计算物理实验作业

## π的计算

### 题目

编写C和Python程序利用莱布尼茨公式、欧拉公式、拉马努金公式计算的近似值

### 理论分析

莱布尼茨公式

在处将展开为泰勒级数

令得到

欧拉公式

拉马努金公式

### 程序设计思路

莱布尼茨公式可以直接用通项判断精度，也即用此算法计算的收敛精度可表示为

在迭代次数不太大时，收敛精度的近似公式可表示为

与上式相对比得到：

不小，也不大，因此此算法的收敛速度较慢

实际上，莱布尼茨公式的截断误差可以通过下面的证明精确地得到：

将在处展开为带有拉格朗日余项的泰勒公式得到：

令

其中，而

在时恒成立，因此在此区间内单调递减

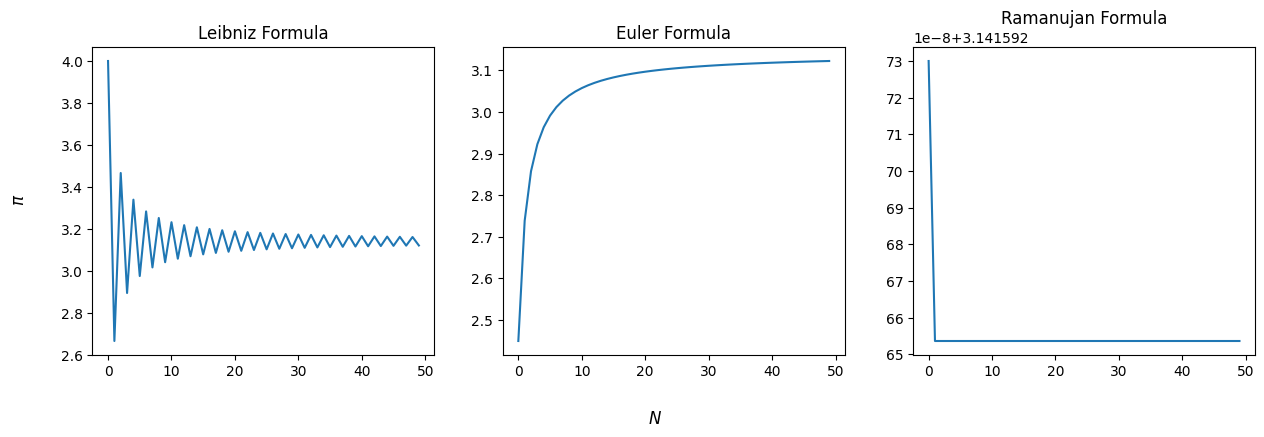
考虑到

所以同样有

这样截断误差就与收敛精度相统一了，可以看出莱布尼茨公式的精度半径在通项绝对值范围内，这也体现了莱布尼茨公式所对应的级数是交错级数的这一特殊性

至于欧拉公式和拉马努金公式公式，他们的余项函数的性质不太好，难以进行分析，因此在程序设计中，莱布尼茨公式采用通项的绝对值来判断精度，而欧拉公式与拉马努金公式则采用给定的值3.14159265358979323846来计算判断精度

