Corso di Informatica A.A. 2009-2010

Lezione 8

Introduzione al calcolo numerico

- Derivazione
- Integrazione
- Soluzione di equazioni

Derivazione numerica

Il calcolo della derivata di una funzione in un punto implica un processo al limite che può solo essere approssimato da un calcolatore. Supponiamo nota la forma funzionale di f(x). Una prima approssimazione della sua derivata è:

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

Per h sufficientemente piccolo. Con h molto piccolo però può diventare delicato dal punto di vista numerico calcolare la differenza fra i due numeri a numeratore (molto vicini tra loro). Per capire come fare meglio analizziamo lo sviluppo in serie di Taylor di f in un intorno di x…

Derivazione numerica (II)

$$f(x+h) = f(x) + hf'(x) + \tfrac{1}{2}h^2 f''(x) + \tfrac{1}{6}h^3 f'''(x) + \ldots$$

da cui

$$\frac{f(x+h)-f(x)}{h} = f'(x) + \tfrac{1}{2}hf''(x) + O(h^2)$$

L'errore fatto sostituendo il rapporto incrementale scritto sopra al posto della derivata di *f* è dunque di ordine *h*.

Possiamo migliorare le cose notando che:

$$f(x-h) = f(x) - hf'(x) + \tfrac{1}{2}h^2 f''(x) - \tfrac{1}{6}h^3 f'''(x) + \ldots$$

da cui

$$\frac{f(x+h)-f(x-h)}{2h} = f'(x) + \tfrac{1}{6}h^2 f'''(x) + O(h^4)$$

Derivazione numerica (III)

Dunque a parità di h l'errore su

$$g(h) = \frac{f(x+h) - f(x-h)}{2h} = f'(x) + \tfrac{1}{6} h^2 f'''(x) + O(h^4)$$

sarà di ordine h^2 e g è quindi una approssimazione migliore di f'.

Per fare ancora meglio notiamo che:

$$g(2h) = f'(x) + \tfrac{2}{3} h^2 f'''(x) + O(h^4)$$

da cui:

$$\frac{4g(h) - g(2h)}{3} = f'(x) + O(h^4)$$

Derivazione numerica (IV)

Risolvendo per *f'(x)* otteniamo:

$$f'(x) = \frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)}{12h} + O(h^4)$$

Questa formula è sensibilmente più accurata e anche più stabile numericamente. Le formule con le differenze "asimmetriche" andrebbero utilizzate solo nel caso in cui sono definite solo la derivata destra e/o quella sinistra, o si ha una funzione per punti e occorre calcolare la derivata ad un estremo. Verificare che

$$f'(x) = \frac{f(x+4h) - 12f(x+2h) + 32f(x+h) - 21f(x)}{12h} + O(h^3)$$

Differenze finite

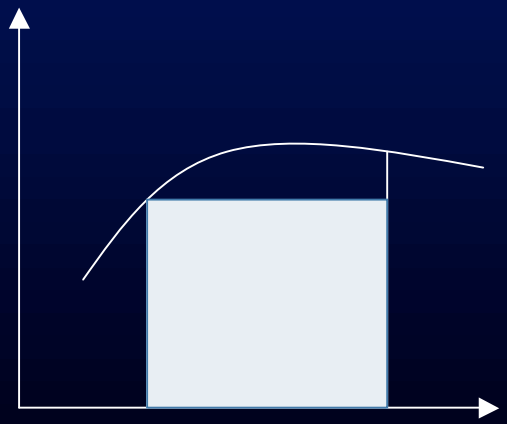
Sia x_i una sequenza di punti equispaziati a distanza h uno dall'altro e f_i i valori assunti dalla funzione f in ciascuno di essi. Utilizzando, come visto, lo sviluppo in serie di Taylor di f si possono ottenere le seguenti tabelle, con un errore $O(h^2)$ e $O(h^4)$ rispettivamente:

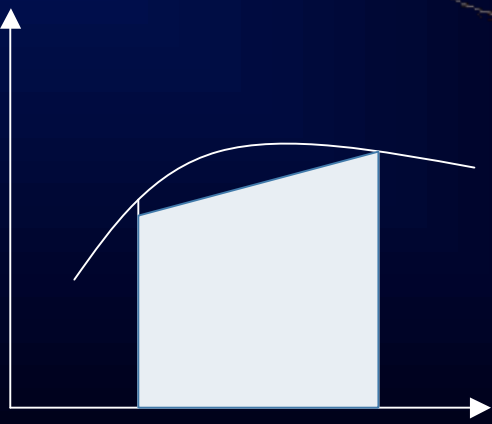
$O(h^2)$	f_{i-2}	f_{i-1}	f_i	f_{i+1}	f_{i+2}
$2hf_i'$		-1	0	1	
$h^2 f_i''$		1	-2	1	
$2h^3 f_i'''$	-1	2	0	-2	1

$O(h^4)$	f_{i-2}	f_{i-1}	f_i	f_{i+1}	f_{i+2}
$12hf_i'$	1	-8	0	8	-1
$12h^2 f_i''$	-1	16	-30	16	-1

Integrazione numerica

L'integrazione numerica si basa semplicemente sulla definizione di integrale definito e sulla interpolazione di una funzione in un certo intervallo con una funzione polinomiale data. Ad esempio se il grado del polinomio è zero si sta approssimando la funzione con una costante e si dovrà calcolare l'area di un rettangolo, se è 1 si sta considerando la funzione lineare e si dovrà calcolare l'area di un trapezio etc.





Il metodo dei trapezi

Con una approssimazione al primo ordine l'integrale definito di una generica funzione f si può scrivere:

$$\int_a^b f(x)dx \approx \frac{(b-a)}{2}(f(b)+f(a)) = I_1$$

Se l'intervallo [a,b] non è piccolo l'approssimazione lineare può risultare troppo rozza: ma sfruttando l'additività dell'integrale si può suddividere l'intervallo in n intervalli più piccoli di ampiezza h =(b-a)/n. Su ogni intervallino si avrà:

$$\int_{x_i}^{x_i+h} f(x)dx \approx \frac{h}{2}(f(x_i)+f(x_i+h)) = \frac{h}{2}(f_i+f_{i+1})$$

Il metodo dei trapezi (II)

Dati n intervallini si ottiene allora la stima dell'integrale data da:

$$\int_a^b f(x)dx \approx I_n = \frac{h}{2}(f_0 + 2f_1 + 2f_2 + \ldots + f_n)$$

Data la funzione *f* e l'intervallo *[a,b]* un calcolo dell'integrale potrà essere fatto per via iterativa calcolando la stima I_n dell'integrale al crescere di *n* e fermandosi quando due stime successive sono entro una certa tolleranza:

$$\int_a^b f(x)dx \approx I_n \text{ quando } \left|I_{nsucc} - I_n\right| < \varepsilon$$

Il metodo dei trapezi (III)

Il modo più efficiente di aumentare *n* è di raddoppiarlo ad ogni iterazione in modo da dover calcolare la funzione ogni volta solo su metà dei punti, avendola calcolata sull'altra metà nell'iterazione precedente:

$$h = \frac{b-a}{n}; \qquad h_{NEW} = \frac{b-a}{2n} = h/2$$

$$I_n = h\left(\frac{f_0}{2} + f_1 + f_2 + \ldots + \frac{f_n}{2}\right)$$

$$I_{2n} = \frac{I_n}{2} + h_{NEW} \sum_{k \text{ dispari}}^{2n-1} f(a + k \cdot h_{NEW})$$

