Università Politecnica delle Marche
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea in Ingegneria Informatica e dell'Automazione

METODI DI INTEGRAZIONE

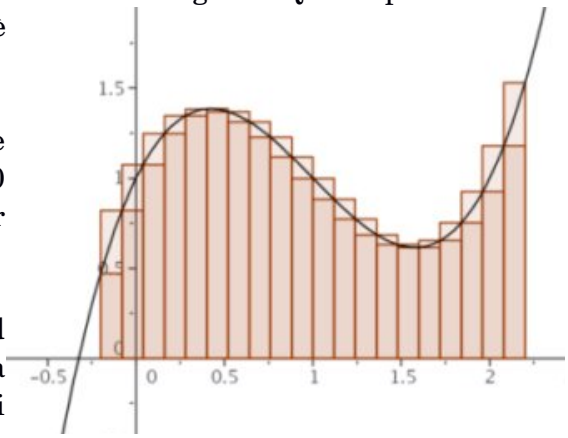
Caratteristiche e analisi dei risultati

Nicola Fioranelli
Matricola 1061904

Corso di Algoritmi e Strutture dati
Prof.ssa Giuseppa Ribighini
A.A. 2014-2015

PREMESSA

Il calcolo integrale è un problema che per secoli ha occupato le menti dei matematici di varie popolazioni. Intuitivamente, per calcolare l'area della regione di piano compresa tra il grafico di $f(x)$ e l'asse delle ascisse nell'intervallo $[a,b]$ potremo pensare di approssimarla mediante l'area di "plurirettangoli" inscritti e circoscritti in tale regione. Questo procedimento, noto sin dai tempi dei matematici ellenici è conosciuto come metodo di elusione.



Le prime basi teoriche per il calcolo integrale vennero gettate da Cavalieri intorno al 1600 attraverso il Metodo degli indivisibili, utilizzato per il calcolo di aree e volumi di figure geometriche.

Tuttavia si deve attendere il 1800 per arrivare ad una definizione rigorosa di integrale: quella elaborata da Riemann. Partendo dalla nozione di somma integrale inferiore, $s_f(P)=\sum_{k=1}^{n}(inf(f))\cdot(x_k-x_{k-1})$ e somma integrale superiore $S_f(P)=\sum_{k=1}^{n}(supe(f))\cdot(x_k-x_{k-1})$ giunge a definire l'integrale secondo Riemann della funzione $f(x)$ nell'intervallo $[a,b]$ come il valor comune fra le due sommatorie: $\int_a^b f(x)\,dx$

IL PROGETTO

Il progetto vuole mettere in luce i metodi di integrazione utilizzati dagli elaboratori nel calcolo integrale, analizzandone le differenti peculiarità.
Dopo aver spiegato gli algoritmi di integrazione tipici dell'analisi numerica (rettangoli, trapezi e Simpson) ne illustreremo le differenze: l'analisi computazionale, i risultati ottenuti in termini di precisione, fissato il numero di sotto intervalli, e in termini di tempo di esecuzione, fissata la precisione.

Il programma permette di scegliere due tipi di utilizzo (precisione o sotto intervalli) prima dell'esecuzione degli algoritmi. Il codice è stato implementato in C++ ('11); l'elaboratore sul quale è stato eseguito il programma monta un processore AMD E-series con 1.5 Ghz di frequenza, 4 Gb di memoria Ram, O.S. Linux Ubuntu v.14.04.
Il compilatore utilizzato è GNU GCC Compiler all'interno della suite Code::Blocks IDE.

