calcolare per via numerica la radice quadrata del numero *z*.

Si tratta di risolvere l'equazione $f(x) = x^2 - z = 0, \quad f'(x) = 2 \cdot x$

La relazione di ricorrenza diventa, quindi:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{x_n^2 - z}{2 \cdot x_n} \Rightarrow \boxed{x_{n+1} = \frac{1}{2}\left(x_n + \frac{z}{x_n}\right)}$$

soluzione del problema

codice sorgente

radice-quadrata.cpp

- Creare un nuovo file, denominarlo “radice-quadrata.cpp”.
  →l’estensione “.cpp” è propria del C++

Il codice può essere modificato con un qualsiasi editor di testo.

È buona norma commentare il codice, in modo da includere informazioni aggiuntive e aumentare la leggibilità:

```
// Autore: Giulia De Bonis
// Data: 10/03/2010

/* radice-quadrata.cpp calcola la radice quadrata di un numero
reale, applicando il metodo di approssimazione di Newton */
```

radice-quadrata.cpp

la funzione main

Ogni programma in C++ deve contenere una funzione **main**

```
int main(int argc, char *argv[]){
  ...
  <istruzioni del programma>
  ...
 return 0;
}
```

Tutte le istruzioni del programma devono essere racchiuse all'interno della coppia di *parentesi graffe* ***{ }*** che definiscono il blocco main.

Tutte le istruzioni del C++ devono terminare con un segno di *punto e virgola* ***;***

… prima del blocco main

direttive al preprocessore

Il preprocessore è un programma che "entra in azione" in fase di compilazione del codice, subito prima del compilatore. La sua funzione è quella di interpretare le direttive al preprocessore, traducendole in un linguaggio comprensibile al compilatore.
Le direttive iniziano sempre con il carattere *cancelletto* **#**. Non essendo istruzioni del programma, non terminano con il punto e virgola.

Direttive di inclusione (inclusione di *header* file)

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

Le direttive di inclusione vengono usate per inserire le **librerie standard** del C++.
In questo esempio:
- iostream → funzioni di input/output
  (inserimento parametri, visualizzazione su schermo)
- iomanip → manipolazioni del formato di input/output

… prima del blocco main

using namespace std

Tutti gli elementi della **libreria standar**d del C++ sono dichiarati all'interno del *namespace std*. Per utilizzare nel codice gli elementi della libreria standard dobbiamo dichiarare l'utilizzo del namespace:

```
using namespace std;
```

dichiarazioni

Tutte le variabili che si intende usare nel programma devono essere dichiarate. La dichiarazione consiste nello specificare, per ogni variabile, un **nome** e un **tipo**. Il nome permette di identificare univocamente la variabile; il tipo permette di classificare le variabili, in modo che sia possibile assegnare lo spazio di memoria opportuno per la memorizzazione.

N.B. Il C++ è case-sensitive, cioè distingue tra lettere maiuscole e minuscole

```
int n,N;
double z,x;
```

È ammessa l'assegnazione contestualmente alla dichiarazione:

```
int n,N=10;
double z=2,x;
```

algoritmo di Newton con il ciclo for

```
Z=2.;
x=1.;
N=10;
for(n=1; n<=N; n++) {
    x=0.5*(x+z/x);
}
```

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{z}{x_n}\right)$$

Il numero di ripetizioni è controllato dal valore della variabile intera n.
La variabile n viene inizializzata (n=1); viene specificata una condizione di controllo (n<=N); la variabile n è incrementata a ogni ciclo.

output: cout <<

In C++ le operazioni di I/O sono fatte operando su “stream” (sequenze) di bytes (libreria <iostream>).

cout = standard output → lo **schermo**

<< = operatore di inserimento

```
Z=2.;
x=1.;
N=10;
for(n=1; n<=N; n++) {
    x=0.5*(x+z/x);
    cout<<"iterazione n="<<n<<"\t\t"<<setprecision(10)<<x<<endl;
  }
```

(libreria <iomanip>)
imposta il numero di cifre con cui viene visualizzato il risultato

stringa (tra virgolette)

tab

leggibilità dell’output

end-line

Il valore che indica la precisione della rappresentazione può essere parametrizzato:

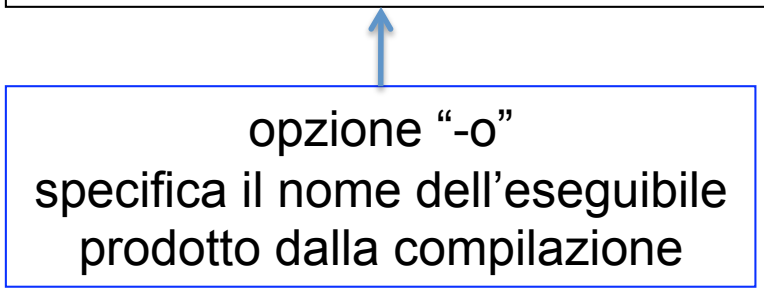
```
int k=10;
...
...
cout<<"iterazione n="<<n<<"\t\t"<<setprecision(k)<<x<<endl;
```

compilazione & esecuzione

Da riga di comando:

- compilazione

```
g++ -o radice-quadrata radice-quadrata.cpp
```



- esecuzione

```
./radice-quadrata
```

input: cin >>

L'assegnazione del valore dei parametri (ad esempio, la scelta del numero z del quale si vuole calcolare la radice quadrata) è un'istruzione del codice. Se si vuole modificare il valore parametri, occorre modificare il codice e ricompilare.

Un'alternativa per rendere il programma più versatile è l'**inserimento dei parametri da riga di comando** in fase di esecuzione utilizzando ((libreria <iostream>):

cin = standard input → la **tastiera**

>> = operatore di estrazione

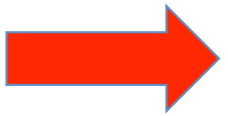
Una variabile di tipo **bool** può assumere solo due possibili valori: **true** (vero)/**false** (falso)

Un consiglio: definire la variabile booleana "user" per abilitare la possibilità che sia l'utente a definire i parametri del calcolo utilizzando lo standard input. Possiamo distinguere due casi:

- user = true → i parametri sono immessi dall'utente;
- user = false → vengono utilizzati i *valori di deafult* per i parametri

Le due modalità possono essere controllate facendo uso di una

struttura if-else



cin >> + if-else

```
bool user=true;
if(user){
    cout<<"Inserisci un numero: ";
    cin>>z;
    cout<<"z = "<< z <<endl;
    cout<<"Inserisci il valore di partenza dell'approssimazione: ";
    cin>>x;
    cout<<"x0 = "<< x <<endl;
    cout<<"Inserisci il numero di iterazioni: ";
    cin>>N;
    cout<<"N = "<< N <<endl;
    cout<<"Specifica la precisione (numero di cifre con cui rappr. il ris.): ";
    cin>>k;
    cout<<"k = "<< k <<endl;
  }
  else{
    z=2;
    cout<<"z = "<< z <<endl;
    x=1.;
    cout<<"x0 = "<< x <<endl;
    N=10;
    cout<<”N = "<< N <<endl;
    k=10;
    cout<<"k = "<< k <<endl;
  }
```

ciclo do-while

Quante iterazioni sono necessarie per ottenere il risultato con il livello di approssimazione richiesto (setprecision)?

Il ciclo *for* esegue N iterazioni, con N fissato. Può accadere che N sia troppo piccolo, cioè che dopo N iterazioni il risultato del calcolo non abbia ancora raggiunto la precisione richiesta; oppure, viceversa, che N sia troppo grande e l'esecuzione "sprechi tempo" nel ciclo, ripetuto senza migliorare il risultato del calcolo.

Il codice può essere modificato sostituendo il ciclo *for* con un ciclo do-while

```
int n=0,k=10;
double z=2.,x=1.,x0;
do {
    n=n+1;
    x0=x;
    x=0.5*(x0+z/x0);
    cout<<"iterazione n="<<n<<"\t\t"<<x<<endl;
  } while (x!=x0);
```


ATTENZIONE ai loop infiniti ! ! !


