数值分析第四章补充阅读

朱宇嘉

1 数值积分与数值微分

1.1 数值积分

对于函数 $f(x)$，如果其存在原函数 $F(x)$，则我们可以轻松地用 Newton-Lebniz 公式计算出 $f(x)$ 在 $[a,b]$ 上的积分：

$$\int_a^b f(x) = F(b) - F(a)$$

但事实上，这样优美的函数往往是不常见的，它们往往出现在数学分析的考试卷当中。更多的函数往往是不存在原函数的，或者存在原函数，但是它的原函数相当复杂或者计算十分困难。这时候，我们就要使用数值积分来快速计算函数的积分值。但是数值积分往往存在两个问题需要分析：

1. 使用数值积分算法算出的积分值与真实的积分值间的**误差**。
2. 数值积分算法是否能**快速收敛**于真实的积分值。

对于数值积分的误差分析，我们往往都使用代数精度这一指标来衡量。代数精度的思想是：一个数值积分公式要准确，那它就必须要可以较好地估计简单的多项式函数的积分。

1.2 数值微分

数值微分则是用来近似特定函数的导数值或微分值，其思想和计算与数值积分非常相似，在这里就不再展开了。

但是目前在很多需要大规模求导的应用领域中（特别是深度学习领域中），数值微分却不是最为常用的求导手段。这是因为，数值微分的缺点也十分明显：速度较慢。对于部分需要大量涉及到求导过程的应用，这显然是不符合实际的。理论导数计算（或称理论梯度求导）是大规模求导问题的解决方法。它使用了计算图（Computional Graph）从理论层面求出每个变量的导数值。关于计算图相关的具体内容，大家可以点击**这里**查看具体内容。

2 数值积分的实现

我们首先来使用编程语言来实现各种数值积分方法。需要实现的方法有复合梯形公式，复合辛普森公式和龙贝格求积公式。

不过由于大家的期末实验有很大可能是需要自己动手编写数值积分算法，所以这里的函数我只给出接口的形式，请大家自行完成函数的编写。建议大家尽早并按照接口的形式完成编写。

为了给大家一些编写的启示，我们还是会用 R 语言和 Python 语言分别来实现一下复合梯形公式。

2.1 复合梯形公式

复合梯形公式把积分区域 $[a,b]n$ 等分成等长区间 $[x_i,x_{i+1}],i=0,1,...,n-1$，并在每个小区间上利用梯形公式，得到复合梯形公式，其公式如下：

$$
\begin{aligned}
I=\int_a^b f(x)dx&=\sum_{i=0}^{n-1}\int_{x_i}^{x_{i+1}} f(x)dx\\
&\approx\sum_{i=0}^{n-1}\frac{h}{2}[f(x_i)+f(x_{i+1})]=\frac{h}{2}[f(a)+2\sum_{i=1}^{n-1}f(x_i)+f(b)]
\end{aligned}
$$

在实际应用中，我们往往需要找到合适的 n 以使得数值积分的误差小于一定的误差范围 ϵ 内。我们往往通过对 n 进行迭代来完成这一过程，不过为了减少计算次数，我们将 n 的下一步迭代数值定为 $2n$ 而不是 $n+1$。

我们可以用以下 R 语言代码来完成复合梯形公式的计算。

```
Trapezoid<-function(f,lo,hi,esp=1e-6){
  ## 输入: f 是输入函数
  ## 输入: lo,hi 是函数积分的下限和上限
  ## 输入: esp 是给定误差范围，默认值为 1e-6
  if(!is.function(f)) stop("f should be a function")
  if(lo>hi) stop("integrate range at least 0")
  if(esp<=0) stop("need positive eps")
  ## 初始化: 整个区间上的梯形公式
  outcome<-vector(mode="numeric",length=0)
  h<-(hi-lo)/2;i<-2;diff<-esp+1
  outcome[1]<-h*(f(lo)+f(hi))
  ## 迭代直至误差复合要求，迭代方案为 h->h/2
  ## 注意要报新一次的积分值与前一次的积分值联系起来，减少运算量
  while(diff>esp){
    outcome[i]<-sum(f(seq(lo+h,hi-h,2*h)))*h+outcome[i-1]/2
```

```
    h<-h/2;diff<-abs(outcome[i]-outcome[(i-1)]);i<-i+1
  }
  ## 输出: value 是数值积分值
  ## 输出: process 是迭代过程中的积分值
  ## 输出: it 是迭代次数
  process<-outcome;value<-outcome[(i-1)];iter<-i-1
  return(list(value=value,process=process,iter=iter))
}
```

我们用课件上的例 1 来测试一下。例 1 的函数是 $f(x)=\frac{\sin x}{x}$。我们在编写函数的时候需要手动修改一下 0 这个瑕点。

```
f<-function(x){
  return(ifelse(x==0,1,sin(x)/x))
}
Trapezoid(f,0,1,esp=1e-4)
```

```
## $value
## [1] 0.9460586
##
## $process
## [1] 0.9207355 0.9397933 0.9445135 0.9456909 0.9459850 0.9460586
##
## $iter
## [1] 6
```

