Metodo Monte Carlo per l'integrazione approssimata

Ruslan Stasula

24 Agosto 2020

1 Introduzione

I metodi Monte Carlo sono tecniche euristiche ampiamente utilizzate in grado di risolvere una varietà di problemi comuni, inclusi problemi di ottimizzazione e integrazione numerica. Questi algoritmi funzionano campionando intelligentemente una distribuzione per simulare il funzionamento di un sistema. Le applicazioni vanno dalla risoluzione dei problemi di fisica teorica alla previsione delle tendenze negli investimenti finanziari. Dunque ci poniamo il problema di produrre un'implementazione dell'approssimazione di Monte Carlo per trovare la soluzione degli integrali di alcune funzioni. Ma vediamo prima alcuni algoritmi di approssimazione alternativi.

2 Integrazione in una dimensione

2.1 Formule di Newton-Cotes

Sono formule di quadratura interpolatorie, in cui i punti dove andiamo ad analizzare la funzione $x_0, ..., x_n$ sono prefissati nell'intervallo $[a, b]$. In questa classe si trovano le formule newtoniane o di Newton-Cotes, che si ricavano scegliendo i nodi x_i equidistanti. I coefficienti di queste formule sono facilmente ricavabili ed esprimibili con semplici numeri razionali. Queste formule hanno grado di precisione che varia tra n ed $n + 1$, ed il loro svantaggio è che per $n \geq 8$ i coefficienti non sono tutti dello stesso segno. Come accennato prima, bisogna prendere $n+1$ punti equidistanti su $[a, b]$. Poniamo dunque $h = (b-a)/n$, siano $x_i = a + ih, i = 0, ..., n$ i nodi equidistanti di passo h, sui quali costruiamo la formula di quadratura interpolatoria S_{n+1}, detta formula di $Newton - Cotes$ degli n+1 punti.

Determiniamo adesso i coefficienti, ma prima eseguiamo il cambiamento di variabile : $x = a + th$, $0 \leq t \leq n$. Si ha allora che $w_i = ha_i$ dove

$$a_i = \int_0^n \prod_{j=0, j\neq i}^{n} \frac{t-j}{i-j} dt$$

Per cui possiamo riscrivere la formula di Newton-Cotes nel seguente modo :

$$S_{n+1} = h \sum_{i=0}^{n} a_i f(x_i)$$

dove i coefficienti $a_i, i = 0, .., n$ ricavati in precedenza dipendono solo da i e n, ma non dai nodi. Sappiamo anche che, essendo i nodi simmetrici rispetto al punto centrale dell'intervallo $[a, b]$, i coefficienti a_i sono tali che $a_i = a_{n-i}$.

Proviamo ad esempio a calcolarci il seguente integrale: $\int_0^3 e^x dx$ con n=1 : Poniamo $x_0 = a$, $x_1 = b$, $h = b - a$, abbiamo che

$$a_0 = \int_0^1 (1 - t)dt = \frac{1}{2}$$

$a_1 = a_0$, poiche sono simmetrici. Da cui otteniamo la formula di quadratura dei due punti :

$$S_2 = \frac{h}{2}[f(x_0) + f(x_1)]$$

che applicata al nostro integrale di interesse, ci restituisce come risultato circa 31.62

Proviamo adesso con la formula dei 3 punti, quindi con $n = 2$:

$$S_3 = \frac{h}{3}[f(x_0) + 4f(x_1) + f(x_2)]$$

Il risultato ottenuto è : 12.78

Per $n = 3$ otteniamo la formula :

$$S_4 = \frac{3h}{8}[f(x_0) + 3f(x_1) + 3f(x_2) + f(x_3)]$$

con risultato : 19.28
E cosi via....