METODI DI INTEGRAZIONE

Per poter calcolare l'integrale di una funzione con un elaboratore è necessario introdurre metodi numerici per il calcolo. I seguenti tre sono i metodi più diffusi e di seguito ne illustrerò brevemente il loro carattere matematico:

RETTANGOLI: Questo metodo parte dall'idea di "interpolare" la funzione in un solo punto (di solito uno dei due estremi di integrazione); così facendo ottengo un integrale molto approssimato con errore di arrotondamento pressoché nullo (dato il numero limitato di operazioni da eseguire) ma errore di troncamento molto elevato.

$$\int_a^b f(x)\,dx \simeq h \cdot f(a+h)$$

Essendo un metodo iterativo è possibile riscrivere la formula utilizzando la sommatoria:

$$\int_a^b f(x)\,dx \simeq \sum_{i=0}^{n-1} h \cdot f(a+i \cdot h)$$

$$\text{dove} \quad h=\frac{b-a}{n} \quad .$$

Il caso limite è $f(x)=c$.

TRAPEZI O BEZOUT: Il metodo dei trapezi risulta avere una precisione leggermente migliore per via dell'interpolazione su due punti (di solito f(a) e f(b)). La formula di iterazione, che prende il nome dal matematico Bezout, rimane quasi invariata se non per l'aggiunta degli estremi di integrazione:

$$\int_a^b f(x)\,dx \simeq [\sum_{i=0}^{n-1} h \cdot f(a+i \cdot h)] + \frac{(f(a)+f(b))}{2}$$

$$\text{dove} \quad h=\frac{b-a}{n} \quad .$$

Il caso limite è $f(x)=mx+q$.

SIMPSON: Quest'ultimo metodo è il più preciso in quanto suddivide l'intervallo in somme dispari e pari (compiendo di fatto anche meno calcoli), imponendo il passaggio per 3 punti. La sommatoria che descrive l'iterazione è più complessa ma ciò per rendere più agile l'algoritmo:

$$\int_a^b f(x)\,dx \simeq \frac{h}{3} \cdot (f(a)+4 \cdot \sum_{i=1}^{n-1,2} f(x)+2 \cdot \sum_{i=2}^{n-2,2} f(x)+f(b))$$

$$\text{dove} \quad h=\frac{b-a}{n} \quad .$$

IMPLEMENTAZIONE IN in C++

Di seguito riporto i codici delle funzioni che implementano i tre algoritmi.

```
double rettangoli(double a, double b, int n, int t){
    double x=0, i=0, h=0, s=0;
    h=(b-a)/n;
    for (i=0; i<n; i++){
        x=a+(i*h);
        s+=funzione(t,x);
    }
    return s*h;}
```

Il codice è intuitivamente molto semplice: subito dopo la dichiarazione dei parametri e l'inizializzazione delle variabili locali, viene calcolata l'ampiezza h dell'intervallo ed effettuato il ciclo `for` che corrisponde alla sommatoria.

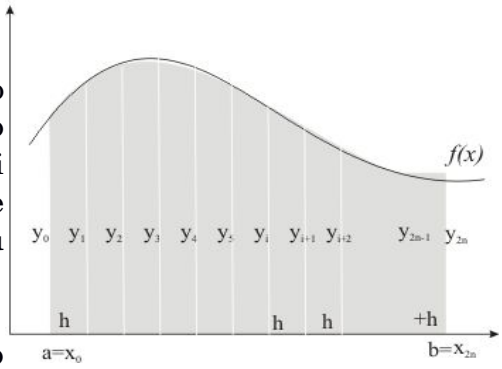
Alla sua semplicità e velocità si affianca però una grande imprecisione di calcolo.

```
double trapezi(double a, double b, int n, int t){
    double x=0, i=0, h=0, s=0;
    h=(b-a)/n;
    s=(funzione(t,a)+funzione(t,b))/2;
    x=a+h;
    for (i=1; i<n; i++){
        s+=funzione(t,x);
        x+=h;
    }
    return s*(h/2);
}
```

Anche il codice di questa funzione è molto semplice e risulta tuttavia più preciso per via del passaggio per i due estremi di integrazione.

```
double simpson(double a, double b, int n, int t){
    double i=0, h=0, SD=0, SP=0, x=0;
    if((n/2*2)!=n) ++n;
    h=(b-a)/n;
    x=a;
    for (i=1; i<n-2; i=i+2){
        x+=h;
        SD+=funzione(t,x);
        x+=h;
        SP+=funzione(t,x);
    }
    x+=h;
    SD+=funzione(t,x);
    return (h/3)* (funzione(t,a) + 4*SD + 2*SP + funzione(t,b));
}
```

Il codice risulta essere molto più complesso nonostante venga fatto comunque un solo ciclo `for`.



