舍入误差

1. 什么是舍入误差

例如计算一个圆的面积，有公式：$A = \pi r^2$。其中π是无限不循环小数3.1415926535.....。假设半径r的值没有误差的条件下，当π取无限不循环小数的时候，计算得到的圆的面积才是准确的。但由于计算机内存有限，不可能存储无限位数据，所有数据的存储空间和位数都是有限且固定的，如在C语言中单精度float的有效位数为8位，双精度double的有效位数为16位，如此π只能取到3.1415926或3.141592653589793，这与π的准确值相比是有误差的，这样的误差叫做舍入误差，那么利用具有舍入误差的π计算得到的圆的面积A也是具有舍入误差的。

1.1 计算机精度验证

Python的默认浮点数是双精度，Python浮点数采用IEEE 754标准来表示。在Python中可以通过 `sys.float_info.epsilon` 方法来获取计算机浮点运算的精度。

In [ ]:
```
import sys
print(sys.float_info.epsilon)
```

```
2.220446049250313e-16
```

以上结果表明计算机浮点运算的精度在2.220446049250313e-16，实际上这个值是2^{-52}，64位双精度浮点型数的二进制位的第0-52位为2进制小数尾值，决定了它的精度在2的负52次方。意味着小数点后第16位之后是不可信的，同时如果当两个浮点数的数据相差10^{16}量级，则较小的数的贡献就会引入较为明显的舍入误差，甚至被忽略。

In [ ]:
```
print(1e-10+8e-24)
print(1e-10+8e-25)
print(1e-10+8e-26)
print(1e-10+8e-27)  # 相差16个数量级，计算结果已经不准确了
print(1e-10+1e-27)  # 相差17个数量级，较小的数的贡献被忽略
```

```
1.00000000000008e-10
1.000000000000008e-10
1.0000000000000008e-10
1.0000000000000002e-10
1e-10
```

再举一个简单的计算的例子

In [ ]:
```
a=4/3   # a是一个无限循环小数，a=1.3333333...
b=a-1   # b=0.333333333...
c=1/3
print(b)    # 输出结果是0.3333333333333326，误差为4e-17
print(b+b+b)    # 输出结果是0.9999999999999998，误差为2e-16
print(c+c+c)    # 输出结果是1.0，说明舍入误差发生在b=a-1浮点运算时
```

```
0.33333333333333326
0.9999999999999998
1.0
```

以上例子说明在计算机中有限位数的浮点运算是必然会产生舍入误差的。

2. 舍入误差能造成多大的影响

数值计算课程上老师告诉我们数值计算中应该注意的几个问题：

1. 避免相近的数相减
2. 避免大数吃小数
3. 简化运算步骤、减少运算次数

那么到底这三条会产生多大的影响，我们逐条举个例子。

2.1 避免相近的数相减——会严重损失有效位数

比如计算$1.234-1.233$，原本各有4位有效数字，最后的理论计算结果为0.001，有效位数只剩下1位，而计算机实际输出的是0.0009999999999763531，误差为10^{-16}。如此再进行后面的计算，如：$10.333\times(1234.567-1234.566)$，计算的理论结果为0.010333，而计算机实际输出的是0.010332999999755657，此时误差为10^{-15}。如果在此基础上再乘10.333，理论结果为0.106770889，而计算机实际输出为0.10677088899747521，误差变为10^{-14}。即如果乘数>10，每计算一次，都会损失一位有效位数，误差也会逐渐累积提升一个数量级。长此以往，后果不堪设想。

In [ ]:
```
a=1.234
b=1.233
c=a-b
print(c)
d1=10.333*c
print(d1)
e1=10.333*d1
print(e1)
```

```
0.0009999999999998899
0.010332999999998862
0.10677088899998824
```

2.2 避免大数吃小数

举一个Google的首席科学家Vincent Vanhoucke曾举过的例子，考虑在10^9的基础上，加上10^{-6}，重复10^6次，再减去10^9，即$10^9+10^6*10^{-6}-10^9$，理论值应该为1。可是计算机最后的计算结果为0.95367431640625，误差高达0.04632568359375。

In [ ]:
```
summ = 1000000000

for indx in range(1000000):
    summ += 0.000001

summ -= 1000000000
```

```
print(summ)
```

```
0.95367431640625
```