3 Integrazione con metodo Monte Carlo

E' un approccio completamente diverso, rispetto alle formule di quadratura interpolatorie viste precedentemente. Questa tecnica consiste nel simulare un processo statistico in cui il valore atteso del risultato è il valore dell'integrale di interesse. Presa una funzione f, definiamo la sua media come

$$< f >= \frac{1}{b-a} \int_a^b f(x)dx$$

e dunque

$$(b-a) < f >= \int_a^b f(x)dx$$

espandendo dunque la media di f e applicando il random sampling (campionamento casuale) , dove si prende un numero arbitrario di campioni (x_i), scelti in modo casuale nell'intervallo di interesse, e si valuta quindi la funzione, si ottiene la seguente approssimazione

$$(b-a)\frac{1}{n}\sum_i f(x_i) \approx \int_a^b f(x)dx$$

Se dunque $n \to \infty$, come risultato otterremmo esattamente l'integrale cercato. Quindi il metodo Monte Carlo consiste dei seguenti passi :
a) si generano n valori $x_1, ..., x_n$ random,che possiamo assumere essere dei valori di una variabile casuale uniformemente distribuita sull'intervallo $[a, b]$;
b) si calcola la media

$$S_n = \frac{b-a}{n}\sum_{i=1}^{n} f(x_i),$$

che si assume come approssimazione di S.

3.1 Applicazione

Poniamoci dunque nella situazione di dover calcolare :

$$\int_0^3 e^x dx$$

che dà come risultato e^3-1, cioè circa 19.08553692 .

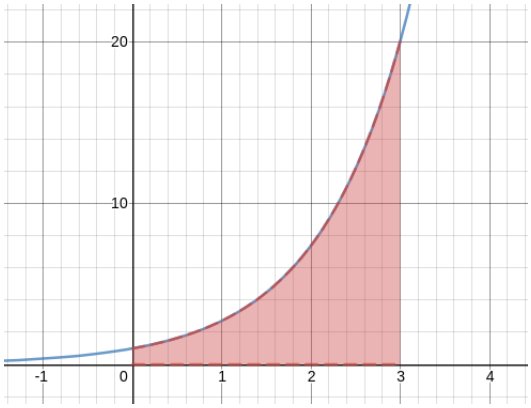

Figure 1: Intervallo da integrare

3.2 Analisi dell'errore

Ma quanto possiamo fidarci del risultato che otteniamo ? Come facciamo a sapere che un certo numero di campioni (o samples) è sufficiente per ottenere una buona approssimazione? Possiamo quantificare la nostra precisione trovando la varianza delle nostre stime. La varianza è definita come la misura di quanto le stime si discostano quadraticamente rispettivamente dalla media aritmetica. E' rappresentabile con questa equazione:

$$\sigma^2 =< f^2 > - < f >^2$$

La formula che usiamo per la varianza è quindi la seguente :

$$\sigma^2 = [\frac{1}{n}\sum_i^n f(x_i)^2] - [\frac{1}{n}\sum_j^n f(x_j)]^2$$

La varianza ci dà un'idea di quanto $f(x)$ varia nel dominio di x. Dovrebbe essere costante con il numero di campioni utilizzati, ma possiamo calcolare l'errore nell'integrazione prendendo la radice quadrata della varianza divisa per il numero di campioni :

$$\epsilon = \frac{\sigma}{n}.$$

Tuttavia questo non è un buon metodo per misurare l'errore. Immaginiamo di condurre diverse approssimazioni dell'integrale, con il risultato di ciascuno di questi uguale a I_n. Questi valori sono stati ottenuti con differenti sequenze di n numeri random. Per il teorema centrale del limite, questi valori sono normalmente distribuiti attorno ad una media $< I >$. Supponiamo di avere un insieme di M di questi calcoli I_n. Una misura più adatta delle differenze tra queste misurazioni è la "deviazione standard delle medie" σ_M :

$$\sigma_M^2 =< I^2 > - < I >^2$$

dove

$$< I >= \frac{1}{M} \sum_{k=1}^{M} I_k$$

e

$$< I^2 >= \frac{1}{M} \sum_{k=1}^{M} I_k^2$$

Può essere dimostrato che :

$$\sigma_M \approx \frac{\sigma}{\sqrt{n}}$$

Considerando che con un certo numero di samples, σ ci darà un valore abbastanza stabile, seppur calcolato in maniera "randomica" , ci accorgiamo subito che per far diminuire σ_M di un fattore 10, dobbiamo aumentare n di un fattore 100, dunque questo approccio non è buonissimo, se lavoriamo con funzioni a una sola variabile, ma diventa molto più competitivo appena aggiungiamo nuove variabili.