我们使用了 6 次迭代计算出了含有 4 位有效数字的积分值。

下面我们用 Python 来重复实现一下。

```
import numpy as np

def Trapezoid(f,lo,hi,esp=1e-6):
    ## 输入: f 是输入函数
    ## 输入: lo,hi 是函数积分的下限和上限
    ## 输入: esp 是给定误差范围, 默认值为 1e-6
    assert callable(f), 'f should be a function'
    assert lo<=hi, 'integrate range at least 0'
    assert esp>0, 'need positive esp'
    ## 初始化: 整个区间上的梯形公式
    outcome = []
```

```
    h = (hi-lo)/2
    i = 1
    diff = esp+1
    outcome.append(h*(f(lo)+f(hi)))
    ## 迭代直至误差复合要求，迭代方案为 h->h/2
    ## 注意要报新一次的积分值与前一次的积分值联系起来，减少运算量
    while diff>esp:
        incr = sum([f(tmp) for tmp in np.linspace(lo+h,hi-h,2**(i-1),endpoint=True)])*h
        outcome.append(incr+outcome[-1]/2)
        h = h/2
        diff = abs(outcome[-1]-outcome[-2])
        i = i+1
    ## 输出: value 是数值积分值
    ## 输出: process 是迭代过程中的积分值
    ## 输出: it 是迭代次数
    process = outcome
    value = outcome[-1]
    it = i
    return value, process, it
```

我们同样输出结果，其与 R 语言输出的结果应该一模一样。

```
import math

def f(x):
    return 1 if x==0 else math.sin(x)/x

value, process, it = Trapezoid(f,0,1,esp=1e-4)
print("value:",round(value,7),"\nprocess:",[round(proc,7) for proc in process],"\nit:",it)
```

```
## value: 0.9460586
## process: [0.9207355, 0.9397933, 0.9445135, 0.9456909, 0.945985, 0.9460586]
## it: 6
```

2.2 复合辛普森公式

复合辛普森公式的 R 语言接口如下，请自行完成：

```
Simpson<-function(f,lo,hi,esp=1e-6){
  ## 输入: f 是输入函数
```

```
  ## 输入: lo,hi 是函数积分的下限和上限
  ## 输入: esp 是给定误差范围, 默认值为 1e-6
  if(!is.function(f)) stop("f should be a function")
  if(lo>hi) stop("integrate range at least 0")
  if(esp<=0) stop("need positive eps")

  ### 在这里完成函数编写

  ## 输出: value 是数值积分值
  ## 输出: process 是迭代过程中的积分值
  ## 输出: iter 是迭代次数
  return(list(value=value,process=process,iter=iter))

}
```

复合辛普森公式的 Python 接口如下，请自行完成：

```
def Simpson(f,lo,hi,esp=1e-6):
    ## 输入: f 是输入函数
    ## 输入: lo,hi 是函数积分的下限和上限
    ## 输入: esp 是给定误差范围, 默认值为 1e-6
    assert callable(f), 'f should be a function'
    assert lo<=hi, 'integrate range at least 0'
    assert esp>0, 'need positive esp'

    ### 在这里完成函数编写

    ## 输出: value 是数值积分值
    ## 输出: process 是迭代过程中的积分值
    ## 输出: it 是迭代次数
    return value, process, it
```

2.3 龙贝格求积公式

龙贝格求积公式的 R 语言接口如下，请自行完成：

```
Romberg<-function(f,lo,hi,esp=1e-6){
  ## 输入: f 是输入函数
  ## 输入: lo,hi 是函数积分的下限和上限
```

```
  ## 输入: esp 是给定误差范围, 默认值为 1e-6
  if(!is.function(f)) stop("f should be a function")
  if(lo>hi) stop("integrate range at least 0")
  if(esp<=0) stop("need positive eps")

  ### 在这里完成函数编写

  ## 输出: value 是数值积分值
  ## 输出: process 是迭代过程中的积分值, 注意此处应该输出一个三角矩阵
  ## 输出: iter 是迭代次数
  return(list(value=value,process=process,iter=iter))
}
```

龙贝格求积公式的 Python 接口如下，请自行完成：

```
def Romberg(f,lo,hi,esp=1e-6):
    ## 输入: f 是输入函数
    ## 输入: lo,hi 是函数积分的下限和上限
    ## 输入: esp 是给定误差范围, 默认值为 1e-6
    assert callable(f), 'f should be a function'
    assert lo<=hi, 'integrate range at least 0'
    assert esp>0, 'need positive esp'

    ### 在这里完成函数编写

    ## 输出: value 是数值积分值
    ## 输出: process 是迭代过程中的积分值, 注意此处应该输出一个三角矩阵
    ## 输出: it 是迭代次数
    return value, process, it
```