以上例子告诉我们，计算过程越多，累计误差越大，所以简化运算步骤、减少运算次数便很容易理解了。

如今解决大数吃小数的问题有一个Kahan算法具有比较好的效果。Kahan算法的核心是记录了小数的负的补全部分compensation，随着这个补全部分的不断积累，当这些截断误差积累到一定量级，它们在求和的时候也就不会被截断了，从而能够相对好地控制整个求和过程的精度。

In [ ]:
```
def KahanSum(input):
    var sum = 0.0
    var c = 0.0
    for i = 1 to input.length do
        var y = input[i] - c    # Initially, c is zero; then it compensates prev
        var t = sum + y         # low-order digits of y are lost
        c = (t - sum) - y       # recover the low-order digits of y, with negati
        sum = t
    next i

    return sum
```

比如，用$10000.0 + pi + e$来说明，我们依旧假设浮点型变量只能保存6位数值。此时，具体写出求和算式应该是：$10000.0 + 3.14159 + 2.71828$，它们的理论结果应该是$10005.85987$，约等于$10005.9$。 但由于截断误差，第一次求和$10000.0 + 3.14159$只能得到结果$10003.1$；这个结果再与$2.71828$相加，得到$10005.81828$，被截断为$10005.8$。此时结果就相差了0.1。 运用Kahan求和法，我们的运行过程是（记住，我们的浮点型变量保存6位数值）

```
    y = 3.14159 - 0.00000

    t = 10000.0 + 3.14159
      = 10003.14159
      = 10003.1                        // low-order digits have lost

    c = (10003.1 - 10000.0) - 3.14159
      = 3.10000 - 3.14159
      = - (.0415900)                   // recover the negative parts
    of compensation errors

    sum = 1003.1
```

下面我们将Kahan算法应用到Vincent Vanhoucke的例子中，可以发现能够获得一个准确的结果。

In [ ]:
```
summ = 1000000000
c = 0.0

for indx in range(1000000):
    y = 0.000001 - c
```

```
    t = summ + y
    c = (t - summ) - y
    summ = t

summ -= 1000000000

print(summ)
```

