**第1次上机实验报告**

**学号：3121005358 班级：人工智能1班 姓名：欧炜标**

目录

[实验1：减少运算次数的实验结果分析 1](#_Toc128511668)

[实验2：求解非线性方程的二分法实现 2](#_Toc128511669)

# 实验1：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：Nitro AN515-57

CPU：11th Gen Inter(R) Core™ i7-11800H @ 2.30GHz(16 CPUs), ~2.3GHz

内存大小：16384MB RAM

操作系统：windows 10家庭中文版64位（10.0，内部版本19044）

【算法介绍】

算法1：直接法

对多项式采用直接法计算，只需要计算每一项的值，再将其相加即可得到的结果，需要执行n次加法，次乘法，具体案例如下：

时，，执行1次加法和1次乘法；

时，，执行2次加法和3次乘法；

……

时，，执行100000次加法和5000050000次乘法。

算法2：秦九韶算法

该算法的基本思路是将多项式分解为若干个乘积形式，然后通过不断合并同类项，减少计算的次数。在对多项式计算时，通过利用公式



进行迭代，最后得到的结果，此操作只需要执行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 | 5000150000 | 100000 | 0.034 |
| 算法2 | 1.234567901234568 | 100000 | 100000 | 0.024 |
| 1 | 算法1 | 5000150000 | 5000150000 | 100000 | 0.039 |
| 算法2 | 5000150001 | 100000 | 100000 | 0.026 |
| 2 | 算法1 | Length：30109 | 5000150000 | 100000 | 9 |
| 算法2 | Length：30109 | 100000 | 100000 | 0.27 |

结果讨论：

从表1中可以看出：

1. 算法二（秦九韶法）的乘法次数和加法次数都比算法一（直接法）要少很多，这是因为秦九韶法通过使用公因式进行简化，避免了重复的计算，从而减少了乘法的次数。除了乘法次数和用时外，算法1与算法2的函数结果和加法次数相同。
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. 从用时（秒）这一指标来看，对于较小的输入，算法一和算法二的运行时间差异并不明显，但随着输入的增加，算法一的运行时间会快速增加，而算法二的运行时间相对较稳定，这也印证了算法二的高效性。

因此，由上述结果可知，随着迭代次数更多的情况下，秦九韶算法求解函数值明显比直接法具有更高的效率，其使用价值更高。

【算法1源代码】

|  |
| --- |
| 把源代码贴于此处  from time import \* #时间统计库  x = 2 #自变量 x  f = 1  countMul = 0 #统计乘法次数  countAdd = 0 #统计加法次数  startT = time() #记录起始时间  for i in range(100000): # i从0开始  f = f + (i+2)\*x\*\*(i+1)#函数  countMul += i +2 #此处只统计算法的加法，忽略i的计数  countAdd += 1 #每次增加的乘法次数  endT = time() #记录结束时间  print("resule = ", len(str(f))) #输出  print("乘法次数", countMul)  print("加法次数", countAdd)  print("time = %.2g 秒\n" % (endT - startT)) |

【算法2源代码】

|  |
| --- |
| 把源代码贴于此处  from time import \* #时间统计库  x = 2 #自变量 x  powN = 100000 #最后一个数的幂次  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() #记录起始时间  S = aN #函数值  for i in range(powN,0,-1): #i从powN开始到1  S = x\*S + (aN - (powN - i +1)) #迭代函数  countMul += 1 #此处只统计算法的加法，忽略i的计数  countAdd += 1 #每次增加的乘法次数  endT = time() #记录结束时间  print("resule = ", len(str(S))) #输出  print("乘法次数", countMul)  print("加法次数", countAdd)  print("time = %.2g 秒\n" % (endT - startT)) |

# 实验2：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】

二分法是一种求解非线性方程的数值方法，其基本思想是将一个区间不断缩小，直到找到方程的根。其原理是基于区间中值定理：如果一个连续的实函数f在区间[a,b]的端点f(a)和f(b)处取值异号，则在[a,b]内至少存在一点c，使得f(c)=0。