3 高斯求积公式

之前的求积公式都具有如下形式：

$$\int_a^b f(x)dx = \sum_{i=0}^{n} A_i f(x_i)$$

其中 $x_i, i = 0, 1, 2..., n$ 是 $[a, b]$ 上的**等距节点**，所求的积分值是由这组节点取值的线性组合求出的。这样的积分算法可以至少拥有 n 次代数精度。但如果我们适当地选取 $x_i, i = 0, 1, 2..., n$，我们可以使求积公式的代数精度达到 $2n + 2$ 次。我们主要介绍以下两种高斯求积公式：高斯-勒让德求积公式和高斯-切

比雪夫求积公式。它们分别使用了勒让德多项式和切比雪夫多项式作为辅助。

3.1 高斯-勒让德求积公式

对于积分 $I=\int_a^b f(x)dx$，计算其 n 阶高斯-勒让德求积公式计算步骤如下：

- **Step 1：**对区间 $[a,b]$ 做线性变换，将积分区间变换至 $[-1,1]$。不失一般性，我们下面只讨论 $[-1,1]$ 上的积分计算方法。此外还要提醒一点，积分和函数逼近不同。函数逼近需要先将 $[a,b]$ 转化到 $[-1,1]$，最后再转回 $[a,b]$；而积分在转化到 $[-1,1]$ 后直接计算即可，无需再转回 $[a,b]$。
- **Step 2：**求出最高次数为 $n+1$ 的勒让德多项式的 $n+1$ 个零点，将其作为 $x_i, i=0,1,2...,n$。
- **Step 3：**求出系数 A_i，其计算方法是 $A_i=\int_0^1 l_i(x)dx$。其中，$l_i(x), i=0,1,2...,n+1$ 是以 $x_i, i=0,1,2...,n$ 为节点进行 Lagrange 插值的基函数。
- **Step 4：**代入计算 $I=\int_a^b f^*(x)dx=\sum_{i=0}^n A_i f^*(x_i)$ 即可。其中 $f^*(.)$ 是 $f(x)$ 经线性变换后的函数。

3.2 高斯-切比雪夫求积公式

由于勒让德多项式的权函数是 $\rho(x)=1$，而切比雪夫多项式的权函数则是 $\rho(x)=\frac{1}{\sqrt{1-x^2}}$。因此高斯-切比雪夫求积公式主要解决的是以下形式的积分：

$$I=\int_{-1}^1 \frac{f(x)}{\sqrt{1-x^2}}dx$$

n 阶高斯-切比雪夫求积公式计算步骤如下：

- **Step 1：**求出最高次数为 $n+1$ 的切比雪夫多项式的 $n+1$ 个零点。其有显式解 $x_i=\cos(\frac{2k+1}{2n+2}\pi), i=0,1,2...,n$。
- **Step 2：**求出系数 A_i，其计算方法是 $A_i=\int_0^1 l_i(x)dx$。在切比雪夫插值下，其也有显式解 $A_0=A_1=...=A_n=\frac{\pi}{n+1}$。
- **Step 3：**代入计算 $I=\int_{-1}^1 \frac{f(x)}{\sqrt{1-x^2}}dx=\sum_{i=0}^n \frac{\pi}{n+1} f(\cos(\frac{2k+1}{2n+2}\pi))$ 即可。

需要指出，一般来说高斯-切比雪夫求积公式不能求 $[a,b]$ 上的积分。我们不能使用线性变换的方法将其变换到 $[-1,1]$ 上再求积分，除非变换后积分项中还有 $\frac{1}{\sqrt{1-x^2}}$ 项。

4 外推法求数值微分的实现

一般来说，要求得较为精确的微分值仅用等距节点的导数值是不够的。往往要使用外推法才能得到较为准确的微分值。因此，我们需要用程序实现外推法求数值微分。**同样的，由于在期末实验中，大家很可能需要自己实现这一方法，因此，我在这里也仅仅给出函数的接口，请大家自行编写。**

外推法求数值微分的 R 语言接口为：

```
NumDiff<-function(f,x,d=1,esp=1e-3){
  ## 输入：f 是输入函数
  ## 输入：我们希望求 x 处的导数值
  ## 输入：我们希望求 d 阶导数，默认值为 1
  ## 输入：esp 是最大误差值，默认值为 1e-3
  ## Hint：对于 d>1 的情况，使用递归法求解
  if(!is.function(f)) stop("f should be a function")
  if(d!=round(d) | d<=0) stop("d should be a positive integer")
  if(esp<=0) stop("need positive eps")

  ## 在这里完成函数编写

  ## 输出：value：f 在 x 的 d 阶导数值
  return(value)
}
```

外推法求数值微分的 Python 接口为：

```
def NumDiff(f,x,d=1,esp=1e-3):
  ## 输入：f 是输入函数
  ## 输入：我们希望求 x 处的导数值
  ## 输入：我们希望求 d 阶导数，默认值为 1
  ## 输入：esp 是最大误差值，默认值为 1e-3
  ## Hint：对于 d>1 的情况，使用递归法求解
  assert callable(f), 'f should be a function'
  assert isinstance(d, int) and d > 0, 'n should be a positive integer'
  assert esp>0, 'need positive esp'

  ## 在这里完成函数编写

  ## 输出：value：f 在 x 的 d 阶导数值
  return value
```