计算方法（A）

实验（三） 实现龙贝格积分

姓　　名	熊恪峥
学　　号	22920202204622
日　　期	2022年4月6日
学　　院	信息学院
课程名称	计算方法（A）

实验（三） 实现龙贝格积分

目录

1 实现

龙贝格积分法是通过外推法不断使用梯形公式来提高计算精度的数值积分法。是一种常用的积分法，并且可以达到较高的精度。

1.1 文件结构

本程序实现了龙贝格积分以及用于基准测试和比较的蒙特卡洛积分法和拟蒙特卡洛（Quasi-Monte Carlo）积分法。主要文件和作用如表格 1.1所示。

文件	作用
integral/romberg.py	龙贝格积分实现
integral/qmc.py	拟蒙特卡洛积分法实现
integral/montecarlo.py	蒙特卡洛积分法实现

表格 1: 主要文件和作用

本程序依赖如下两个第三方库：

- **Numpy**: 用于加速向量运算
- **Scipy**: 用于生成低差异序列（Low-discrepancy sequence）

1.2 接口实现

本程序为龙贝格积分法实现以下接口，它支持通过给定任意可调用对象（Callable）表示的函数进行积分，具有良好的可扩展性。

```
1 def integrate(f: Callable[[np. float], np. float], a: np. float, b: np. float, epsilon: np.
      float = 0.001,
2               n:  int = 32) -> np. float:
```

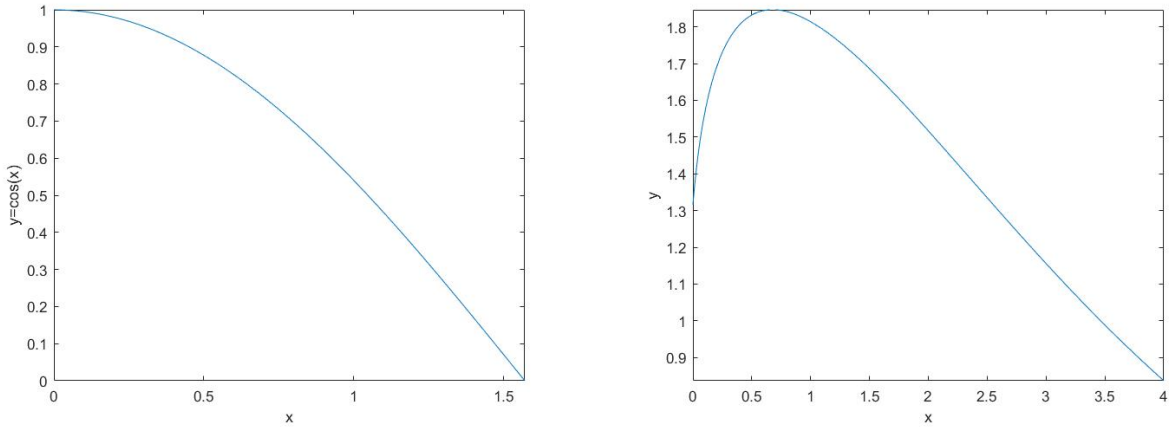
该接口有如下参数：

- **f**: 被积函数的可调用对象。
- **a, b**: 表示积分区间$[a, b]$。
- **epsilon**: 当两次迭代数值变化$\Delta T \leq \varepsilon$时停止迭代，可以控制积分精度。
- **n**: 最大迭代次数。