Svolgiamo dunque $M = 50$ approssimazioni del nostro integrale, e vediamo i valori che otteniamo facendo variare il numero dei campioni per ciasun calcolo da 10^1 a 10^8

n (numero campioni)	risultato(media per $M = 50$)	σ_M (media per $M = 50$)
10^1	22.491368182225617	1.528334152597439
10^2	18.77980676117449	0.5261055941658908
10^3	19.0074468424003	0.16316058816829743
10^4	19.195347741244557	0.05143383035207039
10^5	19.128326256477827	0.01629975624839596
10^6	19.084300215072364	0.005157381410925977
10^7	19.083365287990315	0.0016310313744982933
10^8	19.08572225773986	0.0005157228651129669

Ci accorgiamo del fenomeno che abbiamo descritto prima, ovvero, che incrementando il numero dei campioni di un fattore 100, otteniamo la diminuzione della nostra stima dell'errore di un fattore 10. Come si puo' ben notare, l'integrazione in una dimensione non e' il cavallo di battaglia del metodo Monte Carlo, ma lo sono gli integrali in più dimensioni.

4 Integrazione in più dimensioni

4.1 Formule di Newton-Cotes (Cavalieri-Simpson)

Proviamo adesso a calcolare il seguente integrale :

$$\int_0^5 \int_0^5 \int_0^5 \sqrt{x^5 + y^4 + z^3} dx dy dz$$

risultato del quale è circa : $2.715664340021269 * 10^3$
utilizzando le formule Newtoniane, in particolare la regola di Cavalieri-Simpson:

$$\int_a^b f(x)\,dx \approx \frac{\Delta x}{3}\left(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \cdots + 4f(x_{n-1}) + f(x_n)\right)$$

Notiamo che la formula sopra indicata riguarda una singola dimensione. Per prima cosa bisogna decidere quanto è ampio il passo per ciascuna variabile, cioè $h1, h2, h3$.
Essendo che questo metodo è molto povero in termini di prestazioni, preventivamente ci limiteremo ad utilizzare un numero di punti per ciascuna dimensione non superiore a 500, con un passo, per ciascuna variabile, proporzionale al numero massimo di punti accettabile.
Dobbiamo crearci una tabella che conterrà il valore della funzione calcolata per tutte le possibili combinazioni delle 3 variabili x, y, z.
Dopo aver generato la tabella (che in questo caso sarà tri-dimensionale), si applica la regola di Cavalieri-Simpson a ciascuna riga della tabella per calcolare l'integrale rispetto a due delle variabili tra (x, y e z), successivamente, nello step finale si esegue la computazione della risposta finale, sulla base dei due calcoli compiuti in precedenza.
Quindi, volta volta sfruttiamo la formula sopraindicata per eseguire il calcolo rispetto ad una dimensione sola, aggregando i risultati.
Tutti questi passi verranno messi in evidenza dal codice che implementa il metodo.

n (numero punti)	max. punti	risultato	ampiezza passo (h1=h2=h3)	tempo computazione (sec)
97	100	2692.7912264992196	0.052	0.8037734031677246
193	200	2692.791226905611	0.026	6.086061954498291
295	300	2709.931251141934	0.017	21.666809558868408
386	400	2717.595155655315	0.013	47.98643684387207
456	500	2719.504490615457	0.011	80.63611173629761

Notiamo come l'algoritmo sia instabile, in dipendenza dal numero dei punti che si adoperano per calcolare l'integrale (è una peculiarità delle formule di Newton-Cotes). Inoltre la performance in termini di memoria e di tempo com-

putazionale lascia molto a desiderare, considerando che viene utilizzato un numero piuttosto basso di punti per il calcolo.

