龙贝格公式的实现

21 级计算机科学与技术 陈岳阳

2022 年 10 月 29 日

目录

1 实验目的

1.1 实现四种数值积分方法

这次试验，我们实现了复化梯形方法、复化 Simpson 方法、复化 Cotes 方法和 Romberg 方法，并在给定积分 $\int_1^3 e^x dx$ 和 $\int_1^5 \frac{1}{1+x} dx$ 上进行了测试。原要求的误差限为 10^{-6}，但这样并不能明显的区分四种方法的性能。因此，我们将要求改为 10^{-12}，以便更加清晰地判断四种方法的性能差异。

2 实验内容

2.1 四种积分方法的简介

我们即将提到的几种积分方法均假设积分区间是 $[a, b]$，被积函数为 $f(x)$，步长为 h。

假设被积区间为 n 等分，则 $h = \frac{b-a}{n}$。

2.1.1 梯形法和 Simpson 法

梯形法使用被积函数积分区间在端点处的值进行计算，具有 1 次代数精度，公式为:

$$\int_a^b f(x)dx = \frac{h}{2}[f(a) + f(b)]$$

Simpson 法除去积分区间的端点值外，还取用被积函数在积分区间中点的值进行计算。公式为:

$$\int_a^b f(x)dx = \frac{h}{6}[f(a) + f(\frac{a+b}{2}) + f(a)]$$

2.1.2 Cotes 公式

Newton-Cotes 公式是一种等距插值型数值积分。节点个数确定时，Cotes 系数可通过查表得到。n=4 时，Cotes 公式为:

$$\int_a^b f(x)dx = \frac{b-a}{90}[4f(x_0)+32f(x_1)+12f(x_2)+32f(x_3)+7f(x_4)], x_i = a+i*h$$

Cotes 公式的误差为:

$$R(f) = \int_a^b \frac{f^{(n+1)}(\xi)}{(n+1)!} w_{n+1}(x)dx = \frac{h^{n+2}}{(n+1)!} \int_0^n f^{(n+1)}(\xi)\Big[\prod_{j=0}^{n}(t-j)\Big]dt, \xi \in (a, b)$$

当 n 为偶数时，Cotes 公式至少有 n+1 阶代数精度。

2.1.3 复化型求积公式

复化型求积公式分割积分区间，并在每个区间上进行数值积分，以提高精度。

复化梯形公式:

$$\int_a^b f(x)dx = \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right] = T_n$$

余项为 $-\frac{b-a}{12}h^2 f''(\eta)$

复化 Simpson 公式:

$$\int_a^b f(x)dx = \frac{h}{6}\left[f(a) + 4\sum_{i=0}^{n-1} f(x_{i+\frac{1}{2}}) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right] = S_n$$

余项为 $-\frac{b-a}{2880}h^4 f^{(4)}(\eta)$

复化 Cotes 公式:

$$\int_a^b f(x)dx = \frac{h}{90}\left[7f(a)+32\sum_{i=0}^{n-1} f(x_{i+\frac{1}{4}})+12\sum_{i=0}^{n-1} f(x_{i+\frac{1}{2}})+32\sum_{i=0}^{n-1} f(x_{i+\frac{3}{4}})+14\sum_{i=1}^{n-1} f(x_i)+7f(b)\right] = C_n$$

余项为 $-\frac{2(b-a)}{945}(\frac{h}{4})^6 f^{(6)}(\eta), \eta \in (a,b)$

2.1.4 龙贝格公式

由复化梯形公式，容易得到:

$$T_{2n} = \frac{1}{2}T_n + \frac{h}{2}\sum_{k=0}^{n-1} f(x_{k+\frac{1}{2}})$$

分析三种复化方法的误差，可以得到:

$$S_n = \frac{4}{3}T_{2n} - \frac{1}{3}T_n$$
$$C_n = \frac{16}{15}S_{2n} - \frac{1}{15}S_n$$
$$R_n = \frac{64}{63}C_{2n} - \frac{1}{63}C_n$$