Errore di troncamento

Valutiamo l'errore commesso nel calcolo dell'integrale usando, al solito, uno sviluppo in serie di Taylor:

$$\int_{x_i}^{x_i+h} f(x)dx = \int_0^h f(x_i+t)dt = \int_0^h (f(x_i)+tf'(x_i)+\frac{t^2}{2}f''(x_i)+\frac{t^3}{6}f'''(x_i))dt =$$

$$= f(x_i)h + f'(x_i)\frac{h^2}{2} + f''(x_i)\frac{h^3}{6} + O(h^4)$$

Da confrontare con la regola dei trapezi :

$$\int_{x_i}^{x_i+h} f(x)dx \approx \frac{h}{2}(f(x_i)+f(x_i+h)) =$$

$$\frac{h}{2}(f(x_i)+f(x_i)+f'(x_i)h+f''(x_i)\frac{h^2}{2}+O(h^3))$$

Errore di troncamento (II)

L'errore su una singola striscia vale allora:

$$f''(x_i)\frac{h^3}{12} + O(h^4)$$

L'errore sull'integrale è *n* volte quello su una singola striscia, e siccome *n = (b-a)/h* si ottiene:

$$Errore \text{ su } I_n = O(h^2) = O\left(\frac{1}{n^2}\right)$$

Il metodo di Simpson

L'approssimazione successiva a quella di una retta è una polinomiale di secondo grado (parabola). Il metodo di Simpson usa questa approssimazione per calcolare l'integrale e, come prevedibile, a parità di numero di strisce *n* ha un errore di troncamento inferiore a quello del metodo dei trapezi e pari a $O(1/n^4)$.

Usiamo un *numero n di strisce* e consideriamo per ogni punto x_i quello precedente x_{i-1} e quello successivo x_{i+1}. Si avrà:

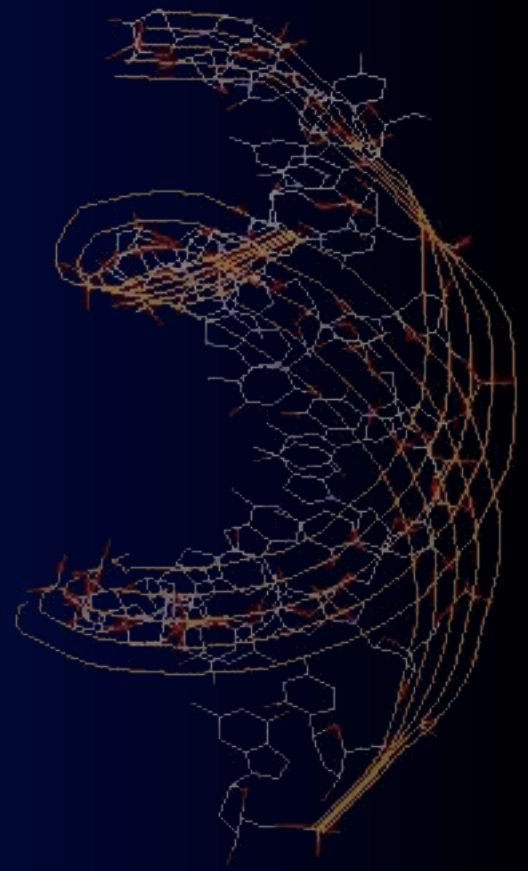
$$\int_{x_i-h}^{x_i+h} f(x)dx \approx \frac{h}{3}(f(x_i-h)+4f(x_i)+f(x_i+h)) = \frac{h}{3}(f_{i-1}+4f_i+f_{i+1})$$

$$I_n = \frac{h}{3}(f_0+4f_1+2f_2+4f_3+2f_4+\ldots+4f_{n-1}+f_n)$$

Applicazione

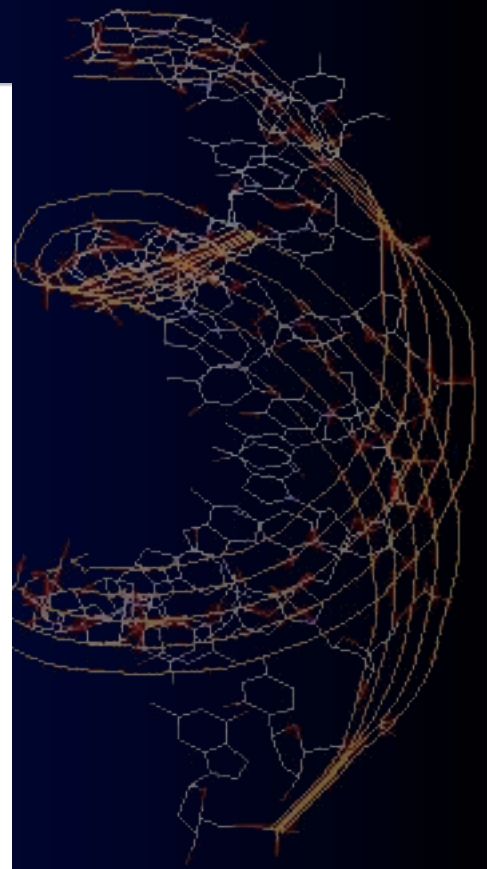
$$\int_0^3 \frac{1}{2+x^2} dx = 0.7992327$$

N	Rect	Trapez	Simpson
10	0.824581	0.798861	0.799231
20	0.812373	0.799140	0.799233
40	0.805925	0.799209	0.799233
100	0.801940	0.799229	0.799233
1000	0.799505	0.799233	0.799233
10000	0.799260	0.799233	0.799233



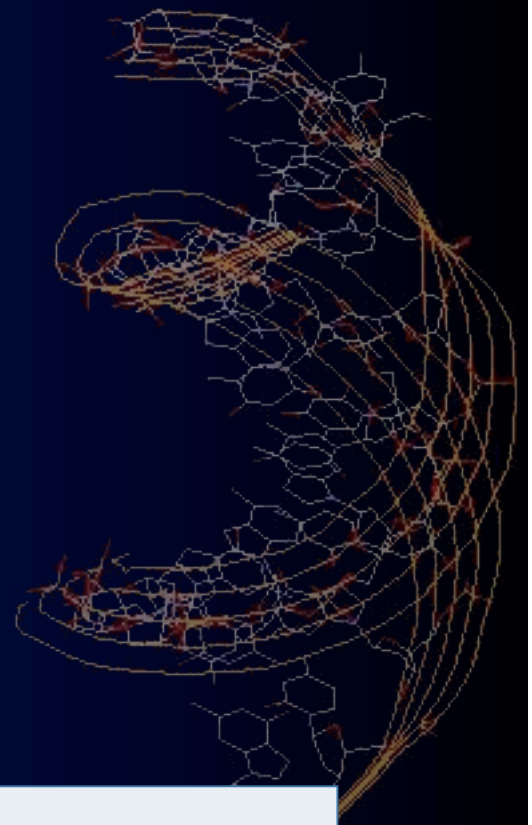
Esempio di codice

```c
#include <stdio.h>
#include <math.h>
//     Here we define various functions called by the main program
double trapezoidal_rule(double a, double b, int n);
double simpson_rule(double a, double b, int n);
double rectangle_rule(double a, double b, int n);
double int_function(double x);
//   Main function begins here
int main()
{
     int n;
     double a, b;
     printf("Read  in the number of integration points \n");
     scanf("%d",&n);
     printf("Read in integration limits \n");
     scanf("%lf %lf",&a,&b);

     printf("Rectangle rule %lf \n",rectangle_rule(a,b,n));
     printf("Trapezoidal rule %lf \n",trapezoidal_rule(a,b,n));
     printf("Simpson rule %lf \n",simpson_rule(a,b,n));
      return 0;
}  // end of main program
//  this function defines the function to integrate
double int_function(double x)
{
  double value = 1./(2.+x*x);
  return value;
} // end of function to evaluate
```



