实验四 数值积分

一、实验目的

① 体会数值积分的基本概念；
② 掌握低阶的插值型数值积分公式；
③ 掌握区间逐次分半的复化求积方法；
④ 掌握龙贝格算法的基本思路和迭代步骤；

二、实验过程和结果

1. Newton-Cotes 求积公式

Newton-Cotes 公式求积分是将积分区间 n 等分，得到 n + 1 个点，然后把积分估计为：

$$\int_a^b f(x)dx \approx (b-a)\sum_{k=0}^{n} C_k^{(n)} f(x_i)$$

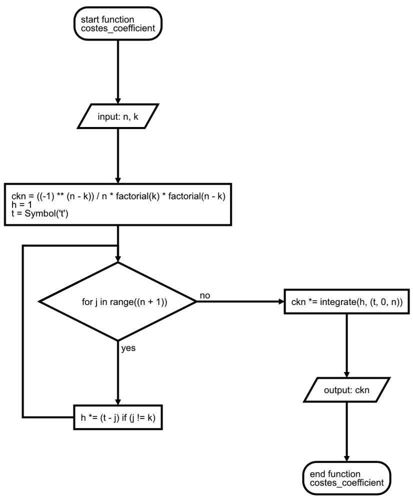
其中的 $C_k^{(n)}$ 为科特斯系数，其计算方法为：

$$C_k^{(n)} = \frac{(-1)^{n-k}}{nk!(n-k)!}\int_0^n \prod_{j=0,j\neq k}^{n} (t-j)dt$$

这个算法容易编程实现[1]，如【图1】所示。下面是调用该算法打印出的一张科特斯系数表：

```
for i in range(6):
    for j in range(i+2):
        print(costes_coefficient(i+1, j), end= "\t ")
    print()
```

```
1/2     1/2
1/6     2/3     1/6
1/8     3/8     3/8     1/8
7/90    16/45   2/15    16/45   7/90
19/288  25/96   25/144  25/144  25/96   19/288
41/840  9/35    9/280   34/105  9/280   9/35    41/840
```



【图1】求柯特斯系数的程序流程图

[1] 该算法的具体代码实现可以从 https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex4/src/newton_cotes.py 获取

实现了科特斯系数的计算后，通用的 Newton-Cotes 公式也就容易编程实现了[2]：

```
def newton_cotes_integral(f, a, b, n):
    step = (b - a) / n
    xs = [a + i * step for i in range(n+1)]
    return (b - a) * sum([
        costes_coefficient(n, k) * f(xs[k])
        for k in range(0, n+1)
    ])
```

当 n 为奇数时，这种方法至少有 n 阶代数精度；当 n 为偶数时，至少有 n+1 阶代数精度。

特殊地，当 Newton-Cotes 公式取 $n=1$，就是梯形求积公式：

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

编程实现：

```
def trapezium_integral(f, a, b):
    return (b - a) * (f(a) + f(b)) / 2
```

取 $n=2$，就是辛普森求积公式：

$$\int_a^b f(x) \approx (b-a)\frac{f(a)+4f(\frac{a+b}{2})+f(b)}{6}$$

编程实现：

```
def simpson_integral(f, a, b):
    return (b - a) * (f(a) + 4 * f((a + b) / 2) + f(b)) / 6
```

[2] 实现源文件见：https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex4/src/newton_cotes.py

2. 复化求积公式

求积区间比较大的时候，如果使用节点数很少的 Newton-Cotes 公式会有较大的截断误差；而分很多节点时， Newton-Cotes 公式不具备数值稳定性，即便可行，科特斯系数的计算需要 $n!$ 级的数作为分母，点数 n 增多时也会带来计算上的麻烦以及可能的舍入误差。

一种解决的方法是把积分区间分成若干个小的区间，在每个区间上各自用低次的 Newton-Cotes 公式进行计算，最后把所有结果求和。这就得到了**复化求积公式**。为了避开决定分多少个小区间的问题，可以采用区间逐次半分的算法来实现复化求积。

在具体编程实现[3]:

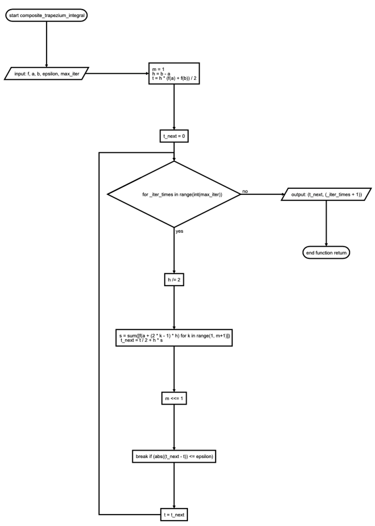
- 复化梯形公式:

`composite_trapezium_integral(f, a, b, epsilon, max_iter=10000)`

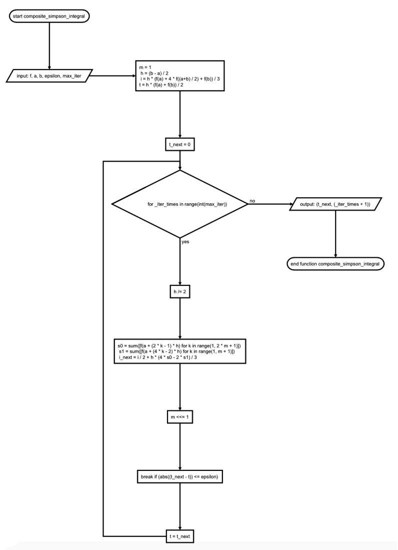
- 复化 Simpson 公式:

`composite_simpson_integral(f, a, b, epsilon, max_iter=1e6)`

这两个程序（函数）都需要输入要求积的函数 f，求积区间 $[a,b]$，以及目标精度 ϵ 和最大迭代次数作为参数。运行结束后返回最终得到的积分值 i 以及迭代次数，当无法在指定最大迭代次数内达到指定精度时，抛出错误。



【图2】复化梯形公式程序流程图



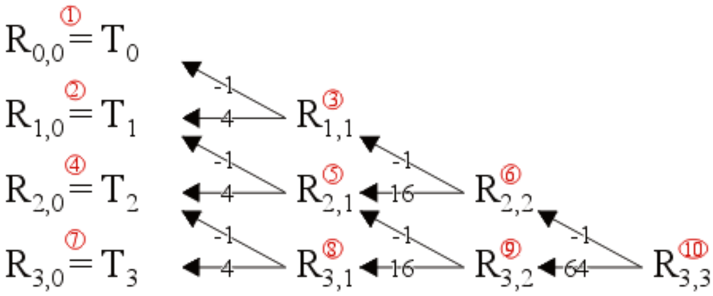
【图3】复化Simpson公式程序流程图

[3] 源代码见：https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex4/src/composite.py

3. 龙贝格算法

Romberg 算法是一种收敛速度很快的求积方法。其计算过程是将区间逐次分半，加速得到积分近似值。

【图4】展示了这种 Romberg 计算的过程，其中 $R_{n,m} = \frac{1}{4^m - 1}(4^m R_{n,m-1} - R_{n-1,m-1})$。



Copyright ©2005 by Douglas Wilhelm Harder
from ece.uwaterloo.ca

【图4】Romberg计算过程

编程实现[4]：

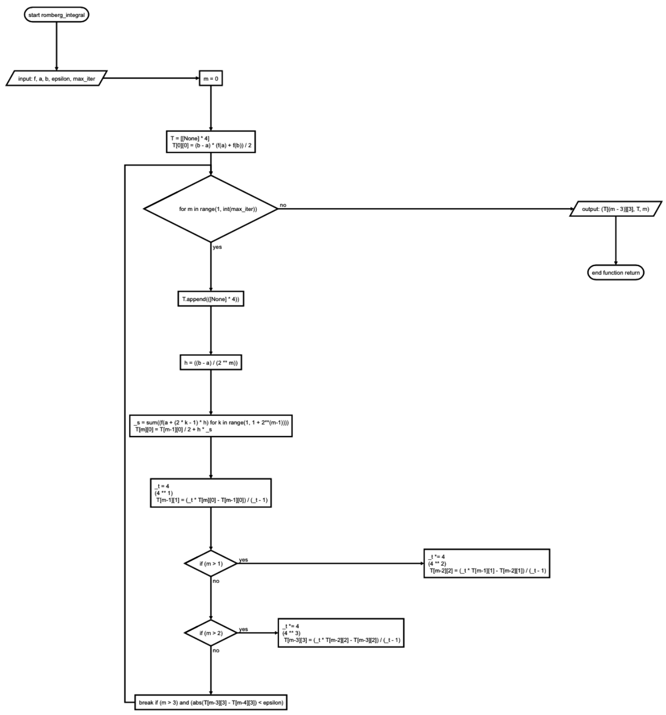
```
romberg_integral(f, a, b,
    epsilon, max_iter=1e6)
```

输入参数：

- `f`: 要求积的函数；
- `a, b`: 求积区间；
- `epsilon`: 目标精度，达到则停止，返回积分值；
- `max_iter`: 最大迭代次数，超出这个次数迭代不到目标精度，则抛出错误。

