**第1次上机实验报告**

**学号：3120004941 班级：计算机科学与技术伏羲班 姓名：吴兴桦**

目录

[实验2：减少运算次数的实验结果分析 1](#_Toc98479336)

[实验3：求解非线性方程的二分法实现 4](#_Toc98479337)

# 实验2：减少运算次数的实验结果分析

【实验目的】比较不同算法求多项式的运算次数与用时。

设计2种不同的算法计算以下函数的值，分别测试*x* = 0.1，1，2。

 (1)

【实验条件】

计算机配置

CPU：2375MHz

内存大小：12.0 GB

操作系统：Windows 64位操作系统, 基于x64的处理器

【算法介绍】

算法1：直接法

对于多项式

 (2)

分别计算的值，再将其相加，总共需要执行*n*次加法和次乘法，例如：

当时，，执行1次加法和1次乘法；

当时，，执行2次加法和3次乘法；

当时，，执行3次加法和6次乘法；

那么当时，，执行100000次加法和5000050000次乘法。

算法2：秦九韶算法

将多项式(2)变形为

 (3)

由内向外逐层计算一次多项式的值，把*n*次多项式的求值问题转化为求*n*个一次多项式值的问题，即利用

 (4)

进行迭代，最后得到的结果。不难发现，通过一次式的反复计算逐步得出高次多项式的值，只需执行*n*次加法和*n*次乘法。

【实验结果及分析】

表1 算法比较结果

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *x* | 算法 | 函数结果*f* | 乘法次数 | 加法次数 | 用时(秒) |
| 0.1 | 算法1 | 1.234567901234568 | 5000050000 | 100000 | 0.089 |
| 算法2 | 1.234567901234568 | 100000 | 100000 | 0.041 |
| 1 | 算法1 | 5000150001 | 5000050000 | 100000 | 0.068 |
| 算法2 | 5000150001 | 100000 | 100000 | 0.051 |
| 2 | 算法1 | Length:30109 | 5000050000 | 100000 | 14 |
| 算法2 | Length:30109 | 100000 | 100000 | 0.6 |
| 10 | 算法1 | Length:100006 | 5000050000 | 100000 | 1.8e+02 |
| 算法2 | Length:100006 | 100000 | 100000 | 0.94 |

结果讨论：

首先验证当时函数结果*f*的正确性。若多项式(2)的系数满足，则

 (5)

当时，可得，结果与表1相符。

从表1可以看出，算法1与算法2的函数结果相同，加法次数相同，乘法次数不同，用时不同，大致得出如下规律：

1. 当*x*值相同时，算法2的用时比算法1少。算法1与算法2的加法执行次数相同，都为100000，但算法1的乘法执行次数高于算法2，因而用时更长。
2. 在同一个算法中，当*x*的值越大，用时越长。当*x*越大，Python进行超大整数的运算的次数越多，越耗时。
3. 当*x*取值较大时，算法2明显优于算法1。算法1执行大整数的乘法运算的次数远远多于算法2，因而耗时更长。

因此，算法2把求一个*n*次多项式的值转化为求*n*个一次多项式的值，减少了运算次数，大大提高了运算效率。

【算法1源代码】

|  |
| --- |
| from time import \* # 引入时间库  x = 0.1 # 自变量 x  f = 1 # 函数值 f  countMul = 0 # 统计乘法次数  countAdd = 0 # 统计加法次数  startT = time() # 记录起始时间  # 直接法  for i in range(100000): # i从0开始  f = f + (i+2) \* x\*\*(i+1) # 函数  countAdd += 1  countMul += i + 1    endT = time() # 记录结束时间  print("result = ", f) # 输出  print("time = %.2g秒\n" % (endT - startT))  print("乘法次数", countMul)  print("加法次数", countAdd) |

【算法2源代码】

|  |
| --- |
| from time import \* # 时间统计库  x = 2 # 自变量 x  powN = 100000  aN = powN + 1 # 最后一个系数值  countMul = 0 # 统计乘法次数  countAdd = 0 # 统计加法次数  startT = time()  S = aN  for i in range(powN, 0, -1):  S = x \* S + i  countAdd += 1  countMul += 1    endT = time()  print("result =", S)  print("乘法次数：", countMul)  print("加法次数：", countAdd)  print("time = %.2g秒\n" % (endT - startT)) |