N.B. L'esecuzione del ciclo avviene almeno una volta (cfr. ciclo while)

opzioni avanzate

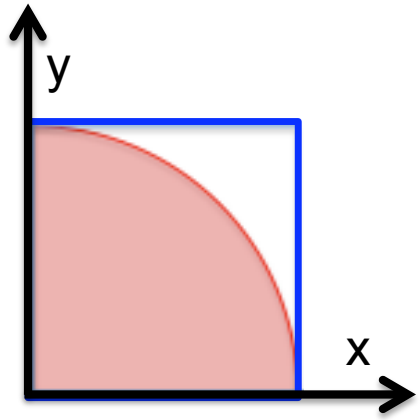
ostringstream

→ manipolare stringhe come se fossero “stream”

```
#include <sstream>
...
std::ostringstream ics,ics_zero;
...

do {
    ics.clear();
    ics.seekp(0, std::ios::beg);
    ics_zero.clear();
    ics_zero.seekp(0, std::ios::beg);

    // Algoritmo di Newton
    n=n+1;
    x0=x;
    x=0.5*(x0+z/x0);
    cout<<"iterazione n="<<n<<"\t\t"<<setprecision(k)<<"x = "<<x<<endl;

    ics<<setprecision(k)<<x;
    ics_zero<<setprecision(k)<<x0;
} while(ics.str()!=ics_zero.str());
...
```

Stima del valore di π=3.1415926535…

π è un numero trascendente, il cui valore di può essere stimato considerando l'area di un settore di circonferenza inscritto in un quadrante di lato r=1.



Il rapporto tra le aree è proporzionale a π →

$$\frac{\text{area del settore circolare}}{\text{area del quadrante}} = \frac{\frac{1}{4}\cdot\left(\pi\cdot r^2\right)}{r^2} = \frac{\pi}{4}$$

Come stimare il rapporto tra le aree? → rapporto di "punti"

Algoritmo:

- estrarre una coppia di numeri casuali (tra 0. e 1.) corrispondente alle coordinate di un punto P nel quadrante positivo del piano *xy*
- verificare se il punto P è all'interno o all'esterno del settore circolare
- il rapporto tra le aree può essere stimato dal rapporto tra il numero di punti estratti all'interno del settore circolare e il numero totale di estrazioni

$$\pi \approx 4\cdot\frac{N_{IN}}{N_{TOT}}$$

Generazione di numeri (pseudo)casuali

Il generatore rand() produce sequenze di **numeri interi pseudo-casuali** distribuiti uniformemente nell'intervallo che va da 0 a RAND_MAX (valore definito nella libreria standard).

Il prototipo della funzione è:

```
int rand ( void );
```

cioè, la funzione non richiede parametri (void) e restituisce un intero.

RAND_MAX dipende dal sistema; per verificare il valore:

```
cout<<"(RAND_MAX = "<<RAND_MAX<<")"<<endl;
```

Il generatore di numeri (pseudo)random deve essere inizializzato assegnando un seme (*seed*) attraverso la funzione srand [`void srand ( unsigned int seed );`]

Solitamente, il seme è assegnato utilizzando il valore della funzione time (definita in **<ctime>**), che varia ogni secondo.

```
#include <ctime>
...
srand((unsigned)time(NULL);
```

rand()

generazione di numeri reali → type casting

Il generatore rand() produce **numeri interi** nell'intervallo (0, RAND_MAX). Per generare **numeri reali tra 0 e 1** è necessario utilizzare l'operatore di *type casting* (conversione di tipo) e dividere per l'ampiezza dell'intervallo di generazione:

```
double x;
x=((double)rand()/(double)RAND_MAX);
```

La funzione rand() restituisce un intero

N.B. se non si esplicita la conversione di tipo, il rapporto tra l'uscita di rand() e il valore RAND_MAX restituisce sempre il valore 0 (= parte intera del risultato).