2.2 四种积分方法的实现

受制于篇幅与排版,四种积分方法的代码均于文末及附件中 (见 Romberg.md)。运行代码需要使用 jupyter lab 运行 Romberg.ipynb。

四种积分方法都有两种形式。一种为 <method>,输入积分区间 [left,right],区间个数 n,函数类型,返回数值积分数值;另一种为 <method>_precision,需要额外输出精度位数和理论值,并打印理论值、数值积分值和达到目标精度时的区间个数。<method>_precision 会调用 <method>_next 函数——一个用于递推的函数。其每次将区间个数翻倍。因此,<method>_precision 打印的区间个数一定是 2 的幂。

```python
def Romberg(left, right, n, *, functype="Exp"):
    """

    :param left: 左边界
    :param right: 右边界
    :param n: 区间个数为2 ** (n + 3)
    :param functype: 函数类型
    :return: 积分值
    """

    cotes1, simp2, trap3 = Cotes(left, right, n, func=functype)
    trap4 = Trapezoidal_next(trap3, functype, left, right, n + 2, functype=functype)[0]
    simp3 = trap4 * 4 / 3 - trap3 / 3
    cotes2 = simp3 * 16 / 15 - simp2 / 15
    return [cotes2 * 64 / 63 - cotes1 / 63, cotes2, simp3, trap4]

def Romberg_next(left, right, n, rom, *, functype):
    trap5 = Trapezoidal_next(rom[3], functype, left, right, n + 3, functype=functype)[0]
    simp4 = trap5 * 4 / 3 - rom[3] / 3
    cotes3 = simp4 * 16 / 15 - rom[2] / 15
    return [cotes3 * 64 / 63 - rom[1] / 63, cotes3, simp4, trap5]

def Romberg_precision(left, right, theory, *, functype, precision):
    epsilon = 10 ** (-precision) / 2
    n = 0
    rom = Romberg(left, right, 0, functype=functype)
    while abs(rom[0] - theory) > epsilon:
        rom = Romberg_next(left, right, n, rom, functype=functype)
        n = n + 1

    print("Romberg method")
    print(f"Theory value: {theory}")
    print(f"Predicted value: {round(rom[0], precision)}")
    print(f"Intervals: {2 ** (n + 3)}\n")
```

图 1: 龙贝格公式

2.3 四种积分方法的测试

两种测试结果如图所示。可以看出,被积函数、积分区间和误差限相同的情况下,龙贝格算法、Cotes 方法、Simpson 方法和梯形法的收敛速率依次递减。

```
[8]: precision_test(1, 3, m.e ** 3 - m.e, 12, func="Exp")

Romberg method
Theory value: 17.36725509472862
Predicted value: 17.367255094729
Intervals: 64

Cotes method
Theory value: 17.36725509472862
Predicted value: 17.367255094729
Intervals: 256

Simpson method
Theory value: 17.36725509472862
Predicted value: 17.367255094729
Intervals: 2048

Trapezoidal method
Theory value: 17.36725509472862
Predicted value: 17.367255094729
Intervals: 4194304
```

图 2: Exp 测试

```
[9]: precision_test(1, 5, np.log(5), 12, func="Frac")

Romberg method
Theory value: 1.6094379124341003
Predicted value: 1.609437912434
Intervals: 256

Cotes method
Theory value: 1.6094379124341003
Predicted value: 1.609437912434
Intervals: 512

Simpson method
Theory value: 1.6094379124341003
Predicted value: 1.609437912435
Intervals: 2048

Trapezoidal method
Theory value: 1.6094379124341003
Predicted value: 1.609437912434
Intervals: 2097152
```

图 3: Frac 测试

2.4 数值积分在 NeRF 中的应用

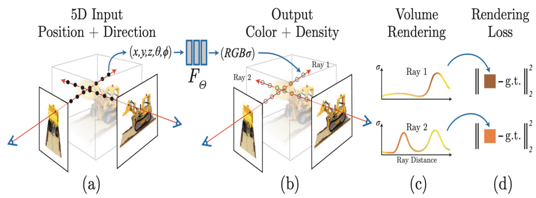

图 4: Pipeline of NeRF

NeRF(Neural Radiance Fields) 是一种隐式三维场景表示。其将场景表现为空间中任何点的 *density* σ 和 *color* c。

其中，*density* $\sigma(x)$ 代表 ray 在 x 处终止的微分概率。对于有近边界 t_n, 远边界 t_f 的 ray $r(t) = o + td$ 的颜色 C(r) 是

$$C(r) = \int_{t_n}^{t_f} T(t)\sigma(r(t))c(r(t), d)dt$$

T(t) 表示沿光线从 t_n 到 t 的累计透射率，也就是光线从 t_n 传播到 t 而没有碰到任何其他粒子的概率

$$T(t) = exp(-\int_{t_n}^{t} \sigma(r(s))ds)$$

均匀划分区间，并在每个 interval 中按均匀分布随机选取一个点。

$$t_i \sim u\Big[t_n + \frac{i-1}{N}(t_f - t_n), t_n + \frac{i}{N}(t_f - t_n)\Big]$$