Il primo controllo è necessario per verificare che il numero di sotto intervalli sia pari. Viene effettuato in questo modo poiché nel caso di divisione per 2 di un numero dispari di tipo `int` il risultato sarebbe troncato nella sua parte decimale e una volta moltiplicato per 2 non si avrebbe più il numero di partenza.

Il `for` procede con passo uguale a 2 incrementando al suo interno ora la somma dispari ora quella pari. L'ultimo passo è calcolare l'area come nella formula.

IL PROGRAMMA

Il programma che ho implementato permette di scegliere se fissare il numero di sotto intervalli per valutare la precisione o se fissare la precisione per valutare le iterazioni. Nel secondo caso ho inserito nel codice delle funzioni un `do... while` che permette di ciclare le operazioni necessarie fintanto che non si ottiene la precisione desiderata.

Il programma permette di scegliere fra nove diverse funzioni in un intervallo arbitrario che verrà richiesto all'inizio dell'esecuzione: tuttavia per comodità noi analizzeremo i dati di x^2 ed e^x solo perché più facilmente calcolabili analiticamente.

ANALISI DEI RISULTATI

Da un punto di vista di analisi computazionale i tre codici fanno all'incirca le stesse operazioni, ovvero un solo ciclo `for` iterato n volte.
Analizzando invece l'occupazione di memoria il metodo di Simpson risulta il più oneroso in quanto è quello con più variabili in gioco.
Quanto al tempo di esecuzione il più veloce è ancora il metodo di Simpson che con meno iterazioni arriva alla soluzione, vediamo alcuni risultati:

$$\int_0^2 x^2\,dx = 2{,}66666$$

con una precisione fissata di 0,0001 ottengo:

	Valore ottenuto	Iterazioni	Errore
METODO DEI RETTANGOLI	2,66661	16	0,00005
METODO DEI TRAPEZI	2,66669	8	0,00003
METODO DI SIMPSON	2,66667	1	0,00001

L'errore minore viene commesso dal Metodo di Simpson con fra l'altro anche il minore numero di iterazioni. Un analogo lo otteniamo andando ad inserire manualmente il numero di sotto intervalli ottenendo:

METODO DEI RETTANGOLI	2,28	2,5872
METODO DEI TRAPEZI	1,34	1,3336
METODO DI SIMPSON	2,6667	2,6667
SOTTOINTERVALLI	10	50

Si può facilmente notare che già a 10 sotto intervalli il metodo di Simpson da il risultato esatto approssimato in eccesso nell'ultima cifra decimale.
Le medesime considerazioni sono state confermate nel calcolo di un altro integrale:

$\int_0^2 e^x dx = 6{,}38906$ con una precisione fissata di 0,0001 ottengo:

METODO DEI RETTANGOLI	6,38896	16	0,0001
METODO DEI TRAPEZI	6,38909	8	0,00003
METODO DI SIMPSON	6,38906	4	0
	Valore ottenuto	Iterazioni	Errore

E fissando il numero di sottointervalli:

METODO DEI RETTANGOLI	5,77143	6,26216
METODO DEI TRAPEZI	5,20517	6,19495
METODO DI SIMPSON	6,38911	6,38906
SOTTOINTERVALLI	10	50

Ho voluto "forzare" il programma in un caso particolare: $\int_{0{,}0001}^{1} \sin\left(\frac{1}{x}\right)$.

La funzione infatti ha un comportamento molto particolare al tendere di x a 0: infatti le oscillazioni in sua prossimità aumentano notevolmente creando fra l'altro problemi nel calcolo dell'integrale con i metodi fin'ora visti. Lo scopo era quello di avvicinarsi con il primo estremo di integrazione allo zero (caso in cui il valore ottenuto è NotANumber).
Il programma con una precisione fissata di 0,00000001 ha prodotto con Simpson un'area di: 0,504067 con 26 iterazioni e 65 secondi di tempo di esecuzione.