返回结果： `(result, T, iter)`

- `result`: 最终得到的积分值
- `T`: 计算过程表
- `iter_times`: 迭代次数



【图5】Romberg积分程序流程图

这个实现中有一个小的细节问题。这个算法中的 T 记录下了完整计算过程中的所有值，也就是说，这个算法的空间复杂度为 $O(n)$。如果迭代次数十分大，则这里会不必要地消耗大量内存（虽然一般 Romberg 算法都收敛地很快，基本可以不考虑这个问题）。

为了避免这种内存浪费，我用**滑动窗口**的方法改写了实现过程，得到一个 `romberg_integral_sw` 函数，这个算法只使用一个滑动窗口来动态储存需要的值，后续计算中不再需要的值将被直接丢弃，这样可以把内存开销控制在 $O(1)$ 级别。

[4] 具体的代码实现存放在 https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex4/src/romberg.py

4. 自适应 Simpson 求积法[5]

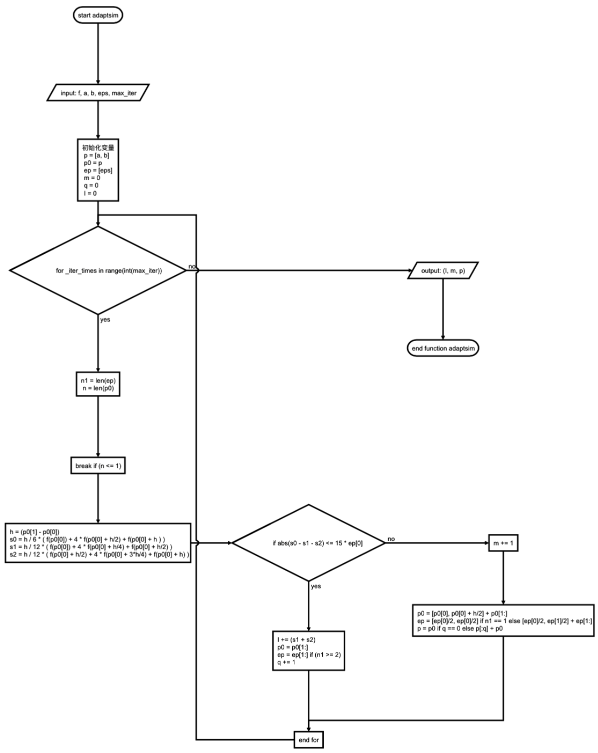
使用 Newton-Cotes 系列的方法求积法时的一个问题是，节点的选择。点分得越多，误差越小，但计算量也越大；分得太少，计算上方便，但误差就会增大。通常我们无从得知分成多少段合适，所以引入自适应求积方法，按照被积函数的变化性态来安排节点。

关于“按照被积函数的变化性态来安排节点”，我的理解是，如果求积中的一段曲线如果近似于 Simpson 法“拟合”被积曲线的二次函数，就直接把这一段用 Simpson 法求出来；如果不够近似，就把这个区间分成两半，递归这个过程去求积。我个人更喜欢这种递归实现，代码十分十分清晰简洁，如【图8】所示。

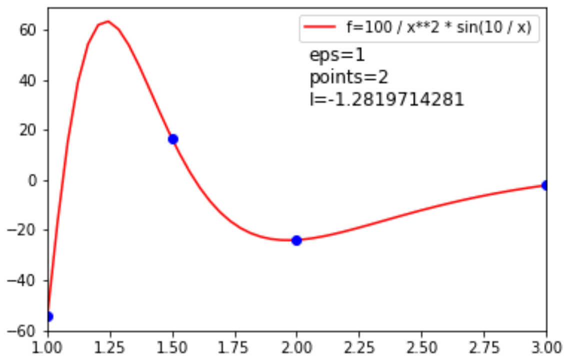
当然也可以用迭代实现自适应 Simpson 求积法（参考课本的实现，程序流程见【图6】）：

```
adaptsim(f, a, b, eps=1e-8,
max_iter=10000)
```

用实现的程序计算积分 $I=\int_1^3 \frac{100}{x^2} sin(\frac{10}{x})$，取不同的精度 eps，得到结果如【图7】[6]：



【图6】自适应 Simpson 求积法程序流程图



【图7】自适应 Simpson 求积结果（动图）

```
def asr(f, a, b, eps=1e-8):
    def simpson(f, a, b):
        return (b - a) * (f(a) + 4 * f((a + b) / 2) + f(b)) / 6

    def asrp(f, a, b, eps, sim):
        mid = (a + b) / 2

        L = simpson(f, a, mid)
        R = simpson(f, mid, b)

        if (abs(L + R - sim) <= eps):
            return L + R + (L + R - sim) / 15

        return asrp(f, a, mid, eps/2, L) + asrp(f, mid, b, eps/2, R)

    return asrp(f, a, b, eps, simpson(f, a, b))
```