continua

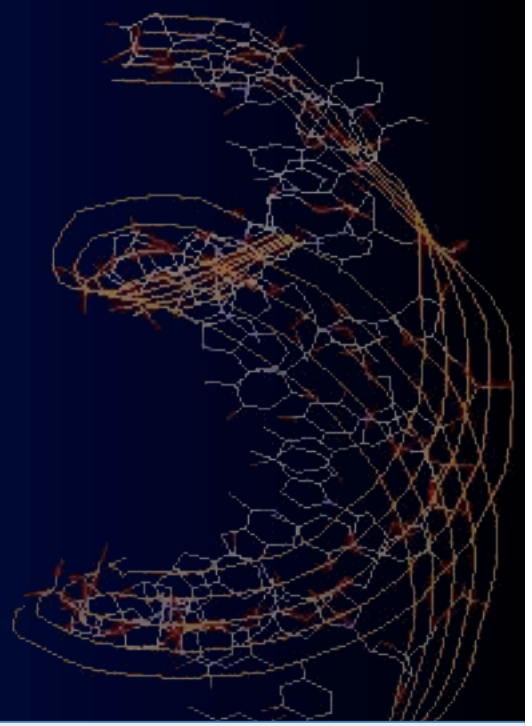
```
double trapezoidal_rule(double a, double b, int n)
{
      double trapez_sum;
      double fa, fb, x, step;
      int    j;
      step=(b-a)/((double) n);
      fa=int_function (a)/2. ;
      fb=int_function(b)/2. ;
      trapez_sum=0.;
      for (j=1; j <= n-1; j++){
         x=j*step+a;
         trapez_sum+=int_function(x);
      }
      trapez_sum=(trapez_sum+fb+fa)*step;
      return trapez_sum;
}  // end trapezoidal_rule
```



$$\int_a^b f(x)dx \approx I_n = \frac{h}{2}(f_0 + 2f_1 + 2f_2 + \ldots + f_n)$$

continua

```
double simpson_rule(double a, double b, int n)
{
      double simpson_sum;
      double fa, fb, fac, x, step;
      int    j;
      step=(b-a)/((double) n);
      fa=int_function (a) ;
      fb=int_function(b) ;
      simpson_sum=fa;
      fac=2.;
      for (j=1; j <= n-1; j++){
    fac=(fac==2.)?4. : 2. ;
         x=j*step+a;
         simpson_sum+=int_function(x)*fac;
      }
      simpson_sum=(simpson_sum+fb)*step/3;
      return simpson_sum;
}  // end simpson_rule
double rectangle_rule(double a, double b, int n)
```



$$\int_{x_i-h}^{x_i+h} f(x)dx \approx \frac{h}{3}(f(x_i-h)+4f(x_i)+f(x_i+h)) = \frac{h}{3}(f_{i-1}+4f_i+f_{i+1})$$

$$I_n = \frac{h}{3}(f_0+4f_1+2f_2+4f_3+2f_4+\ldots+4f_{n-1}+f_n)$$

```
      return rectangle_sum;
}  // end rectangle_rule
```

Soluzione di equazioni

Siccome una generica equazione in una variabile reale può essere messa nella forma

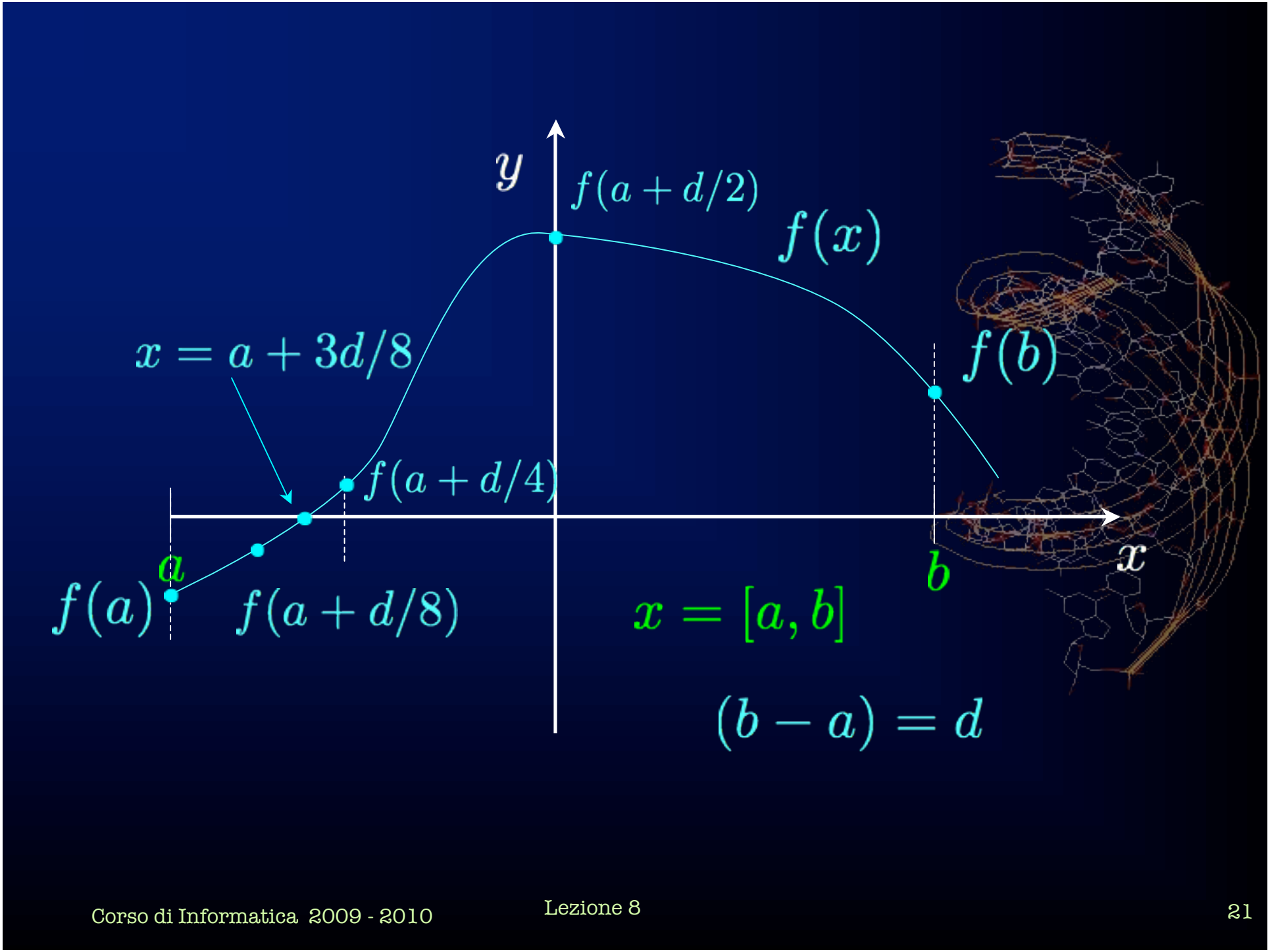
$$f(x) = 0$$

cercare una soluzione è equivalente a cercare lo zero di una funzione.