CASI LIMITE

Di seguito riporto invece i tre casi limite per i metodi considerati:

$\int_0^2 4\,dx = 8$ Con una precisione fissata di 0,0001 si ottiene il risultato esatto e numero di iterazioni per i tre metodi pari a 1.

$\int_0^2 (2x+4)\,dx = 12$ Con una precisione fissata di 0,0001 si ottiene:

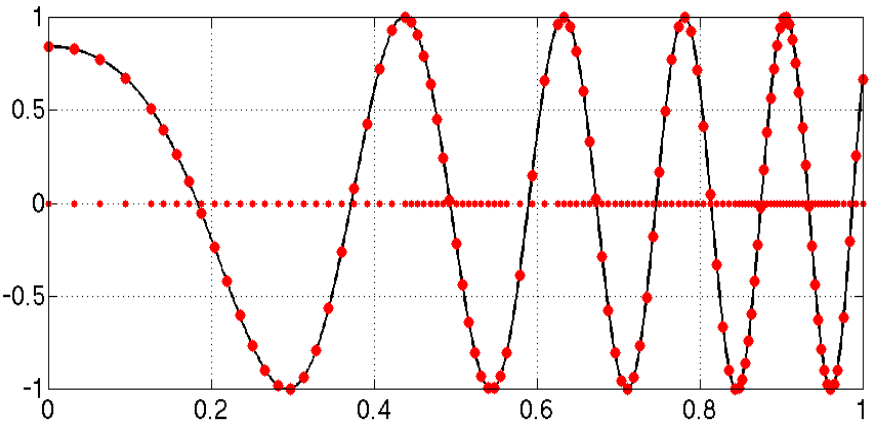
METODO DEI RETTANGOLI	11,9999	16	0,0001
METODO DEI TRAPEZI	12	1	0
METODO DI SIMPSON	12	1	0
	Valore ottenuto	Iterazioni	Errore

$\int_0^2 (x^2+4x+2)\,dx = 14{,}6667$ Con una precisione fissata di 0,0001 si ottiene:

METODO DEI RETTANGOLI	14,6666	17	0,0001
METODO DEI TRAPEZI	14,6667	8	0
METODO DI SIMPSON	14,6667	1	0
	Valore ottenuto	Iterazioni	Errore

OTTIMIZZAZIONE DEL METODO DI SIMPSON: il caso della quadratura adattiva

Fin'ora mi sono occupato solo degli algoritmi trattati nel corso, tuttavia mi è sembrato interessante poter allargare la visione su quello che il campo in cui in un futuro prossimo lavorerò.



MATLAB è un software di calcolo molto potente; originariamente creato per poter lavorare con matrici e vettori, si è evoluto a tal punto da essere uno dei migliori software in circolazione.

Ho cercato fra le funzioni che MATLAB mette a disposizione degli utenti per il calcolo integrale e ho scoperto che `quad(fun,a,b)` usa proprio il metodo di Simpson.

Per rendere efficiente la funzione l'implementazione vista sopra è stata leggermente modificata grazie al metodo di quadratura di Simpson adattivo. Cercherò brevemente di illustrarne le caratteristiche salienti.

Consideriamo la funzione nel grafico sopra: usando il metodo studiato nel corso andremmo a suddividere l'intervallo $[0,1]$ in n sotto intervalli di ampiezza h tutti uguali; così facendo però otterremmo fin troppi punti nell'intervallo $[0,0.2]$ (in cui la funzione è facilmente approssimabile) mentre troppo pochi punti per esempio nell'intervallo $[0.8,1]$ (in cui invece date le oscillazioni non riusciremmo a rendere bene la funzione).