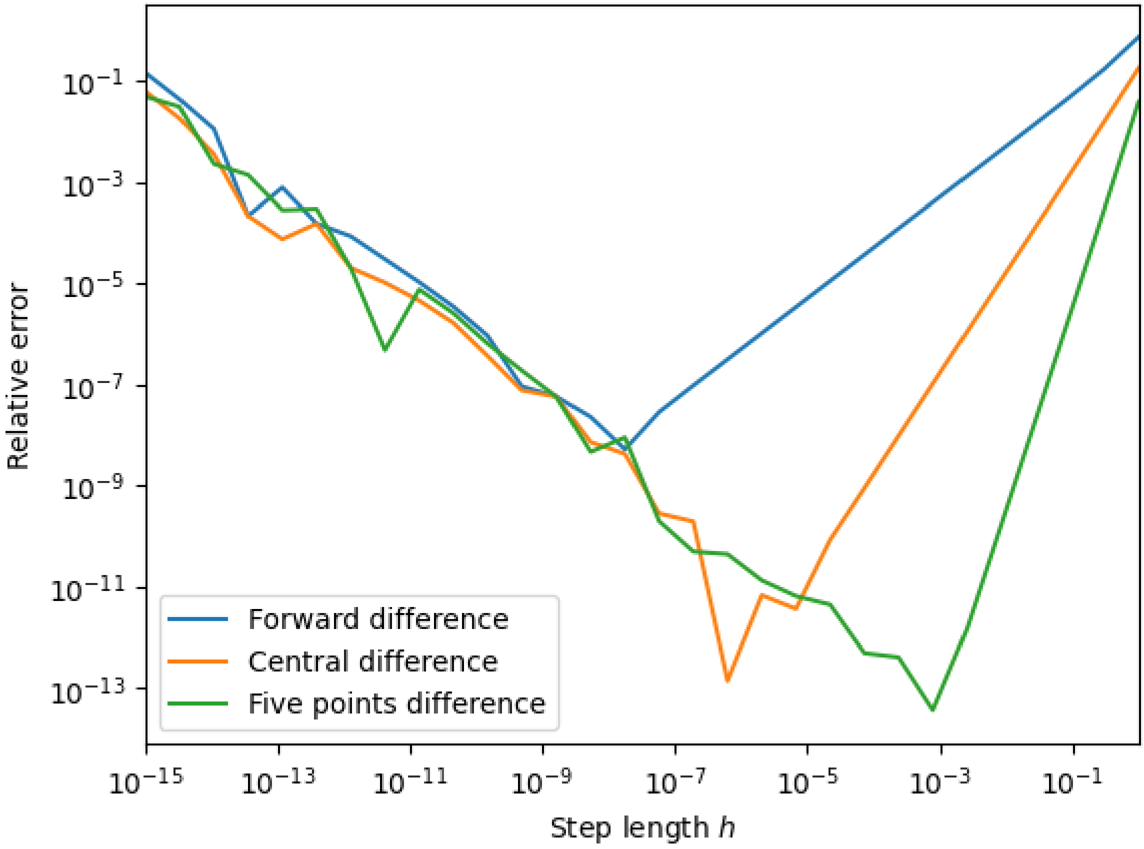
```
1.0
```

2.3 再举一些我们平常经常会用到的场景的例子

考虑分别利用向前差分（一阶精度）、中心差分（二阶精度）和五点差分（三阶精度）计算函数$f(x) = e^x$在$x = 1$处的一阶导数值。

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np

def ForwardDiff(fx, x, h=0.001):
    # Forward difference
    return (fx(x+h) - fx(x))/h

def CentralDiff(fx, x, h=0.001):
    # Central difference
    return (fx(x+h) - fx(x-h))/h*0.5

def FivePointsDiff(fx, x, h=0.001):
    # Five points difference
    return (-fx(x+2*h) + 8*fx(x+h) - 8*fx(x-h) + fx(x-2*h)) / (12.0*h)

# choose h from 0.1 to 10^-t, t>=2
t = 15
hx = 10**np.linspace(0,-t, 30)

# The exact derivative at x=1
x0 = 1
fprimExact = np.exp(1)

# Numerical derivative using the three methods
fprimF = ForwardDiff(np.exp, x0, hx)
fprimC = CentralDiff(np.exp, x0, hx)
fprim5 = FivePointsDiff(np.exp, x0, hx)

# Relative error
felF = abs(fprimExact - fprimF)/abs(fprimExact)
felC = abs(fprimExact - fprimC)/abs(fprimExact)
fel5 = abs(fprimExact - fprim5)/abs(fprimExact)