Metodo della bisezione…(o ricerca binaria)

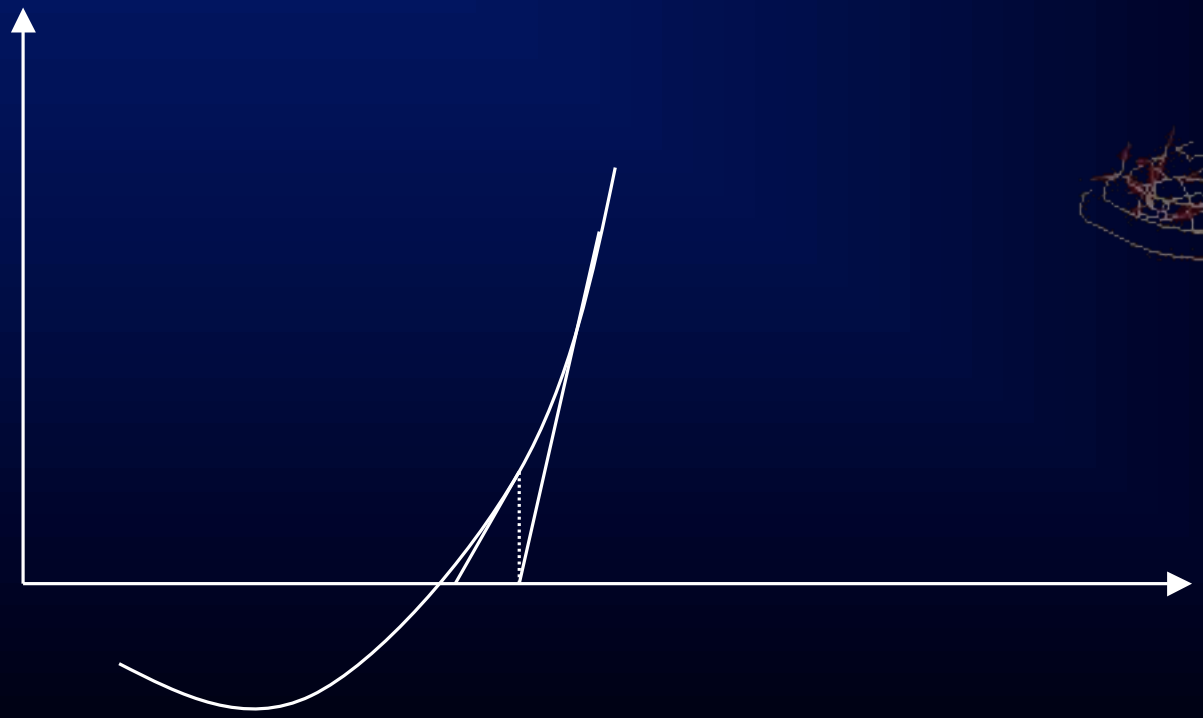
Se *f* è continua e lo zero cercato non è un punto di massimo o minimo relativo un metodo molto semplice ma sicuro e affidabile per trovare la soluzione è quello della bisezione, che ricorda molto da vicino l'algoritmo di ricerca binaria.

Individuato un intorno *[a,b]* dello zero in cui la funzione ha segno opposto si confronta *f(a)* con *f(a+(b-a)/2)* e *f(b)* e si sceglie il "ramo" i cui due estremi abbiano ancora segno opposto. Si itera il procedimento fino a trovare la soluzione cercata.


y
$f(a+d/2)$
$f(x)$
$x = a + 3d/8$
$f(b)$
$f(a+d/4)$
a
$f(a)$
$f(a+d/8)$
b
x
$x = [a, b]$
$(b-a) = d$

Metodo di Newton-Raphson

L'idea è di partire da un punto in cui la derivata di f sia non nulla, usare l'intersezione della tangente al grafico di f con l'asse delle x come prossima stima dello zero e poi iterare.



Metodo di Newton-Raphson (II)

Ricordando che la tangente al grafico di f in un punto ha per coefficiente angolare la derivata prima di f calcolata in quel punto si dimostra facilmente che:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

Lo zero è stato trovato quando due valori successivi di x, x_{i+1} e x_i differiscono tra loro per un valore inferiore all'accuratezza voluta.

Metodo di Newton-Raphson (III)

Il metodo di Newton-Raphson è del secondo ordine, nel senso che

detta $g(x) = x - \frac{f(x)}{f'(x)}$ si può dimostrare che

se c è lo zero della funzione ovvero f(c) = 0, vale

$\lim_{x \to c} \frac{g(x) - c}{(x-c)^2} = \text{costante}$ e quindi per x vicino a c è

$|g(x) - c| \approx K \cdot (x-c)^2$

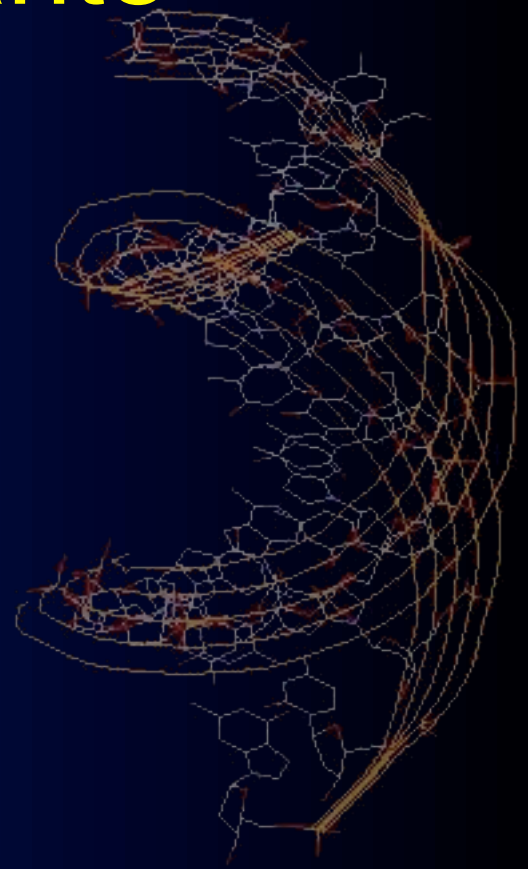
Il metodo della secante

Si parte dalla definizione di derivata

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

si sostituisce nell'iterazione di Newton-Raphson

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



La convergenza è migliore del metodo della bisezione , ma non buona come quella di Newton-Raphson (1.62)