4.2 Metodo Monte Carlo

Calcoliamo adesso lo stesso integrale :

$$\int_0^5 \int_0^5 \int_0^5 \sqrt{x^5 + y^4 + z^3} dx dy dz$$

Utilizzando la formula per l'integrazione con metodo Monte Carlo in piu' dimensioni. Sia da calcolare l'integrale :

$$S = \int_D f(X) dX$$

dove D e' una regione finita in R^r e dX è l'emenento di volume.
Sia P un parallelepipedo in r dimensioni, contenente D

$$P = \{X \in R^r : a_i \leq x_i \leq b_i, i = 1, ..., r\}, D \subseteq P$$

La funzione da integrare puo essere estesa in P assumendo $f(X) = 0$ per $X \in P - D$ e l'integrale diviene

$$S = \int_P f(X) dX$$

L'approssimazione del valore dell'integrale avviene nel modo seguente :
a) si generano n vettori $X_1, ..., X_n$ uniformemente distribuiti in P;
b) si calcola la media

$$S_n = \frac{1}{n} \prod_{i=1}^{r} (b_i - a_i) \sum_{j=1}^{n} f(X_j)$$

Che si assume essere l'approssimazione di S.
Otteniamo i seguenti risultati :

numero campioni	risultato	stima errore	tempo esecuzione (sec)
10^3	2747.1057477565314	0.4366510377331679	0.0016143321990966797
10^4	2710.3195639481396	0.14220480557163673	0.014485359191894531
10^5	2718.7924949326184	0.045600934909172795	0.1537175178527832
10^6	2715.622513395837	0.013889913284078837	1.3855562210083008
10^7	2715.577010525557	0.004479059919044291	14.083071231842041
10^8	2715.562531481497	0.0014548195871784341	146.02776193618774

Notiamo subito come, già in 3 dimensioni, il metodo Monte Carlo sia molto più conveniente e competitivo delle formule di Newton-Cotes. Ovviamente all'aumentare del numero delle dimensioni dell'integrale il divario si fa sempre più ampio.

5 Codice Python

Vediamo di seguito l'implementazione dei vari metodi utilizzati per l'approssimazione .

5.1 Monte Carlo in 1 dimensione

```
import numpy as np
import math
import random
from matplotlib import pyplot as plt
from IPython.display import clear_output

"""
Si usa numpy per trovare il minimo argomento di una lista,
math per definire le funzioni,
random per il campionamento casuale,
matplotlib per visualizzare graficamente i risultati ottenuti
"""

def rand_num(minv, maxv):
    """
    Questa funzione ritorna un numero casuale preso da una
    distribuzione
    uniforme di valori nell'intervallo  [minv, maxv]
    """
    range = maxv - minv
    choice = random.uniform(0, 1)
    return minv + range * choice


def fun(x):
    """
    funzione da integrare

    return math.exp(x)
    """
    return math.exp(x)

def monte_carlo(minv, maxv, num_samp=10000):
    """
    Questa funzione esegue il metodo Montecarlo alla funzione
    definita
    in precedenza, sull'intervallo di interesse, delimitato da minv
     e maxv
    """

    sum_samp = 0
    for i in range(num_samp):
        x = rand_num(minv, maxv)
        sum_samp += fun(x)
    return (maxv - minv) * float(sum_samp / num_samp)