In generale:

```
int a=2;
double b;
b=(double)a;
b=double(a);
```

L'operazione di *type casting* restituisce il valore b=2.0

le due notazioni sono equivalenti

stima-pigreco.cpp

Algoritmo:

- definire il numero di estrazioni **N** → numero di ripetizioni di un **ciclo for**
- **ciclo for**
  - estrarre una coppia di numeri casuali (tra 0. e 1.) → **rand()**
  - **verificare che il punto P sia all'interno del settore circolare** → **if**
    → incrementare il contatore m
- calcolare π come rapporto m/n

stima-pigreco.cpp

Algoritmo:

- definire il numero di estrazioni **N** → numero di ripetizioni di un **ciclo for**
- **ciclo for**
  - estrarre una coppia di numeri casuali (tra 0. e 1.) → **rand()**
  - **verificare che il punto P sia all'interno del settore circolare** → **if**
    → incrementare il contatore m
- calcolare π come rapporto m/n

```
#include <ctime>
...
  int n,N=10000000,m=0;
  double x,y,pi;

 cout<<" --- Stima del valore di pigreco con N="<<N<< " estrazioni ---"<<endl;
 srand((unsigned)time(NULL));

 for(n=1;n<=N;n++){
    x=((double)rand()/(double)RAND_MAX);
    y=((double)rand()/(double)RAND_MAX);
    if(x*x+y*y<=1.) m=m+1;
  }
  pi=4.*((double)m/(double)n);
  cout<<"pigreco = "<<pi<<endl;
```

Il punto x,y è all'interno della circonferenza di raggio r=1.

stima di π come rapporto di aree

Il metodo del rapporto tra aree (rapporto tra punti) fornisce una stima abbastanza grossolana del valore di π. L'algortimo produce il valore esatto entro poche cifre decimali: occorrono ordine N=1e7 estrazioni per ottenere le prime 2 cifre decimali corrette (π=3.14…).

→ esercizio: modificare il codice sorgente, introducendo la possibilità di inserire il parametro N (numero di estrazioni) come **input da tastiera.**

L'equazione analitica della circonferenza $(x^2 + y^2 = r^2)$ definisce la condizione da verificare per l'incremento del contatore m. È possibile, dopo aver incluso <*math.h*>, utilizzare la funzione elevamento a potenza:

```
double pow ( double base, double exponent );
```

L'algortimo descritto per la stima di π può essere modificato, considerando il volume di una sfera (o di una porzione di sfera).

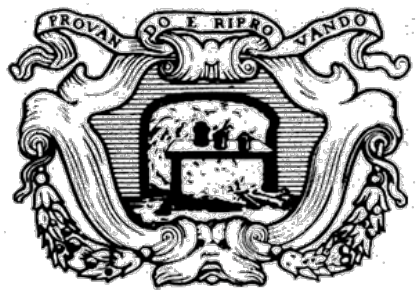
→esercizio: scrivere un codice per la

stima di π come **rapporto di volumi**

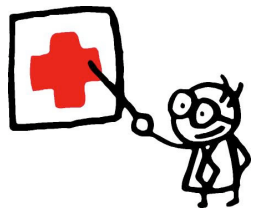
C++
risorse in rete

La miglior difesa contro le “insidie” della programmazione è l’attacco…

… provando e riprovando…



Consiglio: per un “pronto soccorso”, consultare le risorse disponibili in rete → guide, corsi, tutorial, forum… Alcuni esempi:



- Le Guide di HTML.it (in italiano) – http://programmazione.html.it/guide/leggi/34/guida-c/
- The C++ Resources Network (in inglese) – http://www.cplusplus.com/

Per l'utilizzo di ROOT
consultare la **User's Guide**

http://root.cern.ch/

ROOT

ROOT è un tool di analisi sviluppato al CERN e basato su C++.
Interpreta **macro** scritte in C++, introducendo nuove classi, con la possibilità di **rappresentare graficamente i risultati**.

ROOT sfrutta largamente le potenzialità del C++, in particolare la programmazione orientata agli oggetti (*object-oriented*).
Il concetto alla base della programmazione ad oggetti è quello di **classe**. Una classe rappresenta un tipo di dati astratto (cioè un'estensione dei tipi "fondamentali" come `int`, `double`, etc…) contenente elementi in stretta relazione tra loro, che condividono gli stessi attributi e le stesse "azioni" (funzioni membro o **metodi** della classe). Un oggetto è un'**istanza** di una classe. La principale caratteristica della programmazione a oggetti è la **modularità del codice**, che introduce il vantaggio di una maggiore facilità di manutenzione.