2 基准测试

2.1 积分准确度和收敛速度

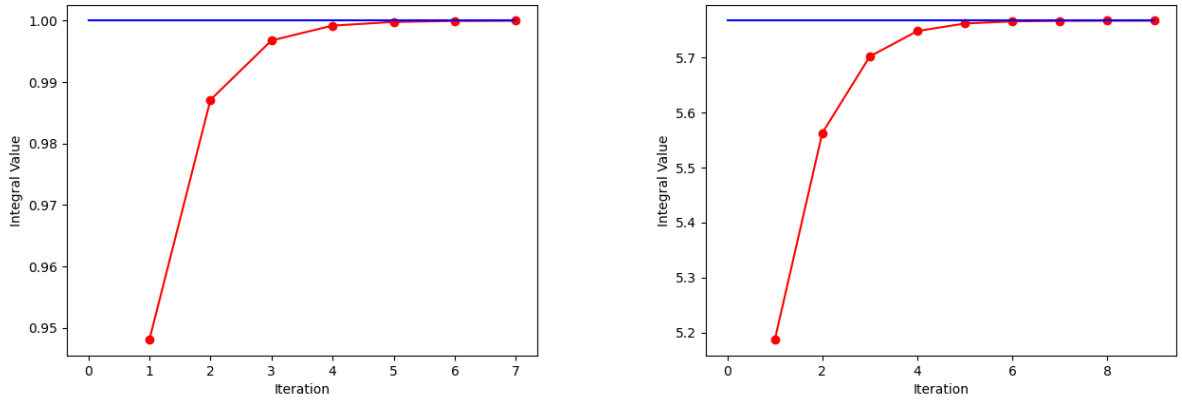
为了测试龙贝格积分法的效果，以Matlab 2021b的积分值为真值、蒙特卡洛积分法和拟蒙特卡洛积分法为基线进行测试，计算积分$\int_0^{\frac{\pi}{2}} \cos x \, \mathrm{d}x$和$\int_0^4 \sqrt[4]{15x^3 + 21x^2 + 41x + 3} \cdot e^{-0.5x} \, \mathrm{d}x$。被积函数的图形如图 1



(a) 函数$f(x)=\cos x$ (b) 函数$g(x)=(15x^3+21x^2+41x+3)^{\frac{1}{4}}\cdot e^{-0.5x}$

图 1: 被积函数图象

调用本程序实现的龙贝格积分法函数，要求精度0.0001，迭代次数最多128次，可以得到每次迭代的积分值如图 2。可以发现龙贝格积分法的收敛速度很快，约3至4次迭代即可得到和真实值相近的积分值。



(a) 积分$\int_0^{\frac{\pi}{2}} f(x)\,\mathrm{d}x$迭代过程 (b) 积分$\int_0^4 g(x)\,\mathrm{d}x$迭代过程

图 2: 被积函数图象

龙贝格积分法积分得出的数据如表 2所示。在迭代次数10以内的条件下积分值的精确度已经达到了极高的水平，相对误差对于两个不同的函数均小于0.001%。这说明了龙贝格法有良好的精度。

函数	测量值	参考值（以Matlab 2021b为基准）	相对误差(%)
$f(x)$	0.9999874501175262	1	0.0012
$g(x)$	5.767412519408859	5.767433490695931	0.00036

表格 2: 龙贝格积分法积分值

2.2 运行效率和比较

平均运行时间用于测量该实现的运行效率，通过1000次循环取平均值来规避偶然波动。结果如图 2.2。可以发现龙贝格积分法的运行速度在函数本身不是非常复杂，但是当函数求值较为复杂时，龙贝格积分法在效率上比其他两者较慢。其中的一个原因可能是龙贝格法每轮迭代中函数求值发生的时间没有较强的时间上的

局部性。蒙特卡洛法和拟蒙特卡洛法对函数求值会在采样点得出之后一次完成，而龙贝格法会在每轮迭代中多次发生。

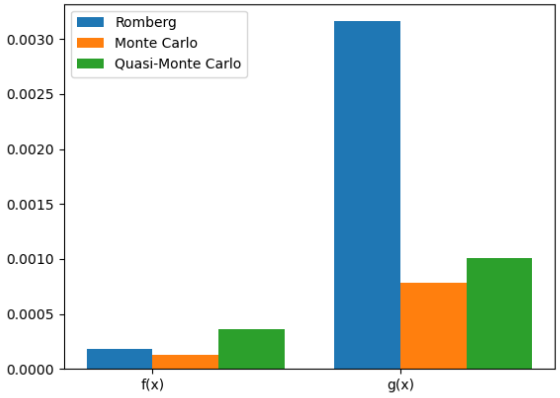
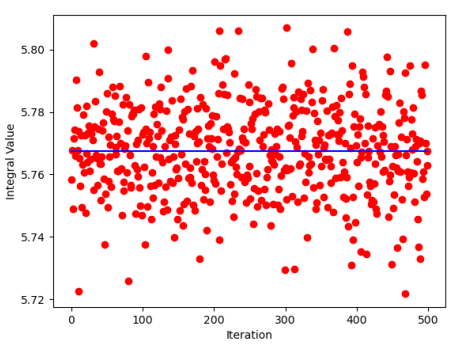