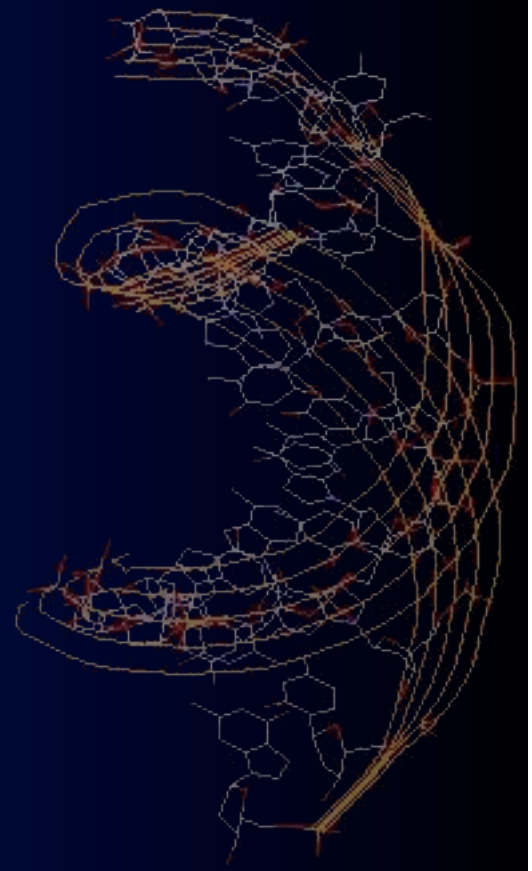
Esempio

Supponiamo di voler calcolare $\sqrt{R}.$

Trovare gli zeri di $f(x) = x^2 - R.$

Soluzione iterativa

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



Esempio

Calcoliamo $\sqrt{13}$ = 3.6055513 utilizzando l'iterazione precedente

Scegliamo $x_0=5$ → $x_1=3.8$

x_2= 3.6105263 x_3= 3.6055547 $x_4=3.6055513$

Con sole 4 iterazioni otteniamo il valore alla settima cifra decimale !!


Le funzioni in C

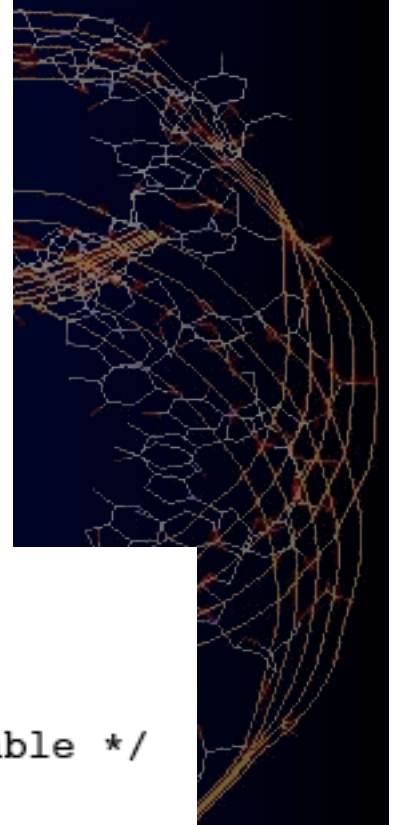
Funzioni

Una funzione e' un segmento di programma che assolve un compito elaborativo circoscritto e piu' facilmente identificabile. Le funzioni sono utilizzate per dividere un programma in un insieme di sottoprogrammi (modularizzazione).

Il C e' stato disegnato per rendere l'uso delle funzioni semplice ed efficiente: i programmi in C sono tipicamente costituiti da molte funzioni "piccole" piuttosto che da poche funzioni che eseguono molti compiti.

Una funzione è un blocco di codice a sé stante ed isolato dal resto del programma. Riceve dati e fornisce un risultato: ciò che avviene al suo interno è sconosciuto alla rimanente parte del programma, con la quale l'unica interazione e' il passaggio di dati di ingresso e uscita.

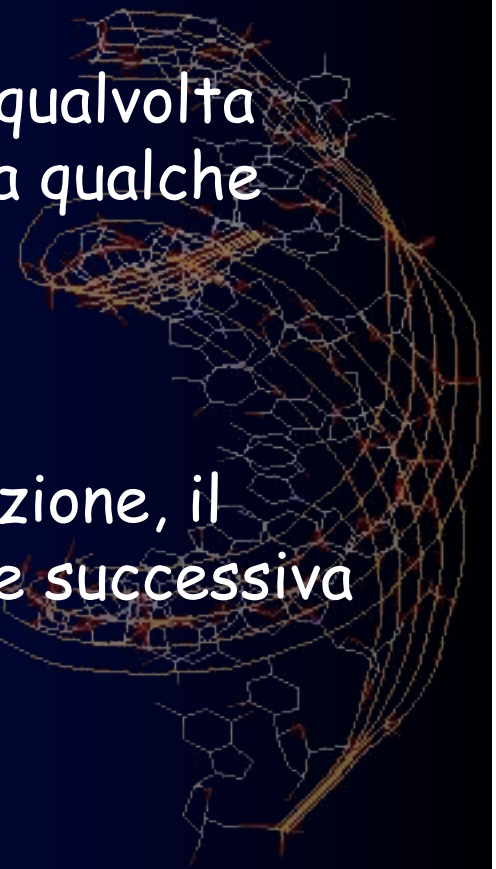
```c
/* Demonstrates a simple function */
#include <stdio.h>

long cube(long x);

long input, answer;

int main()
{
   printf("Enter an integer value: ");
   scanf("%d", &input);
   answer = cube(input);
   /* Note: %ld is the conversion specifier for */
   /* a long integer */
   printf("\nThe cube of %ld is %ld.\n", input, answer);

   return 0;
}

/* Function: cube() - Calculates the cubed value of a variable */
long cube(long x)
{
   long x_cubed;

   x_cubed = x * x * x;
   return x_cubed;
}
```