# Plot
fig, ax = plt.subplots(1)
ax.loglog(hx, felF)
ax.loglog(hx, felC)
ax.loglog(hx, fel5)
ax.autoscale(enable=True, axis='x', tight=True)
ax.set_xlabel(r'Step length $h$')
ax.set_ylabel('Relative error')
ax.legend(['Forward difference','Central difference', 'Five points difference'])
```

Out[ ]:
```
<matplotlib.legend.Legend at 0x1eb2b41fa00>
```



利用excel得到具体的计算结果有：

h	f(x-2h)	f(x-h)	f(x)	f(x+h)	f(x+2h)	向前差分	相对误差	中心差分	相对误差	五点差分	相对误差
0.000000000000000010	2.7182818284590500	2.7182818284590500	2.7182818284590500	2.7182818284590500	2.7182818284590500	0.0000000000000000	1.0000000000000000	0.0000000000000000	1.0000000000000000	(3.7007434154171900)	2.3614274195825700
0.000000000000000100	2.7182818284590400	2.7182818284590500	2.7182818284590500	2.7182818284590500	2.7182818284590500	0.0000000000000000	1.0000000000000000	0.0000000000000000	1.0000000000000000	(1.1102230246251600)	1.4084282258747700
0.000000000000001000	2.7182818284590400	2.7182818284590400	2.7182818284590500	2.7182818284590500	2.7182818284590500	3.1086244689504400	0.1435990324493590	2.8865798640254100	0.0619133872744046	3.0346096006420900	0.1163704840577080
0.000000000000010000	2.7182818284589900	2.7182818284590200	2.7182818284590500	2.7182818284590700	2.7182818284591000	2.7089441800853800	0.0034351288655586	2.7089441800853800	0.0034351288655586	2.7089441800853800	0.0034351288655586
0.000000000000100000	2.7182818284585000	2.7182818284587700	2.7182818284590500	2.7182818284593200	2.7182818284595900	2.7178259642823800	0.0001677030585605	2.7178259642823800	0.0001677030585605	2.7170858155993000	0.0004399885424769
0.000000000001000000	2.7182818284536100	2.7182818284563300	2.7182818284590500	2.7182818284617600	2.7182818284644800	2.7187141427020800	0.0001590395221394	2.7184920980971600	0.0000773538769644	2.7186031203996200	0.0001181966995519
0.000000000010000000	2.7182818284046800	2.7182818284318600	2.7182818284590500	2.7182818284862300	2.7182818285134100	2.7183144624132200	0.0000120053608244	2.7182922579527300	0.0000038367963070	2.7183070609263900	0.0000092825059854
0.000000000100000000	2.7182818279153900	2.7182818281872200	2.7182818284590500	2.7182818287308700	2.7182818290027000	2.7182833761685300	0.0000005693705000	2.7182811557224800	0.0000002474859518	2.7182793053507700	0.0000009281996616
0.000000001000000000	2.7182818230224800	2.7182818257407600	2.7182818284590500	2.7182818311773300	2.7182818338956100	2.7182820439009000	0.0000000792566289	2.7182818218562900	0.0000000024290163	2.7182819698860300	0.0000000520280805
0.000000010000000000	2.7182817740934100	2.7182818012762300	2.7182818284590500	2.7182818556418600	2.7182818828246800	2.7182818218562900	0.0000000024290163	2.7182818218562900	0.0000000024290163	2.7182818181555500	0.0000000037904438
0.000000100000000000	2.7182812848027300	2.7182815566308800	2.7182818284590500	2.7182821002872400	2.7182823721154600	2.7182819684057300	0.0000000514835095	2.7182818285176300	0.0000000000215531	2.7182818299979300	0.0000000005661242
0.000001000000000000	2.7182763919008300	2.7182791101785800	2.7182818284590500	2.7182845467422300	2.7182872650281400	2.7182831874306100	0.0000004999377016	2.7182818282955900	0.0000000000601326	2.7182818283326000	0.0000000000465183
0.000010000000000000	2.7182274633661300	2.7182546457766700	2.7182818284590500	2.7183090114132400	2.7183361946392700	2.7182954199567200	0.0000050000325683	2.7182818285176300	0.0000000000215529	2.7182818284880300	0.0000000000106616