# 实验3：求解非线性方程的二分法实现

【实验目的】使用二分法求解非线性方程

用二分法求解方程在[1.3, 1.5]上的解。

【算法介绍】

原理：若，且，则在上必有一根。

第一步，取，，中点，如果，那么，即为根，算法停止。否则，如果，取，；如果，取，，此时方程的有根区间缩小为。

第二步，区间的中点为，如果，那么，即为根，算法停止。否则，如果，取，；如果，取，，此时方程的有根区间缩小为。

重复上述步骤，不断缩小有根区间，直至找到满足精度的解。

【实验结果及分析】

表2 二分法求解过程数据

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 迭代次数 | 下限*x*low | 上限*x*up | (*x*up + *x*low)/2 | *f*((*x*up + *x*low)/2)  的正负性 |
| 0 | 1.3 | 1.5 | 1.4 | < 0 |
| 1 | 1.4 | 1.5 | 1.45 | > 0 |
| 2 | 1.4 | 1.45 | 1.425 | > 0 |
| 3 | 1.4 | 1.425 | 1.4125 | < 0 |
| 4 | 1.4125 | 1.425 | 1.41875 | > 0 |
| 5 | 1.4125 | 1.41875 | 1.415625 | > 0 |
| 6 | 1.4125 | 1.415625 | 1.4140625 | < 0 |
| 7 | 1.4140625 | 1.415625 | 1.41484375 | > 0 |
| 8 | 1.4140625 | 1.41484375 | 1.414453125 | > 0 |
| 9 | 1.4140625 | 1.414453125 | 1.4142578125 | > 0 |
| 10 | 1.4140625 | 1.4142578125 | 1.41416015625 | < 0 |
| 11 | 1.41416015625 | 1.4142578125 | 1.414208984375 | < 0 |
| 12 | 1.414208984375 | 1.4142578125 | 1.4142333984375 | > 0 |
| 13 | 1.414208984375 | 1.4142333984375 | 1.41422119140625 | > 0 |
| 14 | 1.414208984375 | 1.41422119140625 | 1.41421508789062 | > 0 |
| 15 | 1.414208984375 | 1.41421508789062 | 1.41421203613281 | < 0 |
| 16 | 1.41421203613281 | 1.41421508789062 | 1.41421356201172 | < 0 |
| 17 | 1.41421356201172 | 1.41421508789062 | 1.41421432495117 | > 0 |
| 18 | 1.41421356201172 | 1.41421432495117 | 1.41421394348145 | > 0 |
| 19 | 1.41421356201172 | 1.41421394348145 | 1.41421375274658 | > 0 |
| 20 | 1.41421356201172 | 1.41421375274658 | 1.41421365737915 | > 0 |
| 21 | 1.41421356201172 | 1.41421365737915 | 1.41421360969543 | > 0 |
| 22 | 1.41421356201172 | 1.41421360969543 | 1.41421358585358 | > 0 |
| 23 | 1.41421356201172 | 1.41421358585358 | 1.41421357393265 | > 0 |
| 24 | 1.41421356201172 | 1.41421357393265 | 1.41421356797218 | > 0 |
| 25 | 1.41421356201172 | 1.41421356797218 | 1.41421356499195 | > 0 |
| 26 | 1.41421356201172 | 1.41421356499195 | 1.41421356350183 | > 0 |
| 27 | 1.41421356201172 | 1.41421356350183 | 1.41421356275678 | > 0 |
| 28 | 1.41421356201172 | 1.41421356275678 | 1.41421356238425 | > 0 |
| 29 | 1.41421356201172 | 1.41421356238425 | 1.41421356219798 | < 0 |
| 30 | 1.41421356219798 | 1.41421356238425 | 1.41421356229112 | < 0 |
| 31 | 1.41421356229112 | 1.41421356238425 | 1.41421356233768 | < 0 |

结果讨论：

如表2所示，二分法求解过程中，有根区间不断缩小，*f*((*x*up + *x*low)/2)的正负性不断变化，在进行了31次迭代后停止，最终得到了满足精度要求的实根：1.41421356233768。

下面对结果进行讨论：

首先验证实根的存在性。因为，，且在连续。由二分法的原理知，方程在(1.3,1.5)内必有一实根。

然后验证迭代次数。设初始区间为(*a*, *b*)，区间长度为，第*n*次等分的区间为。在任何情况下，新的区间长度都是初始区间长度大小的一半。重复等分*n*次，直到区间长度的平方减少到一个很小的值，满足

 (6)

第1次等分区间长度减少为，第2次等分区间长度减少为，第*n*次等分区间长度减少为，满足

 (7)

就停止等分，那么迭代次数（等分次数）

 (8)

将代入式(8)中，得**，与表2结果相符。

因此，任意给定一个正数（不论它多么小）和初始的有根区间(*a*, *b*)（不论它多么大），总存在正整数，使得当时，不等式都成立。换句话说，使用二分法总能求出满足精度要求的根，这是二分法的优点。但其缺点是不能求重根。

【算法源代码】

|  |
| --- |
| import math  a = 2 # f(x) = x\*x – a  LIMIT = 1e-20 # 终止条件  # 方程函数定义  def f(x):  """函数值的计算"""  return x\*x - a  # 主执行部分  # 初始设置  xlow = float(input("请输入x值下限："))  xup = float(input("请输入x值上限："))  # 循环处理  iter = 0 # 迭代计数  while (xup - xlow) \* (xup - xlow) > LIMIT:  xmiddle = (xup + xlow) / 2 # 计算新的中值点  iter += 1 # 迭代计数加1  if f(xmiddle) > 0: # 中点函数值为正  xup = xmiddle # 更新xup  if f(xmiddle) < 0: # 中点函数值为负  xlow = xmiddle # 更新xlow    print("{:.15g}, {:.15g}, {:.15g}, {:.15g}, {:.15g}".format(iter, xlow, xup, (xlow+xup)/2, f((xlow+xup)/2)<0))  # 迭代次数验证  # print(int(math.ceil(math.log2((1.5-1.3)/math.sqrt(LIMIT)))))  print(int(math.ceil(math.log((1.5 - 1.3)/math.sqrt(LIMIT))/math.log(2.0)))) |

附加题：对于大量的输出数据，有什么便捷的方法把列表数据输出，方便贴到word文档呢？或者直接把数据矩阵输出到.txt文件中呢？

**方法1** 输出时加逗号，把结果复制到.txt文件中，然后把后缀改成.csv并打开，复制前4列数据到word文档中后，使用“替换”功能将最后一列数据的“0”替换成“> 0”，将“1”替换成“< 0”，再复制第5列到word文档中。

**方法2** 将结果的5列数据分5次运行输出，每输出一次复制一次结果到word文档中。