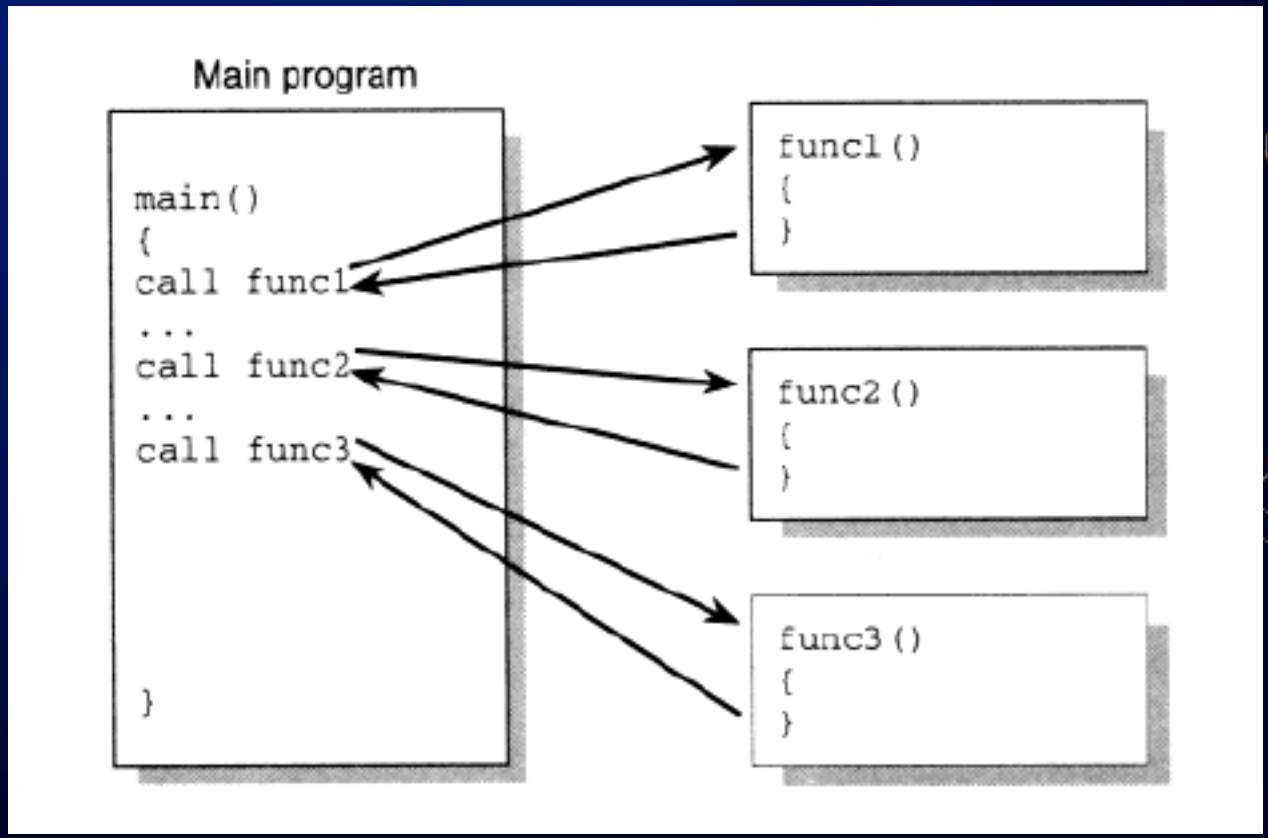


Funzioni

Una funzione esegue il proprio compito ogni qualvolta viene indirizzata ("chiamata" o "invocata") da qualche altra porzione del programma.

Dopo il completamento del compito della funzione, il flusso del programma continua dall'istruzione successiva a quella della chiamata.




Main program
main()
{
call func1
...
call func2
...
call func3
}
func1()
{
}
func2()
{
}
func3()
{
}

Sintassi delle funzioni

Function Prototype

```
return_type function_name( arg-type name-1, . . . ,arg-type name-n);
```

Function Definition

```
return_type function_name( arg-type name-1, . . . ,arg-type name-n)
{
    /* statements; */
}
```

Funzioni

La modularizzazione favorisce:

1. la *comprensibilita'* del codice (i dettagli possono essere nascosti e l'organizzazione logica quindi ne risulta piu' chiara);

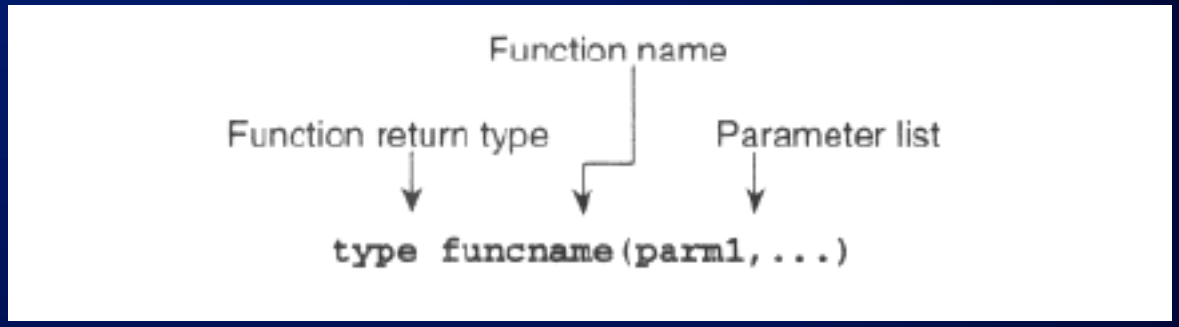
2. le sue possibili *modifiche* (circoscritte e piu' facilmente localizzabili);

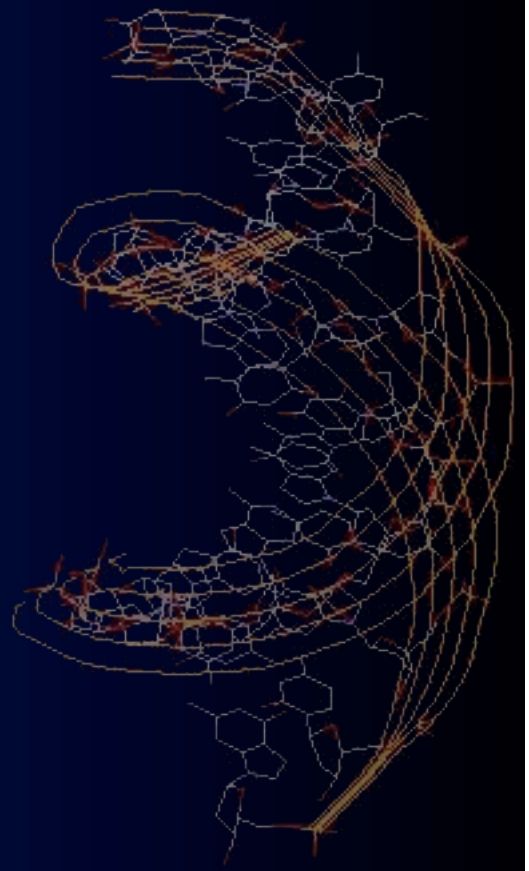
3. la *riutilizzazione* del codice (librerie di funzioni);

4. la localizzazione degli errori (debugging)

Un programma puo' inoltre essere diviso in piu' files compilabili separatamente.