Il metodo adattivo, come dice la stessa parola, si adatta alla funzione valutando di volta in volta la differenza tra l'errore ottenuto e quello richiesto. In pratica l'algoritmo suddivide l'intervallo considerato in due sotto intervalli, dove applica la regola di Simpson vista nel codice che avevo scritto; mediante una formula valuta l'errore in entrambi i sotto intervalli e se questo eccede la precisione richiesta procede iterativamente dividendo ancora in due il sotto intervallo.

La cosa apparentemente sembrerebbe uguale a dimezzare l'ampiezza h dei sotto intervalli (come era stato fatto nel programma), ma consideriamo il caso particolare di una funzione, costante per $x<0$ e con andamento cubico per $x\geq 0$ di cui vogliamo calcolare l'integrale nell'intervallo $[-1,1]$: ciò creerebbe non pochi problemi al codice implementato nella forma tradizionale. Infatti andremmo a dimezzare l'ampiezza dei sotto intervalli fino a raggiungere la precisione fissata, compiendo inutili calcoli nella funzione costante.

La quadratura adattiva, se pur a livello teorico e senza averne constatato i reali vantaggi, andrebbe a suddividere in sotto intervalli più piccolo solo l'intervallo relativo alla cubica, velocizzando l'esecuzione e riducendo la possibilità d'errore di approssimazione in calcoli già ben fatti.

La procedura implementata su Matlab utilizza come funzione `int(fun,min,max)` poiché la funzione `quad` non è ancora disponibile su tutte le versioni del software, il principio di funzionamento rimane tuttavia invariato. Nel codice utilizzo `syms` per creare la variabile simbolica x sulla quale poter valutare le due funzioni exp(x) e x^2.

Di seguito riporto il codice del programma Matlab che implementa il calcolo dell'integrale definito di e^x e di x^2

```
1. syms x
2. int (exp(x), 0, 2)
3.
4. syms x
5. int (x^2, 0, 2)
```

Le due espressioni danno come risultati `exp(2)-1` ovvero `6,3896` e `8/3` ovvero `2,6667`. E' chiaro come il concetto alla base di Matlab sia nettamente più avanzato del programma scritto in C++, non a caso i risultati ottenuti non sono mai numeri ma espressioni numeriche più complesse.

CONCLUSIONI

Dopo aver analizzato i diversi algoritmi e i risultati ottenuti con il loro utilizzo, la conclusione che ne deriva è che il miglior metodo è quello di Simpson, in quanto con un numero sufficientemente piccolo di sotto intervalli riesce già ad ottenere un risultato soddisfacente ed iterando il procedimento è possibile avere un valore non troppo contaminato da errori.

La conferma di quanto sostenuto in queste conclusioni è data dalla funzione utilizzata dal MATLAB, che dovendo essere ottimizzata conferma la bontà dell'algoritmo.