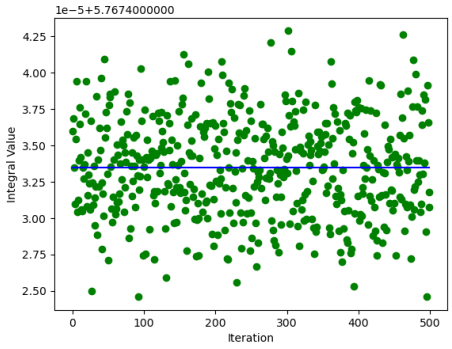
图 3: 平均运行时间

2.3　蒙特卡洛和拟蒙特卡洛法的性能

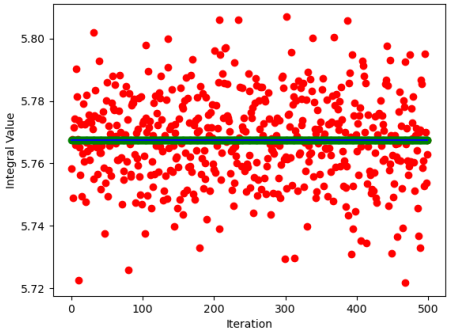
但是由于运行时间的差异实际上并不大，但这并不能否认龙贝格积分法是一个优秀的数值积分法。因为龙贝格积分发可以给出一个确定的解，它在任何情况下多次调用都是稳定的。如图 4 蒙特卡洛法具有一定的随机性，每次调用的值都分布在实际值附近，但是有所波动。



(a) 蒙特卡洛法500次分布



(b) 拟蒙特卡洛法500次分布



(c) 同坐标系显示

图 4: 蒙特卡洛和拟蒙特卡罗的分布情况

值得一提的是采用拉丁超立方采样的拟蒙特卡洛积分法具有相当小的波动和相当好的准确度，如图 4(c)。和普通的蒙特卡洛法相比，该方法的分布和真实值相当接近，几乎没有误差。因此这实际上是一个非常常用的数值积分法。这两类随机方法的均值和方差如表 2.3，拟蒙特卡洛积分法的方差达到了10^{-11}级别，证明了几乎没有显著的波动。

算法	均值	方差
蒙特卡洛法	5.766607543456825	0.00018877067621830997
拟蒙特卡洛法	5.767433680428228	$1.0746874157286956 \times 10^{-11}$

表格 3: 随机方法的均值与方差

3 结论

本次实验实现了龙贝格积分法，并将它与常用的蒙特卡洛法和拟蒙特卡洛法测试和对比，并简要分析了其中的原因。

龙贝格积分法是一种迭代地逐次提高精度的确定性的方法。它的精度是可以通过提高迭代次数等方法提高。它迭代提高精度的方式是将误差的一部分补进积分值。经过实验，这种方法在大多数情况下都有较好的效率，并且在得出确定性结果的同时达到指定精度。

导致龙贝格积分法在某些条件下相对其他两者效率一般的原因可能是因为这个递推过程很难被向量化优化，尤其是对被积函数求值的过程在时间上的局部性较差。这导致了该实现没有有效利用现代硬件的优化。但是这造成的性能差异并不显著，并且有可能通过微调算法优化。可见龙贝格积分法是一种非常高效的算法。

尽管在为了达到与蒙特卡洛积分法特别是使用了拉丁超立方采样的拟蒙特卡洛积分法可以比拟的精度，龙贝格积分法需要较多的迭代次数从而影响了运行效率，但是考虑到拉丁超立方采样在多变量积分和图形渲染领域的大量应用侧面证明的高效性和准确性，这是正常的。

A 附录：蒙特卡洛积分法和拟蒙特卡洛积分

使用蒙特卡洛积分是一种采用蒙特卡洛法估计积分的方法。设有积分(1)

$$I = \int_a^b f(x) d\,\mathrm{d}x \tag{1}$$

那么可以使用(2)估计这个积分。其中$pdf(X)$是概率密度函数。特别地在实现中常取均匀分布$X \sim \mathcal{U}(a,b)$，则$pdf(x) = 1/(b-a)$。

$$\bar{I_n} = F_n(X) = \frac{1}{n}\sum_{k=1}^{n}\frac{f(X_k)}{pdf(X_k)} \tag{2}$$