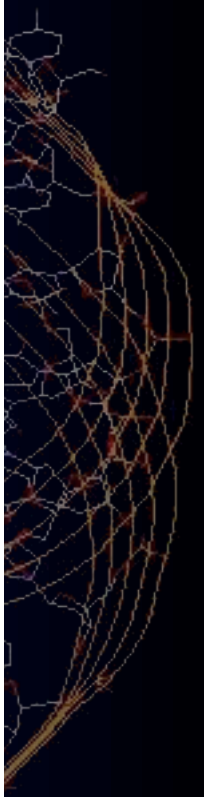
Intestazione della funzione (header)

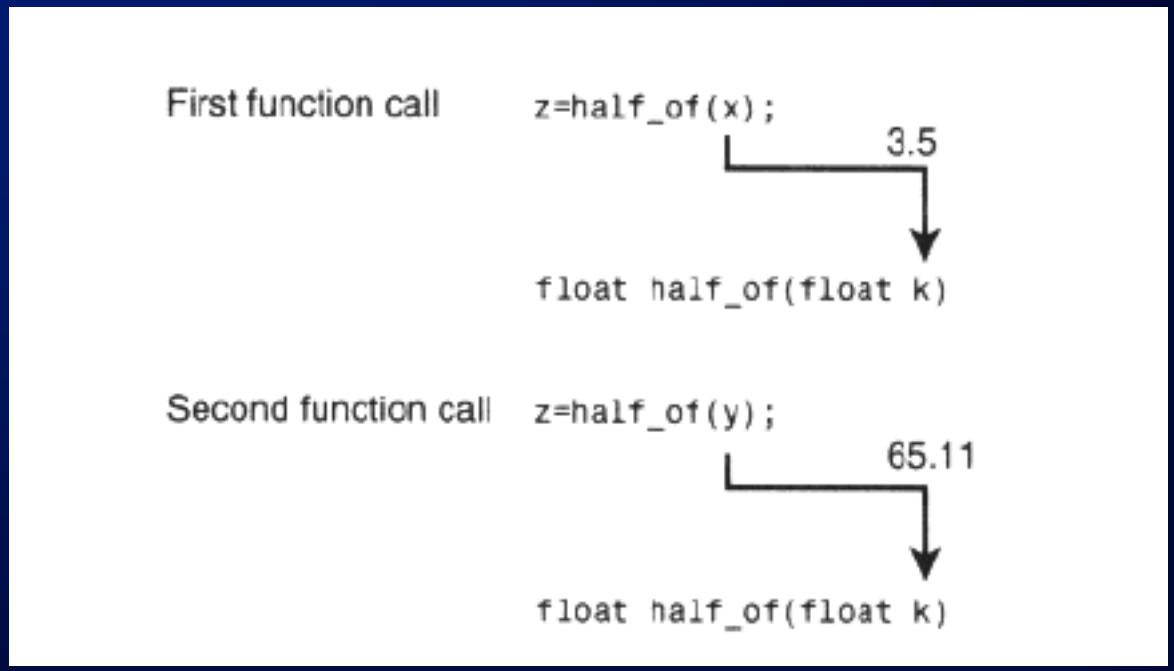


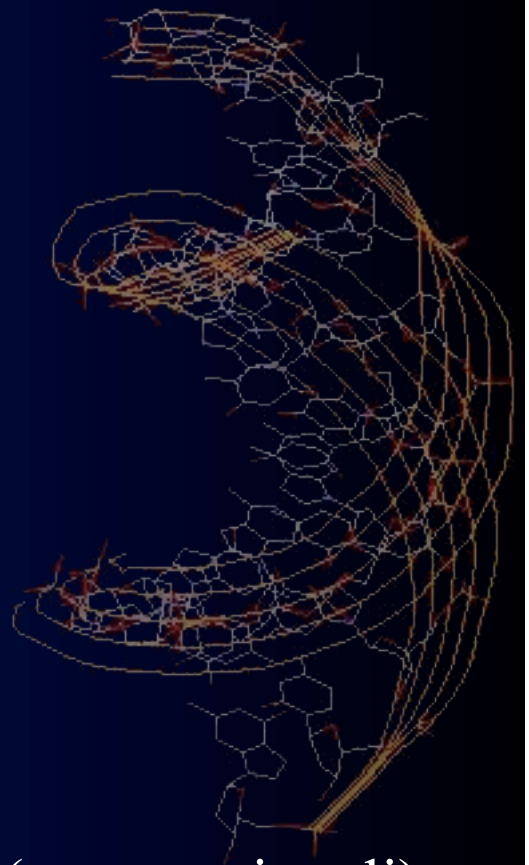


- Parametri formali
- Parametri reali (argomenti)

```c
/* Illustrates the difference between arguments and parameters. */

#include <stdio.h>

float x = 3.5, y = 65.11, z;

float half_of(float k);

int main()
{
    /* In this call, x is the argument to half_of(). */
    z = half_of(x);
    printf("The value of z = %f\n", z);

    /* In this call, y is the argument to half_of(). */
    z = half_of(y);
    printf("The value of z = %f\n", z);

    return 0;
}

float half_of(float k)
{
    /* k is the parameter. Each time half_of() is called, */
    /* k has the value that was passed as an argument. */

    return (k/2);
}
```



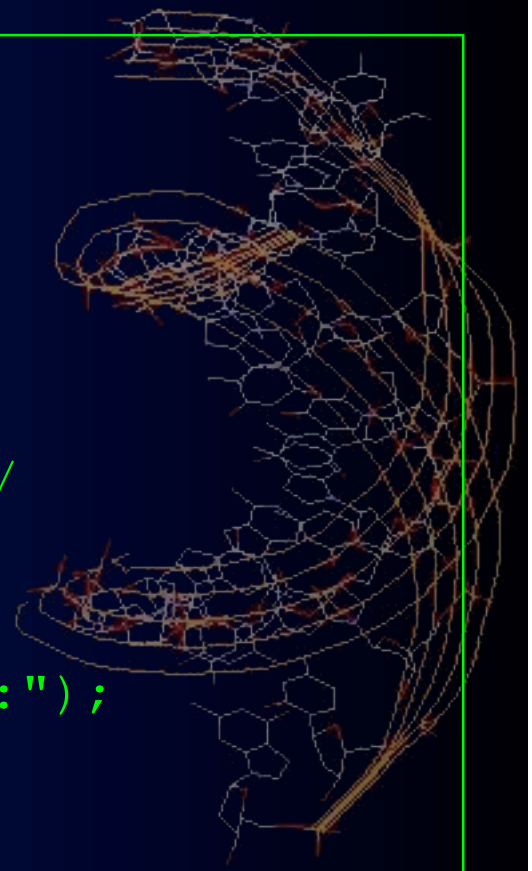




Ad ogni chiamata della funzione gli argomenti (parametri reali) sono passati ai parametri formali della funzione

Esempio

```
#include <stdio.h>
/* prototipi delle funzioni*/
int Somma(int a, int b);

main()
{
/*Dichiarazione variabili e costanti*/
int a,b,somma;

printf("\n Inserire due numeri interi:");
scanf("%d %d", &a, &b);
somma = Somma(a,b);
printf("La somma e' %d",somma);
/*Fine blocco principale e uscita dal programma*/
}
```



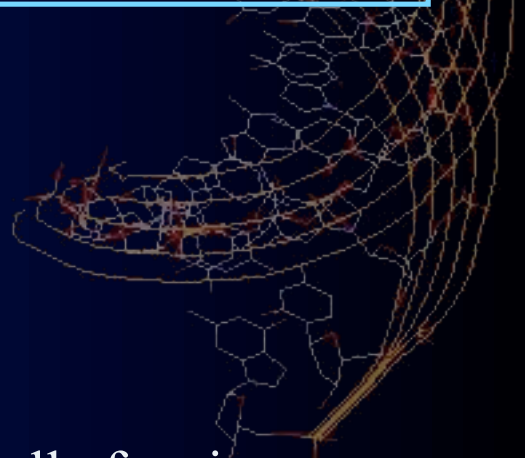
. . .definizione della funzione

intestazione (header)

argomenti o parametri formali: NON devono essere ridefiniti nel corpo della funzione

```
int Somma(int a, int b)
{
    int somma; //variabili locali
    somma=a+b;
    return somma;
}
```

corpo



IMPORTANTE: Tutte le variabili create nel corpo della funzione sono locali, nel senso che sono create al momento della dichiarazione e distrutte quando la funzione termina: non sono riconosciute all'esterno della funzione stessa.