```

```
47 def varianza_fun(minv, maxv, num_samp):
48     """
49     Questa funzione ritorna la varianza di f(x)
50     """
51     # si calcola la media dei quadrati
52     x = []
53     somma = 0
54     for i in range(num_samp):
55         x.append(rand_num(minv, maxv))
56         somma += fun(x[i]) ** 2
57     media_quadrati = somma / num_samp
58 
59     # gsi calcola il quadrato della media
60     somma = 0
61     for i in range(num_samp):
62         somma += fun(x[i])
63     quadrato_media = (somma / num_samp) ** 2
64     return media_quadrati - quadrato_media
65 
66 
67 def sigma_m(varianza, num_samp):
68     return math.sqrt(varianza) / math.sqrt(num_samp)
69 
70 def calc_avg(list):
71     sum_num = 0
72     for t in list:
73         sum_num = sum_num + t
74     avg = sum_num / len(list)
75     return avg
76 
77 
78 num_samp = 10
79 results = []
80 variances = []
81 sigmas = []
82 areas = []
83 while num_samp <= 100000000:
84     print(num_samp)
85     ris = []
86     var = []
87     sig_m = []
88     for i in range(0, 50):
89         tmp = monte_carlo(0, 3, num_samp)
90         ris.append(tmp)
91         areas.append(tmp)
92         var.append(varianza_fun(0, 3, num_samp))
93         sig_m.append(sigma_m(var[-1], num_samp))
94     results.append(ris)
95     variances.append(var)
96     sigmas.append(sig_m)
97     num_samp = num_samp * 10
98 avg_results = []
99 avg_variances = []
100 avg_sigmas = []
101 
102 
103 
```

```
for i in range(0, 8):
    avg_results.append(calc_avg(results[i]))
    avg_variances.append(calc_avg(variances[i]))
    avg_sigmas.append(calc_avg(sigmas[i]))

areas = [n for n in areas if n < 19.3 and n > 18.9]
_ = plt.title("Distribuzione delle aree calcolate")
_ = plt.hist(areas, bins=20, ec="black")
_ = plt.xlabel("Aree")
plt.show()
```

Breve descrizione

Vediamo brevemente le funzioni che sono state implementate.

- **rand_num**: funzione che serve ad ottenere un numero random nell'intervallo di integrazione per generare i campioni.
  Complessità : *O(1)*
- **fun**: funzione che calcola $f(x)$, dato un determinato x (ottenuto con rand_num).
  Complessità : *O(1)*
- **monte_carlo**: funzione che esegue il metodo Monte Carlo sommando tutti i valori di $f(x)$ calcolato nella variabile sum_samp. Dopo di che viene banalmente eseguito il calcolo finale della formula di S_n.
  Complessità : *O(n)*
- **varianza_fun**: funzione che calcola la varianza di $f(x)$ nell'intervallo di interesse, calcolando la media di $f(x_i)^2$, poi la media al quadrato di $f(x_i)$ e ritornando la differenza di questi valori.
  Complessità : *O(n)*
- **sigma_m** : funzione che calcola la stima dell'errore commesso : σ_M.
  Complessità : *O(1)*
- **calc_avg**: funzione che calcola la media dei valori contenuti in una lista, serve per condurre i test.
  Complessità : *O(elementi in lista)*

Veniamo adesso al codice per il testing . Si parte con num_samp $= 10$ in un ciclo while, fino a che (incrementando a ciascuna iterazione num_samp di un fattore 10) si arriva al massimo numero di campioni (n) che si vuole testare, per ciascuno di questi valori vengono effettuati $M = 50$ test, con chiamate a funzioni: monte_carlo, varianza_fun, sigma_m, si memorizzano i risultati di questi calcoli in delle liste, dopo di che vengono calcolate le medie (su 50 esperimenti) per ciascun valore di num_samp assunto, e vengono prodotti i rispettivi risultati.

Se plottiamo tutte le aree ottenute durante la computazione, tagliando fuori quelle troppo poco precise (ottenute approssimando l'integrale con un numero basso di campioni), otteniamo un grafico simile a quello di una distribuzione normale di valori, attorno a quello che è il risultato desiderato. Ovviamente più saranno i campioni analizzati, e più questo grafico sarà graduale e regolare.