### 输出结果与程序分析



方便起见，同时又为了体现不同的程序实现方案，Python程序使用迭代次数作为函数参数而C语言程序使用计算精度作为函数参数

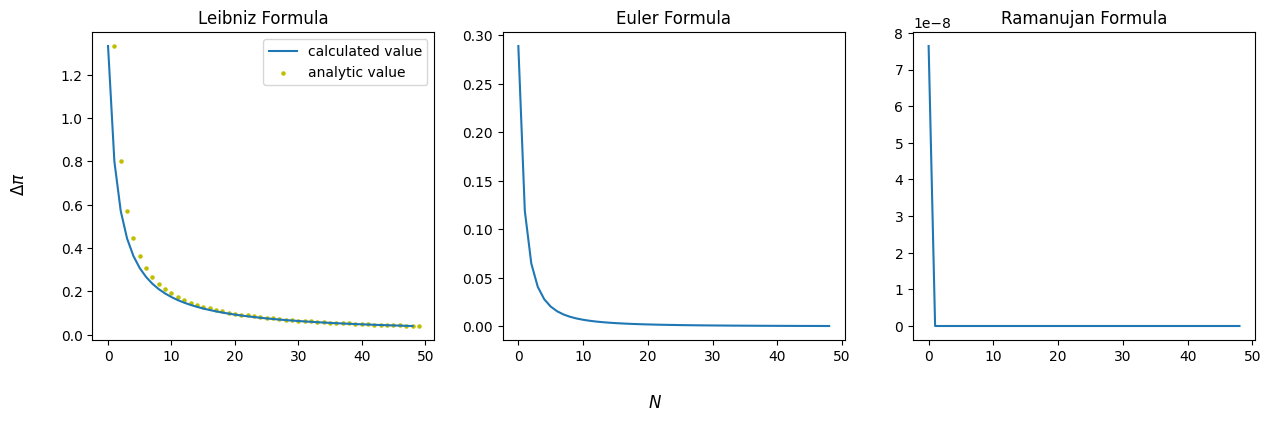
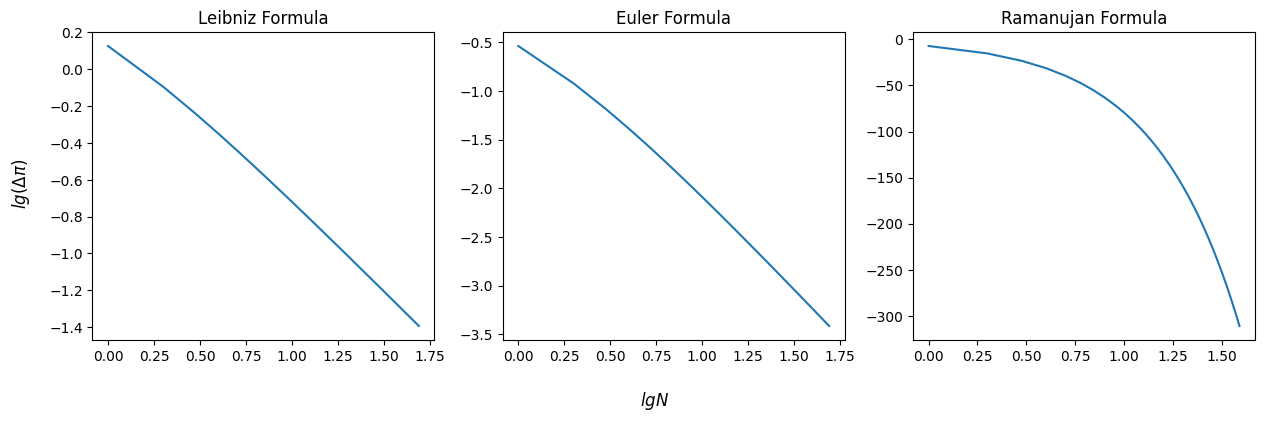
在程序设计时需要注意，迭代参数的初始值设定，是否应该从0开始迭代需要仔细考虑

通过对比C语言与Python程序的运行时间可以发现C语言跑循环比Python快得多

从上面展示的图像结果中可以发现，拉马努金公式效率最高，只需很少的迭代次数就可以达到很高的精度，但受限于Python的计算精度限制，使用一般的方法难以看出拉马努金公式的威力，在附录中提供了一种方式，采用Python中的decimal模块，利用拉马努金公式将π的精确值计算到了小数点后上千位而程序用时不到1s！

而对于莱布尼茨公式与欧拉公式，计算效率就很慢了，利用Python计算，莱布尼茨公式需要近两百万次迭代才能达到的精度，用时7.9s，近两千万次的迭代达到精度，用时10.6s，欧拉公式在近三十万次的迭代后，达到精度，用时0.2s，在迭代三百万次后达到精度，用时1.5s，在迭代三千万次后，精度达，用时12.6s

下图是对相邻的计算值作差得到,考察差值随迭代次数的变化，从图中可以发现拉马努金公式迅速收敛，而莱布尼茨与欧拉公式收敛速度较慢



为了得到更加精确的收敛速度，对取对数，得到，做关于迭代次数的图像，得到上图所示的变化关系，根据以下代码，与收敛精度的近似公式做比较可得莱布尼茨公式近似为一阶收敛，与理论值符合较好，欧拉公式近似为二阶收敛，而拉马努金公式近似为220阶收敛，计算效率是莱布尼茨公式的近200倍！，是欧拉公式的近100倍！这再一次证明了拉马努金公式超高的计算效率！

1. #计算收敛精度中的β,也就是计算效率
2. from numpy import polyfit
3. coeff=polyfit(lgn,lg\_Delta\_Leibniz,1)
4. beta\_Leibniz=-coeff[0]
5. print('莱布尼茨公式:beta=%f'%beta\_Leibniz)
6. coeff=polyfit(lgn,lg\_Delta\_Euler,1)
7. beta\_Euler=0-coeff[0]
8. print('欧拉公式:beta=%f'%beta\_Euler)
9. coeff=polyfit(lgn\_Ramanujan,lg\_Delta\_Ramanujan,1)
10. beta\_Ramanujan=-coeff[0]
11. print('拉马努金公式:beta=%f'%beta\_Ramanujan)