Proof. 要证明这个估计是正确的，就需要证明该估计的数学期望和积分值相等。

$$\begin{aligned} E[F_n] &= E\left[\frac{1}{n}\sum_{k=1}^{n}\frac{f(X_k)}{pdf(X_k)}\right] \\ &= \frac{1}{n}\sum_{k=1}^{n}\int \frac{f(x)}{pdf(x)}\cdot pdf(x)\,\mathrm{d}x \\ &= \frac{1}{n}\sum_{k=1}^{n}\int f(x)\,\mathrm{d}x \\ &= \int f(x)\,\mathrm{d}x \end{aligned}$$

□

根据以上结论，可以得到算法 1

算法 1 蒙特卡洛法积分

Input: 函数F，区间$[a,b]$，点数n

```
procedure INTEGRATE(F, a, b, n)
    X = [x_1, x_2, x_3, ...], x_i ~ U(a, b)
    Y = F(X)
    return (b − a) (1/n) Σ_{i=1}^{n} Y_i
```

作为一种随机算法，实际上在n较大时有较好的精度，它的收敛速度是$\mathcal{O}(n^{-0.5})$ 如果采用低差异序列替换纯随机数，积分收敛速度能够达到$\mathcal{O}(n^{-1})$，而且能够避免随机采样中采样点的聚集。一种简单的一维低差异序列是*Van Der Corput*序列。设有b进制数 $x = \sum_{k=0}^{m} d_k \cdot b^k$，则对应的Van der Corput序列可由(3)计算。

$$g_b(x) = \sum_{k=0}^{m} d_k \cdot b^{-k-1} \tag{3}$$

$g_b(x)$满足$g_b(x) \in [0,1]$，因此通过区间映射并替代算法 1中随机数可以得到拟蒙特卡洛积分算法。此外，还可以使用Latin Hypercube Sampling（拉丁超立方采样）来替代随机数。

B 附录：代码

B.1 龙贝格积分实现

代码 1: 龙贝格积分实现

```python
import numpy as np

from typing import Callable


def integrate(f: Callable[[np. float], np. float], a: np. float, b: np. float, epsilon: np. float = 0.001,
              n:  int = 32) -> np. float:
    r: np.ndarray = np.zeros((n + 1, n + 1), dtype= float)
    h: np. float = b - a
    r[0, 0] = 0.5 * h * (f(a) + f(b))

    p2:  int = 1
    for k in  range(1, n + 1):
        h *= 0.5
        p2 *= 2

        r[k, 0] = 0.0
        for i in  range(1, p2, 2):
            r[k, 0] += f(a + i * h)
        r[k, 0] *= h
        r[k, 0] += 0.5 * r[k - 1, 0]

        p4 = 1
        for m in  range(1, k + 1):
            p4 *= 4
            r[k, m] = r[k, (m - 1)] + (r[k, (m - 1)] - r[k - 1, (m - 1)]) / (p4 - 1)

        if k != 1 and np. abs(r[k, 0] - r[(k - 1), 0]) < epsilon:
            return r[k, 0]

    return r[n, 0]
```

B.2 蒙特卡洛法和拟蒙特卡罗法实现

代码 2: 龙贝格积分实现

```python
import numpy as np
from scipy.stats import qmc

from typing import Callable
def integrate(f: Callable, a: np. float, b: np. float, n:  int = 32) -> np. float:
    x = qmc.scale(qmc.LatinHypercube(d=1).random(n), a, b)
    y = f(x)
    y_mean = np. sum(y) / n
    return (b - a) * y_mean


def integrate(f: Callable, a: np. float, b: np. float, n:  int = 32) -> np. float:
```

```
    x = np.random.uniform(a, b, n)
    y = f(x)
    y_mean = np. sum(y) / n
    return (b - a) * y_mean
```