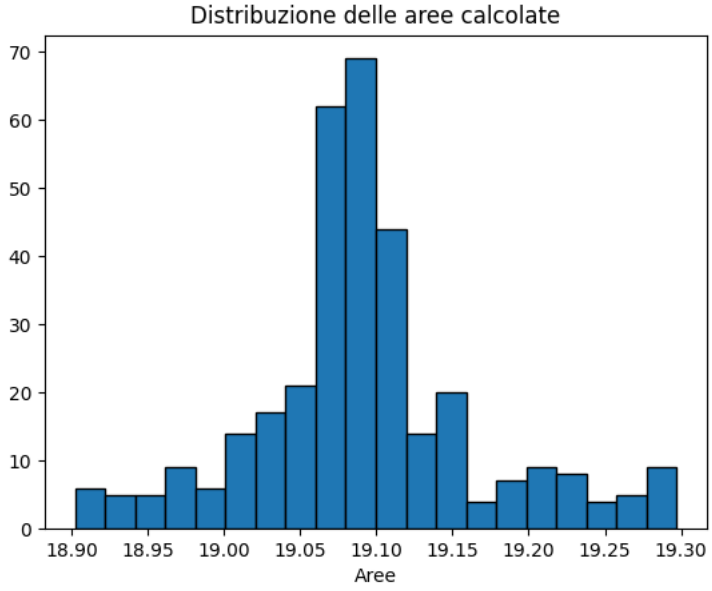

Figure 2: Distribuzione dei risultati

5.2 Metodo di Cavalieri-Simpson in 3 dimensioni

```
import time

# Funzione che vogliamo integrare
def givenFunction(x, y, z):

    return pow(pow(x, 4) + pow(y, 5) + pow(z, 3), 0.5)


# Funzione che calcola il valore dell'integrale triplo
def tripleIntegral(h1, h2, h3, lx, ux, ly, uy, lz, uz):
    start = time.time()
    # g contiene la tabella dei valori di f(x,y,z)
    g = [[[None for i in range(500)] for j in range(500)] for k in
    range(500)]
    ax = [None] * 500
    ay = [None] * 500

    # Calcola il numero dei punti
    nx = round((ux - lx) / h1 + 1)
    ny = round((uy - ly) / h2 + 1)
    nz = round((uz - lz) / h3 + 1)
    # Calcola i valori da inserire in tabella
    for i in range(0, nx):
        for j in range(0, ny):
            for k in range(0, nz):
                g[i][j][k] = givenFunction(lx + i * h1, ly + j * h2
    , lz + k * h3)

    # Calcola il valore dell'integrale rispetto a una variabile per
     volta
    for i in range(0, nx):
        ax[i] = 0
        for j in range(0, ny):
            ay[j] = 0
            for k in range(0, nz):

                if k == 0 or k == ny - 1:
                    ay[j] += g[i][j][k]
                elif k % 2 == 0:
                    ay[j] += 2 * g[i][j][k]
                else:
                    ay[j] += 4 * g[i][j][k]

            ay[j] *= h3 / 3

            if j == 0 or j == nx - 1:
                ax[i] += ay[j]
            elif j % 2 == 0:
                ax[i] += 2 * ay[j]
            else:
                ax[i] += 4 * ay[j]
        ax[i] *= h2 / 3

    answer = 0

```

```
    # Calcola il valore dell'integrale finale
    # usando gli integrali calcolati precedentemente
    for i in range(0, nx):
        if i == 0 or i == nx - 1:
            answer += ax[i]
        elif i % 2 == 0:
            answer += 2 * ax[i]
        else:
            answer += 4 * ax[i]

    answer *= h1 / 3

    end = time.time()

    return answer, end - start


# Driver Code
if __name__ == "__main__":

    # lx e ux sono i limiti inferire e superiore dell'integrale in
    x
    # ly e uy sono i limiti inferire e superiore dell'integrale in
    y
    # lz e uz sono i limiti inferire e superiore dell'integrale in
    z
    # h1 e' il passo per l'integrazione in x
    # h2 e' il passo per l'integrazione in y
    # h3 e' il passo per l'integrazione in z
    lx, ux = 0, 5
    ly, uy = 0, 5
    lz, uz = 0, 5
    h1 = 0.021
    h2 = 0.021
    h3 = 0.021

    print(tripleIntegral(h1, h2, h3, lx, ux, ly, uy, lz, uz))
```