Digitare `root` da riga di comando per avviare la sessione di ROOT.
Tutti i comandi di ROOT sono preceduti da un punto (`.`) - esempi:

- `.q` → per uscire dalla sessione corrente di ROOT
- `.?` → per la lista dei comandi
- `.x nome file.C` → per eseguire un programma scritto precedentemente (macro)

stima di π in ROOT

stima-pigreco.C

Modificare il codice stima-pigreco.cpp per renderlo eseguibile in ROOT con il comando:

```
.x stima-pigreco.C
```

CONVENZIONI di ROOT

- le classi iniziano con T → TTree, TBrowser, ...
- i tipi finiscono con _t (*machine idependent types*) → Int_t, Double_t, ...
- le costanti iniziano con k → kTRUE, kRed, ...
- le funzioni membro iniziano con la lettera maiuscola → Draw(), Fill()), ...
- le variabili globali iniziano con g → esempi:
  - *gROOT*: puntatore all'oggetto TROOT della sessione corrente; permette di modificare le proprietà comuni a tutta la sessione (es. stile grafico)
  - *gRandom*: puntatore al **generatore di numeri casuali**; di default punta ad un oggetto TRandom che genera numeri a partire da una distribuzione uniforme.

Una **puntatore** è una variabile che contiene l'indirizzo di memoria di un'altra variabile. I puntatori sono una caratteristica del C++.

stima-pigreco.C

creazione di una *canvas*

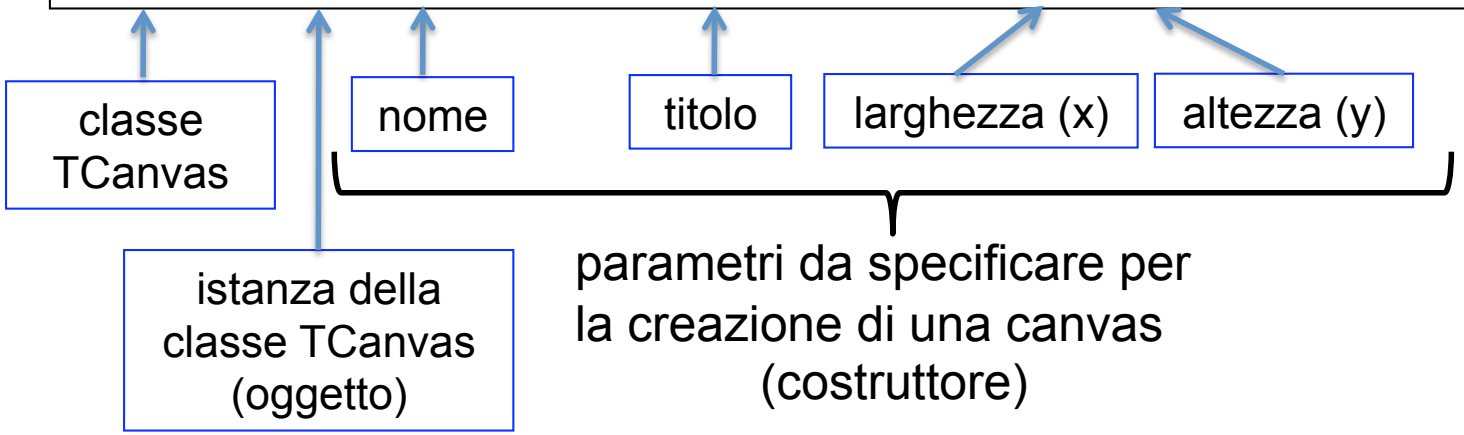
- INIZIO – parentesi graffe, variabili globali, dichiarazioni/assegnazioni:

```
{
  gROOT->Reset();
  gROOT->SetStyle("Plain");
  Int_t n,N,m=0;
  Double_t x,y,pi;
...
}
```

(machine independent types)

- Creare una "tela" (*canvas*), cioè il pannello dove rappresentare i risultati:

```
TCanvas c1("c1","Stima di pigreco",500,500);
```