Nicola Fioranelli
Ancona, 26/04/2015

Bibliografia

1. Piero Montecchiari, Francesca G. Alessio, Note di Analisi Matematica 1. Esculapio (2013)
2. P. Foggia, M. Vento, Algoritmi e strutture dati. McGraw-Hill (2011)
3. Wikipedia: Adaptive Simpson's method, Clenshaw-Curtis quadrature.

```
1. #include <iostream>
2. #include <math.h>
3. #include <stdlib.h>
4.
5. using namespace std;
6.
7. double rettangoli(double a, double b, int n, double eps, int t, int test);
8. double trapezi(double a, double b, int n, double eps, int t, int test);
9. double simpson(double a, double b, int n, double eps, int t, int test);
10.double funzione(int t, double d);
11.
12. int main()
13.{
14.      double A, B, EPS;
15.      int T, TEST;
16.
17.      cout << "Benvenuto/a nel programma del calcolo integrale" << endl;
18.      cout << "Inserisci il primo estremo: ";
19.      cin >> A;
20.      cout << "Inserisci il secondo estremo: ";
21.      cin >> B;
22.      cout << "=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=" << endl;
23.      cout << "Scegli la funzione:" << endl;
24.      cout << " 1) Integrale di x*e^(-x)*cos(2x)" << endl;
25.      cout << " 2) Integrale di 1/(1+x^2)" << endl;
26.      cout << " 3) Integrale di x*sin(x)" << endl;
27.      cout << " 4) Integrale di e(-x^2)" << endl;
28.      cout << " 5) Integrale di e^x" << endl << "==== CASI LIMITE ====";
29.      cout << " 6) Integrale di 4" << endl;
30.      cout << " 7) Integrale di 2*x+4 << endl;
31.      cout << " 8) Integrale di x^2+4*x+2" << endl;
32.      cout << " 9) Integrale di sin(1/x)" << endl;
33.      cout << "Per uscire dal programma puoi inserire 0 nel numero di
   sottointervalli..." << endl;
34.      cout << "-->";
35.      do {
36.      cin >> T;
37.      if (T!=1 && T!=2 && T!=3 && T!=4 && T!=5 && T!=6 && T!=7 && T!=8
   &&T!=9){
38.      cout << "! Valore errato, inserisci -->";
39.          }
40.      } while(T!=1 && T!=2 && T!=3 && T!=4 && T!=5 && T!=6 && T!=7 && T!=8
   &&T!=9);
41.      cout << "=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=" << endl;
42.
43.      // il test permette di proseguire in due metodi diversi
44.      cout << "Inserisci '1' se vuoi valutare il numero di iterazioni
   fissata la precisione" << endl;
45.      cout << "Inserisci '0' se vuoi valutare i risultati fissando di
   volta in volta il numero di sottointervalli" << endl;
46.      cin >> TEST;
47.
48.      // con il test vero si valuta il numero di iterazioni
49.      // con il test falso l'errore con il numero di sotto intervalli
50.      //(nelle funzioni non vengono eseguiti i cicli)
51.          if (TEST == 1){
52.              int N;
53.              double RET, TRA, SIM;
```

```
54.             cout << "Inserisci la precisione: ";
55.             cin >> EPS;
56.             N=1;
57.             cout << endl << endl << "==NUMERO DI ITERAZIONI==" << endl;
58.             RET=rettangoli(A,B,N,EPS,T,TEST);
59.             TRA=trapezi(A,B,N,EPS,T,TEST);
60.             SIM=simpson(A,B,N,EPS,T,TEST);
61.              cout << endl << endl << "L'integrale della funzione " << T
    << " fra " << A << " e " << B << " e':" << endl;
62.             cout << "METODO DEI RETTANGOLI: " << RET << endl;
63.             cout << "METODO DEI TRAPEZI: " << TRA << endl;
64.             cout << "METODO DI SIMPSON: " << SIM << endl;
65.             return 0;
66.         }
67.         else {
68.             for(;;){
69.             int N;
70.             double ERR_RET, ERR_TRA, RET, TRA, SIM;
71.             cout << "Inserisci il numero di sottointervalli: ";
72.             cin >> N;
73.             if (N==0) return 0;
74.
75.             RET=rettangoli(A,B,N,EPS,T,TEST);
76.             TRA=trapezi(A,B,N,EPS,T,TEST);
77.             SIM=simpson(A,B,N,EPS,T,TEST);
78.              cout << endl << endl << "L'integrale della funzione " << T
    << " fra " << A << " e " << B << " e':" << endl;
79.             cout << "METODO DEI RETTANGOLI: " << RET << endl;
80.             cout << "METODO DEI TRAPEZI: " << TRA << endl;
81.             cout << "METODO DI SIMPSON: " << SIM << endl;
82.             ERR_RET=(RET-SIM)/2;
83.             ERR_TRA=(TRA-SIM)/2;