代码 3: 基准测试代码

```
import timeit

import numpy as np

from integral.romberg import integrate as i1
from integral.montecarlo import integrate as i2
from integral.qmc import integrate as i3

import matplotlib.pyplot as plt
from integral.romberg import integrate2


def f(x):
    return np.cos(x)


def f2(x):
    return np.power(15 * np.power(x, 3) + 21 * np.power(x, 2) + 41 * x + 3, 1.0 / 4.0) * np.exp(-0.5
        * x)


def test_cov(ff, real, a, b):
    val, iters, vals = integrate2(ff, a, b, 0.0001, 128)

    print(val)
    plt.plot(iters, vals, 'ro-', [0,  len(iters)], [real, real], 'b')
    plt.xlabel('Iteration')
    plt.ylabel('Integral Value')
    plt.show()


def test_speed():
    rm = []
    mc = []
    qmc = []

    start = timeit.default_timer()
    for i in  range(1, 1000):
        i1(f, 0, np.pi / 2, 0.0001, 128)
    rm.append((timeit.default_timer() - start) / 1000)

    start = timeit.default_timer()
    for i in  range(1, 1000):
        i1(f2, 0, 4, 0.0001, 128)
    rm.append((timeit.default_timer() - start) / 1000)

    start = timeit.default_timer()
    for i in  range(1, 1000):
```

```
        i2(f, 0, np.pi / 2, 8192)
    mc.append((timeit.default_timer() - start) / 1000)

    start = timeit.default_timer()
    for i in  range(1, 1000):
        i2(f2, 0, 4, 8192)
    mc.append((timeit.default_timer() - start) / 1000)

    start = timeit.default_timer()
    for i in  range(1, 1000):
        i3(f, 0, np.pi / 2, 8192)
    qmc.append((timeit.default_timer() - start) / 1000)

    start = timeit.default_timer()
    for i in  range(1, 1000):
        i3(f2, 0, 4, 8192)
    qmc.append((timeit.default_timer() - start) / 1000)

    x = np.arange(2)
    total_width, n = 0.8, 3
    width = total_width / n
    x = x - (total_width - width) / 2

    tick_labels = ["f(x)", "g(x)"]
    plt.bar(x, rm, width=width, label='Romberg')
    plt.bar(x + width, mc, width=width, label='Monte Carlo')
    plt.bar(x + 2 * width, qmc, width=width, label='Quasi-Monte Carlo')
    plt.legend()
    plt.xticks(x + width / 2, tick_labels)
    plt.show()


def test_variance():
    iters = []
    vals1 = []
    vals2 = []
    for i in  range(1, 500):
        iters.append(i)
        vals1.append(i2(f2, 0, 4, 8192))
        vals2.append(i3(f2, 0, 4, 8192))

    print(np.mean(np.array(vals1, dtype= float)))
    print(np.var(np.array(vals1, dtype= float)))
    print(np.mean(np.array(vals2, dtype= float)))
    print(np.var(np.array(vals2, dtype= float)))

    real = 5.767433490695931
    plt.plot(iters, vals1, 'ro', iters, vals2, 'go', [0,  len(iters)], [real, real], 'b')
    plt.xlabel('Iteration')
    plt.ylabel('Integral Value')
    plt.show()

    plt.plot(iters, vals1, 'ro', [0,  len(iters)], [real, real], 'b')
    plt.xlabel('Iteration')
    plt.ylabel('Integral Value')
```

```
103     plt.show()
104
105     plt.plot(iters, vals2, 'go', [0,  len(iters)], [real, real], 'b')
106     plt.xlabel('Iteration')
107     plt.ylabel('Integral Value')
108     plt.show()
109
110 test_variance()
111
112 start = timeit.default_timer()
113 for i in  range(1, 1000):
114     i1(f2, 0, 4, 0.0001, 128)
115 print(timeit.default_timer() - start)
116
117 start = timeit.default_timer()
118 for i in  range(1, 10):
119     i2(f, 0, np.pi / 2, 8192)
120 print(timeit.default_timer() - start)
121
122 start = timeit.default_timer()
123 for i in  range(1, 1000):
124     i3(f2, 0, 4, 8192)
125 print(timeit.default_timer() - start)
126
127 print(i1(f, 0, np.pi / 2, 0.0001, 128),
128       i2(f, 0, np.pi / 2, 16384),
129       i3(f, 0, np.pi / 2, 8192))
```