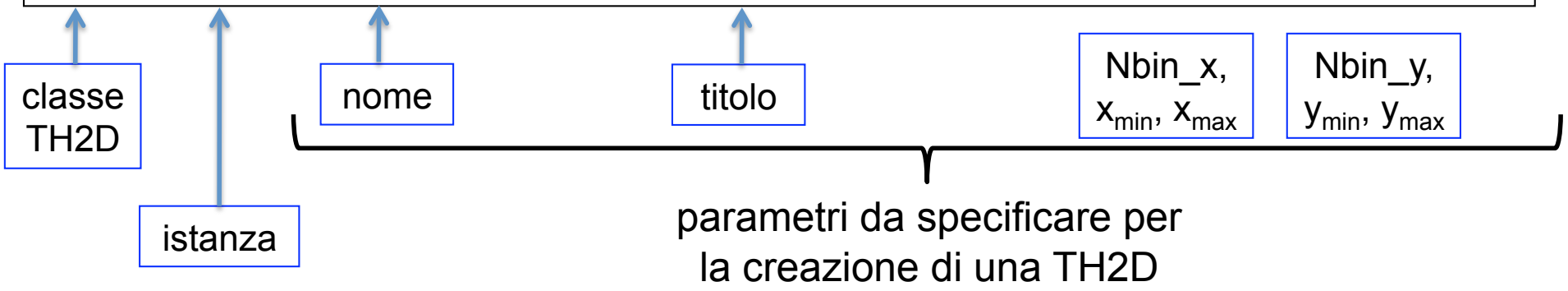


stima-pigreco.C

istogramma bidimensionale

• Creare un **istogramma** a due dimensioni, per visualizzare la distribuzione dei punti (x,y) generati casualmente nel primo quadrante

```
TH2D histo("histo","Estraz. di numeri casuali",50,0.,1.,50,0.,1.);
```



Classe TH2D:

H → histogramma

2 → 2 dimensioni

D→ "double" – gli istogrammi di questa classe possono contenere dati di tipo *double*

stima-pigreco.C

generazione di numeri casuali
riempimento dell'istogramma

• Per la generazione dei numeri casuali si utilizzano i metodi *SetSeed* e *Rndm* dell'oggetto gRandom (classe TRandom = classe dei generatori di numeri random):

```
 gRandom->SetSeed(0);


for(n=1;n<=N;n++){
    x=gRandom->Rndm();
    y=gRandom->Rndm();

    if(x*x+y*y<=1.) m=m+1;
    histo->Fill(x,y);
  }


  pi=4.*((double)m/(double)n);
  cout<<"pigreco = "<<pi<<endl;
```

inizializzazione del seme del generatore gRandom; seed=0 → il seme è assegnato utilizzando il valore corrente dell'orologio.

Rndm = generatore di numeri reali random distribuiti uniformemente tra 0. e 1.

metodo ***Fill()*** della classe **TH2D**. Ad ogni ciclo, l'istogramma è riempito con la coppia di valori (x,y) ("*entry*" dell'istogramma).

stima-pigreco.C
visualizzazione (1)

• Gli oggetti sono disegnati sulla *canvas* utilizzando il metodo **Draw()**.

I metodi "Set" consentono di impostare i parametri (colore, riempimento, dimensioni,…).

→ Visualizzazione dell'istogramma (distribuzione di numeri casuali):

```
histo.GetXaxis().SetTitle("x");
histo.GetYaxis().SetTitle("y");
histo->Draw();
```

→ Creazione e visualizzazione del settore circolare (puntatore):

```
TEllipse *settore_circolare = new TEllipse(0.,0.,1.,1.,0.,90.,0.);
settore_circolare->SetLineColor(kRed);
settore_circolare->SetLineWidth(2);
settore_circolare->SetFillColor(kRed-10);
settore_circolare->SetFillStyle(3001);
settore_circolare->Draw("same");
```

x_0, y_0 [coordinate del centro], r_1, r_2 [assi], φ_{min}, φ_{max}, θ

opzione "same" → sovrapporre l'oggetto a quanto già disegnato

stima-pigreco.C
visualizzazione (2)