Parametri Formali e Reali

```
#include <stdio.h>
/* prototipi delle funzioni */
int Somma(int, int);

main()
{
/*Dichiarazione variabili e costanti*/
int a,b,somma;

printf("\n Inserire due numeri interi:");
scanf("%d %d", &a, &b);
somma = Somma(a,b);
printf("La somma e' %d",somma);
}
```

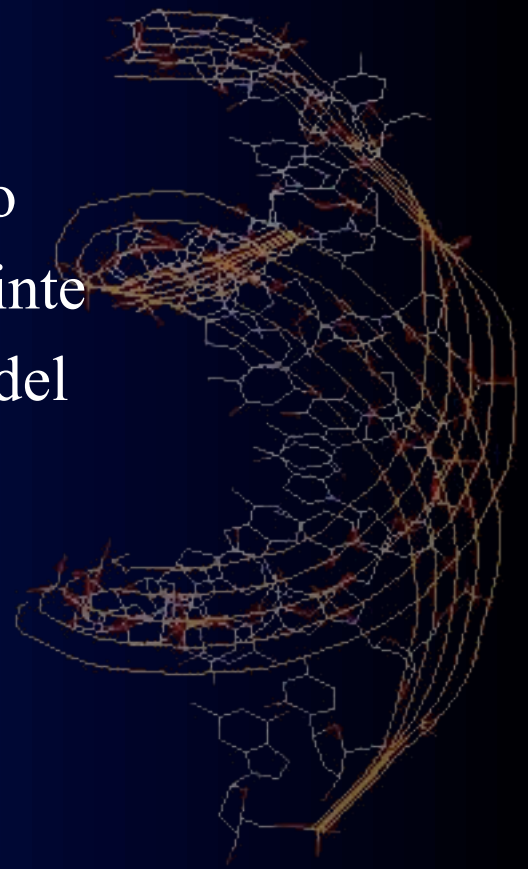
dichiarazione della funzione: prototipo. Notare che basta indicare il tipo delle variabili

parametri reali:

"actual parameters"

Variabili locali

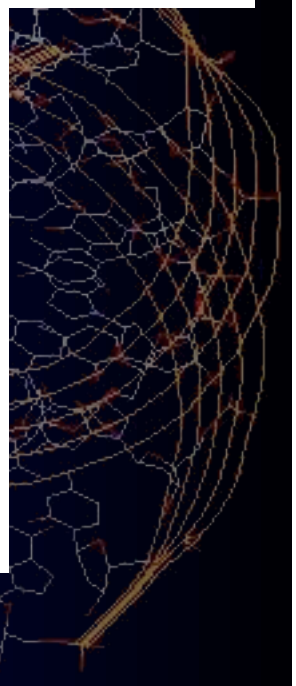
Le variabili definite all'interno di una funzione sono chiamate variabili locali e sono completamente distinte da tutte le altre variabili definite altrove all'interno del programma (anche se hanno lo stesso nome)



```
/* Demonstrates local variables. */

#include <stdio.h>

int x = 1, y = 2;

void demo(void);

int main()
{
  printf("\nBefore calling demo(), x = %d and y = %d.", x, y);
  demo();
  printf("\nAfter calling demo(), x = %d and y = %d\n.", x, y);

  return 0;
}

void demo(void)
{
    /* Declare and initialize two local variables. */

     int x = 88, y = 99;

     /* Display their values. */

     printf("\nWithin demo(), x = %d and y = %d.", x, y);
 }
```

OUTPUT:

```
Before calling demo(), x = 1 and y = 2.
Within demo(), x = 88 and y = 99.
After calling demo(), x = 1 and y = 2.
```



Ritornare un valore

Per ritornare un valore da una funzione si utilizza l'istruzione `return` seguita da un'espressione in C

Il valore ritornato dalla funzione è quello assunto dalla espressione

```
int func1(int var)
{
    int x;
    /* Function code goes here . . . */
    return x;
}
```

```c
/* Demonstrates using multiple return statements in a function. */

#include <stdio.h>

int x, y, z;

int larger_of( int a, int b);

int main()
{
    puts("Enter two different integer values: ");
    scanf("%d%d", &x, &y);

    z = larger_of(x,y);

    printf("\nThe larger value is %d.", z);

    return 0;
}

int larger_of( int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}
```

Chiamare una funzione

Esistono due modi di chiamare una funzione

1. Ogni funzione può essere chiamata utilizzando il suo nome da solo in un'istruzione

```
wait(12);
```

2. Al posto di una qualsiasi espressione in C (solo per funzioni che ritornano un valore)

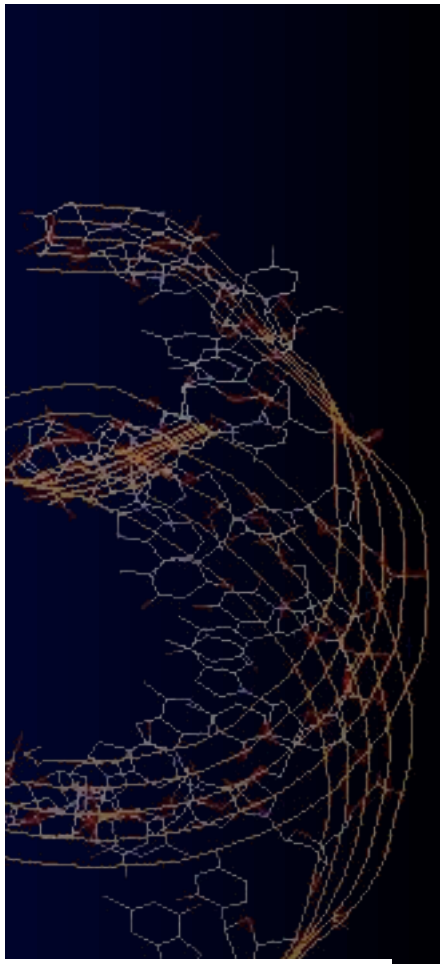
```
printf("Half of %d is %d.", x, half_of (x));
```

Ricorsione

Si parla di ricorsione quando una funzione chiama se stessa. Il C consente l'uso di funzioni ricorsive

Un tipico esempio in cui la ricorsione risulta essere estremamente utile è il calcolo del fattoriale di un numero

```c
/* Demonstrates function recursion. Calculates the */
/* factorial of a number. */

#include <stdio.h>

unsigned int f, x;
unsigned int factorial(unsigned int a);

int main()
{
    puts("Enter an integer value between 1 and 8: ");
    scanf("%d", &x);

    if( x > 8 || x < 1)
    {
        printf("Only values from 1 to 8 are acceptable!");
    }
    else
    {
        f = factorial(x);
        printf("%u factorial equals %u\n", x, f);
    }

    return 0;
}
```



```c
unsigned int factorial(unsigned int a)
{
    if (a == 1)
        return 1;
    else
    {
        a *= factorial(a-1);
        return a;
    }
}
```