然后使用采样点对 $C(r)$ 进行估计。

$$C(r) = \sum_{i=1}^{N} T_i(1 - exp(-\sigma_i\delta_i))c_i, where\ T_i = exp\Big(-\sum_{j=1}^{i-1}\sigma_j\delta_j\Big),\ \delta_i = t_i - t_{i-1}$$

由于我们的设备性能有限，在实现了代码后，仍然无法在实验课前完整运行程序一次，无法进行相应演示，只能给出一张 tiny_nerf 的结果图片。tiny_nerf 的 render 部分代码由我们修改并实现。

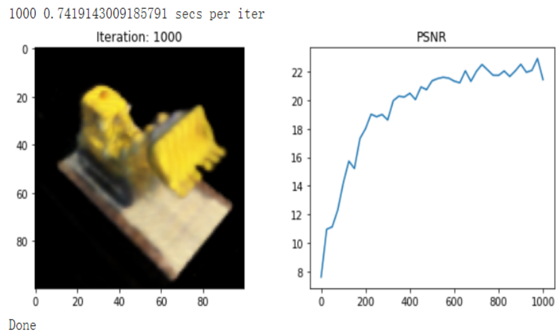

图 5: Tiny NeRF

2.5 自适应 Simpson 法

2.5.1 自适应 Simpson 法的介绍

自适应 Simpson 法是 Simpson 法的改进。将积分区间等分，并对得到的两个新区间各使用一次 Simpson 公式，得到的结果的算术平均值与整个区间上进行 Simpson 公式的结果比较。如果差别较小 (通常认为 $\delta \leq 15\epsilon$)，则返回，否则将 ϵ 减半，并对左右区间递归进行自适应 Simpson 法。ϵ 减半的操作可以保证结果与理论值的误差一定不大于 ϵ。

2.5.2 自适应 Simpson 法的实现和测试

按照介绍，我们可以实现自适应 Simpson 法。其可以很快的计算出误差在 1e-12 内的结果。

```python
def simpson(left, right, *, func):
    return Simpson(left, right, 1, func=func)[0]

def asr(left, right, simp, *, epsilon, func):
    mid = (left + right) / 2
    l = simpson(left, mid, func=func)
    r = simpson(mid, right, func=func)
    if abs(l + r - simp) <= 15 * epsilon:
        return l + r + (l + r - simp) / 15
    else:
        return asr(left, mid, l, epsilon=epsilon/2, func=func) + asr(mid, right, r, epsilon=epsilon/2, func=func)
```

```python
def asr_test(left, right, func, epsilon, theory):
    answer = asr(left, right, simpson(left, right, func=func), epsilon=epsilon, func=func)
    print("Adaptive Simpson method")
    print(f"Theory value: {theory}")
    print(f"Predicted value: {answer}")
```

```python
asr_test(1, 5, "Frac", 10 ** (-12), np.log(5))
```

```
Adaptive Simpson method
Theory value: 1.6094379124341003
Predicted value: 1.6094379124341003
```

```python
asr_test(1, 3, "Exp", 10 ** (-12), m.e ** 3 - m.e)
```

```
Adaptive Simpson method
Theory value: 17.36725509472862
Predicted value: 17.367255094728623
```

图 6: 自适应 Simpson 法

2.6 实例分析

由于 NeRF 的代码无法放出，我们选取了额外的实际问题用于演示。

卫星轨道是一个椭圆，椭圆周长计算公式是

$$S = 4a\int_0^{\frac{\pi}{2}} \sqrt{1 - (\frac{c}{a})^2 sin^2\theta d\theta}$$

这里 a 是椭圆半长轴，c 是地球中心与轨道中心的距离。记 h 为近地点距离，H 为远地点距离，R=6371km 为地球半径，则

$$a = (2R + H + h)/2,\ c = (H - h)/2$$

我国第一颗人造卫星近地点距离 h=439km，远地点距离 H=2384km，试求卫星轨道的周长。

解：显然，这是个没有解析解的椭圆积分。我们使用自适应 Simpson 法解出该题。

```
[15]: 4 * 7782.5 * asr(0, m.pi/2, simpson(0, m.pi/2, func="Real"), epsilon=10**(-12), func="Real")

[15]: 48707.43851187989
```

图 7: 实例

3 实验总结

这次实验，我们实现了四种数值积分方法，比较了它们的性能。相较于其它公式，龙贝格公式有着更快的收敛速度，是一种非常优秀的算法。

此外，NeRF 是当前大热的 CV 技术。数值积分在 NeRF 上也有一定的应用。在现实中，数值积分也有很大的应用。其用处可见一斑。

实验的过程中，我的代码能力加强了许多，对数值积分的理解更加深入，受益良多。

4 代码实现