84.
85.             }
86.             return 0;
87.         }
88.
89.      }
90.
91.      // la function 'rettangoli' restituisce l'integrale calcolato con
    tale metodo
92.
93. double rettangoli(double a, double b, int n, double eps, int t, int test)
    {
94.         double x=0, i=0, h=0, s=0, area=0, area1=0, diff=0, alfa=0;
95.         int CONT=0;
96.         h=(b-a)/n;
97.         for (i=0; i<n; i++){
98.             x=a+(i*h);
99.             s+=funzione(t,x);
100.        }
101.        area=s*h;
102.        if(test==1){
103.        do {
104.
105.            for (i=0; i<n; i++){
106.                x=a+(h/2)+(i*h);
107.                s=s+funzione(t,x);
108.            }
```

```
109.            h=h/2;
110.            area1=s*h;
111.            alfa=area1-area;
112.            diff=fabs(area1-area);
113.            n=n*2;
114.            area=area1;
115.            CONT+=1;
116.        } while (eps<diff);
117.        cout << "Rettangoli: " << CONT << " - ";
118.        }
119.    return area;
120.    }
121.
122.
123.double trapezi(double a, double b, int n, double eps, int t, int test){
124.        double x=0, i=0, h=0, s=0, area=0, area1=0, diff=0, alfa=0;
125.        int CONT=0;
126.        h=(b-a)/n;
127.        s=(funzione(t,a)+funzione(t,b))/2;
128.        x=a+h;
129.        for (i=1; i<n; i++){
130.            s=s+funzione(t,x);
131.            x+=h;
132.        }
133.        area=s*(h/2);
134.        if(test==1){
135.        do {
136.            x=a+(h/2);
137.            for (i=0; i<n; i++){
138.                s=s+funzione(t,x);
139.                x+=h;
140.            }
141.            area1=s*(h/2);
142.            h=h/2;
143.            alfa=area1-area;
144.            diff=fabs(alfa);
145.            n=n*2;
146.            area=area1;
147.            CONT+=1;
148.        } while (eps<diff);
149.        cout << "Trapezi: " << CONT << " - ";
150.        }
151.    return area;
152.    }
153.
154.double simpson(double a, double b, int n, double eps, int t, int test){
155.          double i=0, h=0, SD=0, SP=0, x=0, area=0, area1=0, diff=0,
   alfa=0;
156.        int CONT=0;
157.        if((n/2*2)!=n) ++n;
158.        h=(b-a)/n;
159.        x=a;
160.        for (i=1; i<n-2; i=i+2){
161.            x+=h;
162.            SD+=funzione(t,x);
163.            x+=h;
164.            SP+=funzione(t,x);
165.        }
166.        x+=h;
```

```
167.          SD+=funzione(t,x);
168.          area=(h/3)* (funzione(t,a) + 4*SD + 2*SP + funzione(t,b));
169.          if(test==1){
170.          do{
171.              x=a+(h/2);
172.              SP=SP+SD;
173.              SD=0;
174.              for(i=1; i<(n+1); i++){
175.                  SD=SD+funzione(t,x);
176.                  x+=h;
177.              }
178.              h=h/2;
179.              area1=(h/3)* (funzione(t,a) + 4*SD + 2*SP + funzione(t,b));
180.              alfa=area1-area;
181.              diff=fabs(alfa);
182.              n=n*2;
183.              area=area1;
184.              CONT+=1;
185.
186.          } while (eps<diff);
187.          cout << "Simpson: " << CONT;
188.          }
189.      return area;
190.      }
191.
192.      double funzione(int t, double d)
193.      {
194.           switch(t)
195.             {
196.              case 1:
197.                  return d*(exp(-d))*(cos(2*d));
198.                  break;
199.              case 2:
200.                  return 1./(1+pow(d,2));
201.                  break;
202.              case 3:
203.                  return d*sin(d);
204.                  break;
205.              case 4:
206.                  return exp(-(d*d));
207.                  break;
208.              case 5:
209.                  return exp(d);
210.                  break;
211.              case 6:
212.                  return 4;
213.                  break;
214.              case 7:
215.                  return 2*x+4;
216.                  break;
217.              case 8:
218.                  return pow(d,2)+4*x+2;
219.                  break;
220.              case 9:
221.                  return sin(1/d);
222.                  break;
223.          }
224.      }
```