0.000100000000000000	2.7177382264553700	2.7180100138671600	2.7182818284590500	2.7185536702337500	2.7188255391940000	2.7184177470829200	0.0000500016673974	2.7182818329896100	0.0000000016667020	2.7182818284602700	0.0000000000004509
0.001000000000000000	2.7128506977432200	2.7155649053185700	2.7182818284590500	2.7210014698815800	2.7237238323058100	2.7196414225332300	0.0005001667082295	2.7182822815057200	0.0000001666665590	2.7182818284586400	0.0000000000001480
0.010000000000000000	2.6644562419294200	2.6912344723492600	2.7182818284590500	2.7456010150169200	2.7731947639643000	2.7319186557871200	0.0050167084168053	2.7183271333827100	0.0000166667500017	2.7182818275529600	0.0000000003333317
0.100000000000000000	2.2255409284924700	2.4596031111569500	2.7182818284590500	3.0041660239464300	3.3201169227365500	2.8588419548738800	0.0517091807564779	2.7228145639474200	0.0016675001984405	2.7182727567264900	0.0000033373039028
1.000000000000000000	0.3678794411714420	1.0000000000000000	2.7182818284590500	7.3890560989306500	20.0855369231877000	4.6707742704716100	0.7182818284590460	3.1945280494653300	0.1752011936438020	2.6162326091190800	0.0375418097827679
2.000000000000000000	0.0497870683678639	0.3678794411714420	2.7182818284590500	20.0855369231877000	148.4131591025770000	8.6836275473643100	2.1945280494653300	4.9294143705040600	0.8134302039235100	0.3907453259133780	0.8562528278626340
4.000000000000000000	0.0009118819655545	0.0497870683678639	2.7182818284590500	148.4131591025770000	8103.0839275753800000	36.4237193185294000	12.3995375082861000	18.5454215042761000	5.8224792992819400	(144.0870008212450000)	54.0066453421887000
8.000000000000000000	0.0000003059023205	0.0009118819655545	2.7182818284590500	8103.0839275753800000	24154952.7535753000000000	1012.5457057183700000	371.4947483802160000	506.4426884808390000	185.3098532236940000	(250938.8342650980000000)	92316.2381176576000000
10.000000000000000000	0.0000000056027964	0.0001234098040867	2.7182818284590500	59874.1417151978000000	1318815734.4832100000000000	5987.1423433369400000	2201.5465794806700000	2993.7070795894000000	1100.3232874703400000	(10986139.5112540000000000)	4041575.8640316200000000

对于五点差分中几个特殊结果				
h=1e-17	分子部分的误差由舍入误差引起	-4.44089E-16	除以十分小的分母后误差变得十分显著	-3.700743415
h=1e-16	分子部分的误差由舍入误差引起	-1.33227E-15	除以十分小的分母后误差变得十分显著	-1.110223025
h=4	误差由离散格式的精度引起			
h=8				
h=10				

这是一个非常有趣的结果，我们发现：这三种方法的相对误差，随着h的不同，呈现出两种不同的特性。

- 当h较大时（右半侧），误差曲线相对规则，是由离散化误差主导的。
- 当h较小时（左半侧），误差曲线波动较大，是由舍入误差主导。

舍入误差主导且随着h减小而增大的主要原因是：

当h较小时，有限差分法的分子上相近的数相减会造成类似前面例子中的舍入误差。 由于这个例子中的$f(x)$在e附近，因此这个误差应为10^{-16}左右。它除以较小的h时，就会被相应的放大，当h越小，这个舍入误差越大。 因此，当我们使用上述方法时，需要注意，尽可能取h在误差曲线的右半侧，这样我们对于误差才有完全的控制。

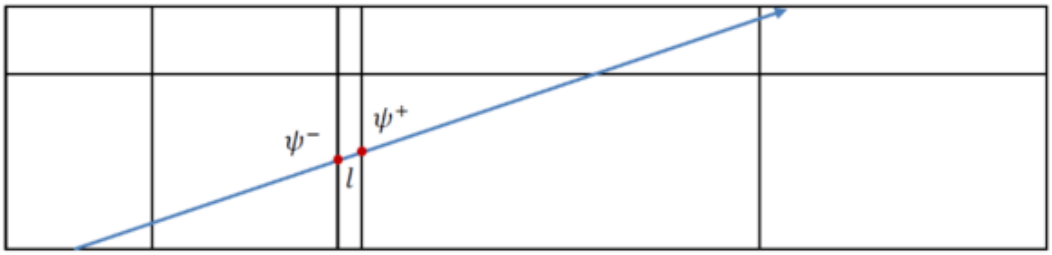
同时，我们也观察到，对于越是高阶的差分（如五点差分），它的离散化误差随h的下降速率越大，但也越早到达舍入误差的区域。 因此，当我们遇到类似问题上，应选择合适阶

数的有限差分方法，并根据它的特性选择适合的h值。并不一定是越高阶的方法越好，h越小越好。

MOC精细结构计算

在特征线方法中，核心的公式就是在每一段segment上利用入射角通量的指数衰减计算segment上的通量变化量，然后再将所有segment的通量变化量求积，即：

$$\delta\phi = \varphi^- - \varphi^+ = (\varphi^- - Q/\Sigma_T)(1 - e^{-l\Sigma_T})$$



考虑图中这一精细结构的计算，假设线段长度l为1cm，则光学距离约为0.106，那么$(1 - e^{-l\Sigma_T}) \approx 0.1$，如果$\delta\phi$恰好也在0.1附近，则$(\varphi^- - Q/\Sigma_T)$便会遇到相近的数相减的问题，从而损失有效位数引入误差，而这种误差会在标通量和源的计算中不断累积，且网格细化并不能解决这个问题。

另一方面，如果结构太过精细，也可能会踩到大数吃小数的坑。

这或许也是MOC谜一样的敏感性分析的一项可能原因。