Breve descrizione

- **givenFunction** : è la funzione di cui si vuole calcolare l'integrale.
  Complessità : *O(1)*

- **tripeIntegral** : funzione che calcola l'integrale in 3 dimensioni, creando una tabella tri-dimensionale, necessaria per contenere i valori assunti dalla funzione in tutti i punti (x, y e z) nei quali si va ad esaminare la funzione. Dopo di che vengono calcolati gli integrali in relazione ad una variabile per volta, usando la formula generica di Cavalieri-Simplson vista in precedenza. Infine viene calcolato il valore del risultato finale nella variabile answer, che viene poi restituito, insieme ai timestamp necessari per vedere il tempo di computazione dell'integrale.
  Complessità : $O(N^3)$

5.3 Metodo Monte Carlo in 3 dimensioni

```
1
2  import numpy as np
3  import math
4  import random
5  import time
6  from matplotlib import pyplot as plt
7
8  """
9  Si usa numpy per trovare il minimo argomento di una lista,
10 math per definire le funzioni,
11 random per il campionamento casuale,
12 matplotlib per visualizzare graficamente i risultati ottenuti
13 """
14
15
16 def rand_num(minv, maxv):
17     """
18     Questa funzione ritorna un numero casuale preso da una
       distribuzione
19     uniforme di valori nell'intervallo  [minv, maxv]
20     """
21     range = maxv - minv
22     choice = random.uniform(0, 1)
23     return minv + range * choice
24
25
26
27 def givenFunction(x, y, z):
28
29     return pow(pow(x, 4) + pow(y, 5) + pow(z, 3), 0.5)
30
31
32 def monte_carlo(lx, ux, ly, uy, lz, uz, num_samp=100000):
33     """
34     Questa funzione esegue il metodo Montecarlo alla funzione
       definita
35     in precedenza, sull'intervallo di interesse, delimitato da minv
        e maxv
36     """
37     start = time.time()
38     sum_samp = 0
39
40     for i in range(num_samp):
41         x = rand_num(0, 5)
42         y = rand_num(0, 5)
43         z = rand_num(0, 5)
44         sum_samp += givenFunction(x, y, z)
45
46     print(sum_samp)
47     end = time.time()
48     return 125 * (sum_samp / num_samp)
49
50
51
52
```

```
53
54
55  def varianza_fun(lx, ux, ly, uy, lz, uz, num_samp=1000):
56
57      """
58      Questa funzione ritorna la varianza di f(x)
59      """
60
61      # si calcola la media dei quadrati
62      x = []
63      y = []
64      z = []
65      somma = 0
66      for i in range(num_samp):
67          x.append(rand_num(lx, ux))
68          y.append(rand_num(ly, uy))
69          z.append(rand_num(lz, uz))
70          somma += givenFunction(x[i], y[i], z[i]) ** 2
71      media_quadrati = somma / num_samp
72
73
74
75      # si calcola il quadrato della media
76      somma = 0
77      for i in range(num_samp):
78          somma += givenFunction(x[i], y[i], z[i])
79      quadrato_media = (somma / num_samp) ** 2
80
81      return media_quadrati - quadrato_media
82
83
84
85
86  def sigma_m(varianza, num_samp):
87      return math.sqrt(varianza) / math.sqrt(num_samp)
88
89
90
91
92  def calc_avg(list):
93      sum_num = 0
94      for t in list:
95          sum_num = sum_num + t
96      avg = sum_num / len(list)
97      return avg
98
99
100
101
102
103
104
105
106
107
108
109
```

```
#TESTING
lx, ux = 0, 5
ly, uy = 0, 5
lz, uz = 0, 5

num_samp = 10
results = []
variances = []
sigmas = []
areas = []  # lista contenente tutte le aree calcolate nel corso
    della computazione
while num_samp <= 100000000:
    print(num_samp)
    ris = []
    var = []
    sig_m = []
    for i in range(0, 50):
        tmp = monte_carlo(lx, ux, ly, uy, lz, uz, num_samp)
        print(tmp)
        ris.append(tmp)
        areas.append(tmp)
        var.append(varianza_fun(lx, ux, ly, uy, lz, uz))
        sig_m.append(sigma_m(var[-1], num_samp))
    results.append(ris)
    variances.append(var)
    sigmas.append(sig_m)
    num_samp = num_samp * 10
avg_results = []
avg_variances = []
avg_sigmas = []
for i in range(0, 8):
    avg_results.append(calc_avg(results[i]))
    avg_variances.append(calc_avg(variances[i]))
    avg_sigmas.append(calc_avg(sigmas[i]))
print(avg_results, avg_sigmas, avg_variances)
areas = [n for n in areas if n < 2730 and n > 2700]
print("areas")
print(areas)
_ = plt.title("Distribuzione delle aree calcolate")
_ = plt.hist(areas, bins=20, ec="black")
_ = plt.xlabel("Aree")
plt.show()
```