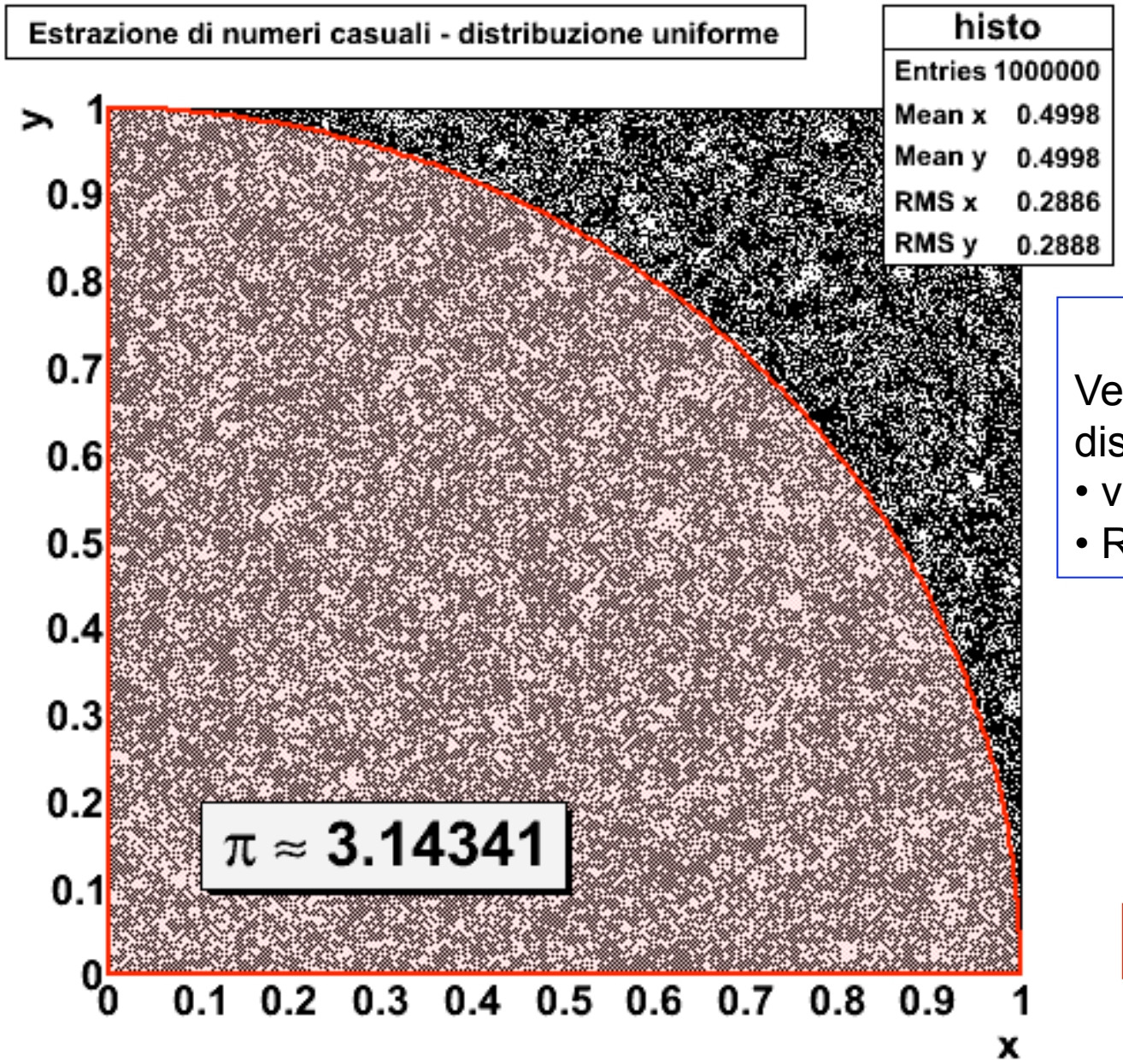
→ Creazione e visualizzazione di un'etichetta che mostra il risultato del calcolo:

```
TPaveLabel labelDisplay(.1,.1,.5,.2,"");
```

posizione dell'etichetta nella canvas

```
std::ostringstream label;
label <<"#pi #approx "<<pi;
labelDisplay.SetLabel(label.str().c_str());
labelDisplay.SetTextSize(0.75);
labelDisplay.Draw();
```

root[ ].x stima-pigreco.C



statistics box

Verificare i parametri della distribuz. uniforme (tra 0 e 1)

- valor medio = 0.5
- RMS = $1/\sqrt{12}$ =0.2887

enjoy ROOT

http://root.cern.ch/