基本原理：基于区间中值定理，即如果一个连续的实函数f在区间[a,b]的端点f(a)和f(b)处取值异号，则在[a,b]内至少存在一点c，使得f(c)=0。

步骤：

1. 确定一个包含方程根的初始区间[a,b]，即使得f(a)和f(b)异号。
2. 求区间中点c=(a+b)/2，并计算f(c)的值。
3. 如果f(c)等于零，c即为方程的解，结束迭代。
4. 如果f(c)不为零，则根据f(c)的符号，将区间[a,b]缩小为[a,c]或[c,b]。
5. 重复执行步骤2~4，直到满足收敛条件或迭代次数达到预设的最大值为止。

常用的收敛条件包括：达到预设的迭代次数、区间长度小于预设的阈值、方程的解满足预设的精度等。如果满足任意一个收敛条件，则可以认为已经找到了方程的解。

【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：

实验结果：根据表格中的数据，可以发现在迭代次数为0时，函数在区间中点的函数值小于0，因此解位于区间的左半部分。然后在每次迭代中，将原始区间缩小到其左侧或右侧的一半，直到找到解。在迭代的前几步中，函数值从负值变为正值，然后保持为正值，最终趋近于零。特别地，在第17步时，函数值从正变为负，这意味着我们要将区间缩小到右侧一半，以继续查找解。然后在下一步中，函数值又变为正，说明解仍在右侧区间的左半部分。在第30步时，区间缩小到足够小，可以认为已经找到了解，即1.41421356229112。

下面对实验结果进行分析：

1. 确定初始区间：从表2中的数据可以看出，初始区间为[1.3,1.5]。这意味着函数f(x)在[1.3,1.5]内存在零点，因为f(1.3)和f(1.5)的正负性不同。
2. 进行迭代：二分法的核心就是区间缩减，根据表2中的数据，可以发现：

在第0次迭代中，区间为[1.3,1.5]，中点为1.4，f(1.4)为负数，因此零点在右半区间[1.4,1.5]内。

在第1次迭代中，区间为[1.4,1.5]，中点为1.45，f(1.45)为正数，因此零点在左半区间[1.4,1.45]内。

在第2次迭代中，区间为[1.4,1.45]，中点为1.425，f(1.425)为正数，因此零点在左半区间[1.4,1.425]内。

...

以此类推，可以发现每一次迭代都会将区间缩小一半，并将零点所在的区间缩小至原来的一半。通过不断迭代，最终可以得到零点的近似值。

1. 判断迭代中值条件：

根据表2中的数据，迭代过程中最终停止在第31次迭代，最终区间为[1.41421356229112,1.41421356238425]，长度为6.912e-08。

综上所述，对于上述实验结果，可以通过检查收敛性和比较不同方法的结果来验证其正确性。从实验结果中可以看到，迭代次数逐渐增加，上限和下限之间的差值逐渐缩小，最终得到了函数的根。因此，可以认为该实验结果是可靠的。

【算法源代码】

|  |
| --- |
| 把源代码贴于此处  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："))  with open('result.txt', 'w') as f: f.write('iter, xlow, xup, xmiddle, f(xmiddle) < 0\n') 打开文件并准备写入结果  # 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    f.write("{:.15g}, {:.15g}, {:.15g}, {:.15g}, {:.15g}\n".format(iter, xlow, xup, (xlow+xup)/2, f((xlow+xup)/2)<0)) #将结果写入文件result.txt    # 迭代次数验证[1]  # f.write(str(int(math.ceil(math.log2((1.5-1.3)/math.sqrt(LIMIT))))) + '\n')  f.write(str(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：将列表数据直接输出到 .txt 文件中，使用 Python 内置的文件操作函数 open() 和 write() 就可以完成，想将.txt文件输出成表格，只需要将后缀改成.csv即可。

参考文献：

1. [Python 教程 (w3school.com.cn)](https://www.w3school.com.cn/python/index.asp)