【图8】自适应 Simpson 的递归实现源码

---

[5] 自适应 Simpson 求积法实现源码：https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex4/src/adaptsim.py

[6] 具体的计算、绘图源码：https://github.com/cdfmlr/NumericalAnalysis/blob/master/ex4/src/ex4.ipynb

5. 实验内容题目

I. 计算积分 $I_1 = \int_0^1 e^{-x^2} dx$

```python
import math

def f1(x):
    if not isinstance(x, float):
        x = float(x)
    return math.exp(-x ** 2)

result_trapezium = trapezium_integral(f1, 0, 1)
result_simpson = simpson_integral(f1, 0, 1)
result_composite_simpson, times_composite_simpson = composite_simpson_integral(f1, 0, 1, 1e-6)
result_composite_trapezium, times_composite_trapezium = composite_trapezium_integral(f1, 0, 1, 1e-6)
result_romberg, _, times_romberg = romberg_integral(f1, 0, 1, 1e-6)

actual = integrate(exp(-Symbol('x') ** 2), (Symbol('x'), 0, 1))

print(f'''integral exp(-x ** 2) dx from 0 to 1

actual result (by sympy): {actual} = {actual.evalf()}

result_trapezium={result_trapezium}
result_simpson={result_simpson}
result_composite_trapezium={result_composite_trapezium}, times_composite_trapezium={times_composite_trapezium}
result_composite_simpson={result_composite_simpson}, times_composite_simpson={times_composite_simpson}
result_romberg={result_romberg}, times_romberg={times_romberg}
''')
```

```
integral exp(-x ** 2) dx from 0 to 1

actual result (by sympy): sqrt(pi)*erf(1)/2 = 0.746824132812427

result_trapezium=0.6839397205857212
result_simpson=0.7471804289095104
result_composite_trapezium=0.7468238989209475, times_composite_trapezium=9
result_composite_simpson=0.7468241406069852, times_composite_simpson=4
result_romberg=0.7468241326473878, times_romberg=4
```

II. 计算积分 $I_2 = \int_0^1 \frac{\sin(x)}{x} dx$

```python
def f2(x):
    if not isinstance(x, float):
        x = float(x)
    return math.sin(x) / x

result_trapezium = trapezium_integral(f2, 1e-32, 1)
result_simpson = simpson_integral(f2, 1e-32, 1)
result_composite_simpson, times_composite_simpson = composite_simpson_integral(f2, 1e-32, 1, 1e-6)
result_composite_trapezium, times_composite_trapezium = composite_trapezium_integral(f2, 1e-32, 1, 1e-6)
result_romberg, _, times_romberg = romberg_integral(f2, 1e-32, 1, 1e-6)

actual = integrate(sin(Symbol('x')) / Symbol('x'), (Symbol('x'), 0, 1))

print(f'''integral sin(x) / x dx from 0 to 1

actual result (by sympy): {actual} = {actual.evalf()}

result_trapezium={result_trapezium}
result_simpson={result_simpson}
result_composite_trapezium={result_composite_trapezium}, times_composite_trapezium={times_composite_trapezium}
result_composite_simpson={result_composite_simpson}, times_composite_simpson={times_composite_simpson}
result_romberg={result_romberg}, times_romberg={times_romberg}
''')
```

```
integral sin(x) / x dx from 0 to 1

actual result (by sympy): Si(1) = 0.946083070367183

result_trapezium=0.9207354924039483
result_simpson=0.9461458822735868
result_composite_trapezium=0.9460829746282349, times_composite_trapezium=9
result_composite_simpson=0.9460830853849476, times_composite_simpson=3
result_romberg=0.9460830703672595, times_romberg=4
```

三、思考题分析解答

（1） 为什么多节点的Newton-Cotes求积公式不宜使用?

在前文中叙述过，求积区间比较大的时候，如果使用节点数很少的 Newton-Cotes 公式会有较大的截断误差；而分很多节点时， Newton-Cotes 公式不具备数值稳定性，即便可行，科特斯系数的计算需要 $n!$ 级的数作为分母，点数 n 增多时也会带来计算上的麻烦以及可能的舍入误差。

（2） 简述什么是复化求积方法?

详见前文 实验内容 2. 复化求积公式。

（3） 简述自适应求积方法，并试着编程实现该方法计算上面4.4节的两个定积分。

详见前文 实验内容 4. 自适应 Simpson 求积法。

四、重难点分析

重点：

① 体会数值积分的基本概念；
② 掌握低阶的插值型数值积分公式；
③ 掌握区间逐次分半的复化求积方法；
④ 掌握龙贝格算法的基本思路和迭代步骤；

难点：

龙贝格算法的实现、自适应求积方法的实现。