Breve descrizione

L'implementazione del metodo Monte Carlo in 3 dimensioni, è pressoche' uguale a quella in una dimensione sola, quindi con bassa complessità del codice, rendendolo sicuramente la scelta preferibile per l'approssimazione degli integrali in più dimensioni. Di seguito il grafico della distribuzione delle aree calcolate.

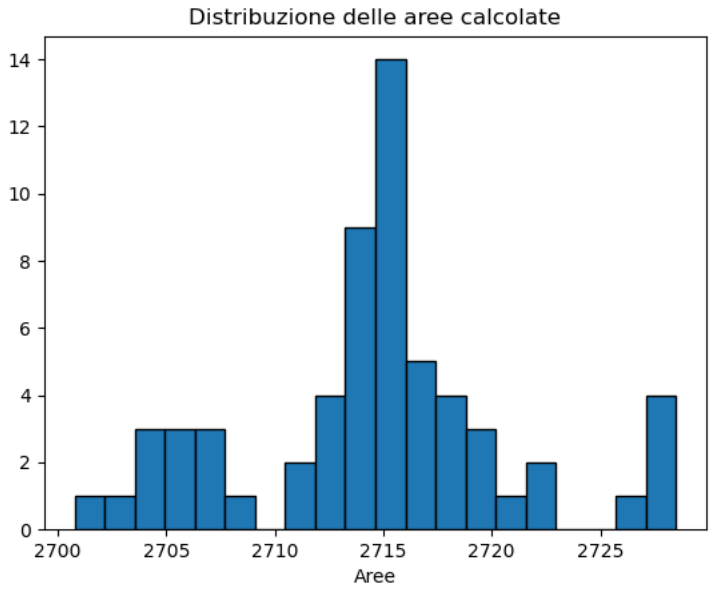

Figure 3: Distribuzione dei risultati

References

[1] Stefano Berrone ,*Quadratura Numerica*, 2011
http://calvino.polito.it/~sberrone/Faculty/01ILRFW.2011/4_Quadratura.pdf

[2] Integrazione Numerica
http://www.bplab.bs.unicatt.it/~jovanni/file_3/file3.html

[3] Newton–Cotes formulas [Wikipedia]
https://en.wikipedia.org/wiki/Newton-Cotes_formulas

[4] Monte Carlo method
https://www.britannica.com/science/Monte-Carlo-method

[5] Monte Carlo integration [Wikipedia]
https://en.wikipedia.org/wiki/Monte_Carlo_integration

[6] Monte Carlo Methods in Practice
https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice/monte-carlo-integration

[7] Simpson's Rule [Wikipedia]
https://en.wikipedia.org/wiki/Simpson's_rule

[8] Venelin Todorov, Ivan Dimov, *Monte Carlo methods for multidimensional integration for European option pricing* , [Conference Paper] Ottobre 2016