## 天体运动

### 题目

通过计算行星在平方反比引力场中的运动，掌握用数值方法求解常微分方程的一般步骤，并对结果进行分析，理解行星运动规律。

### 理论分析

令

运动方程为

其中

总能量为

当时轨迹为椭圆，时轨迹为抛物线，时，轨迹为双曲线

### 程序设计思路

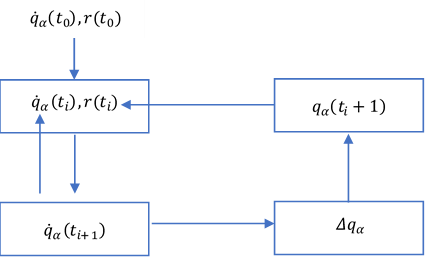
将时空离散化并将运动方程写成分量形式得到

而

从而得到

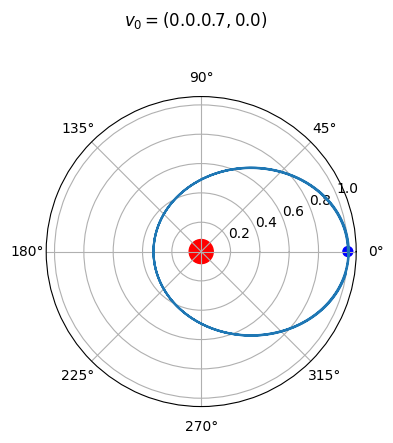
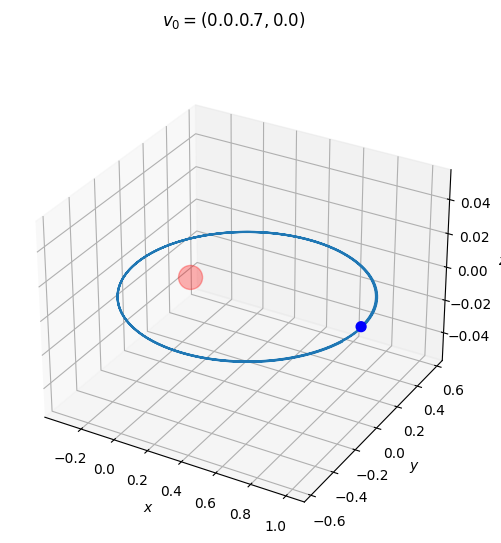
之后再算下一时刻的力,重复上述步骤，注意上式中力的公式中的负号，表示吸引力

化成图表的形式有(令)：

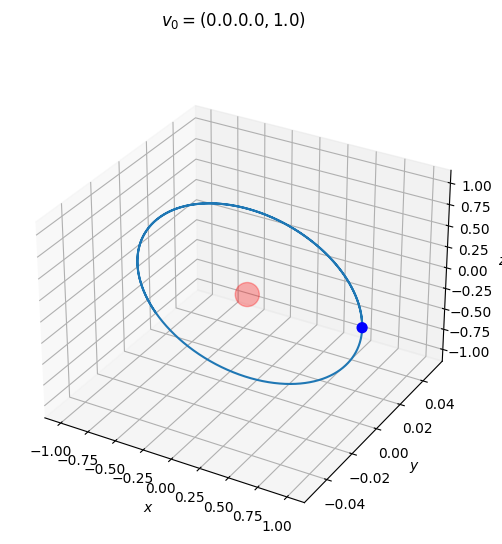
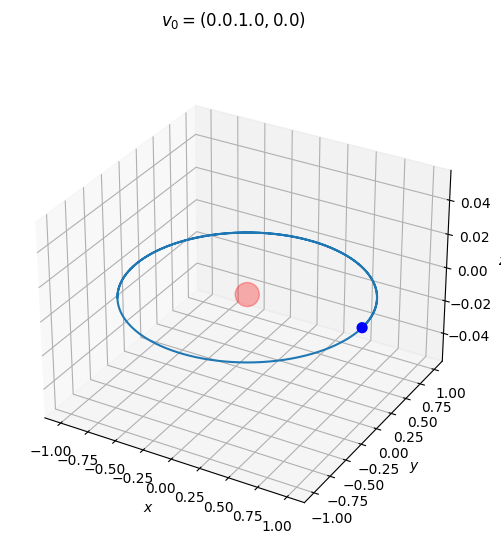


### 输出结果与程序分析

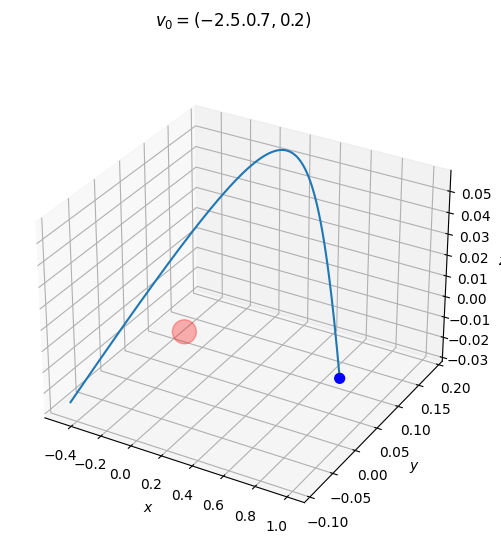
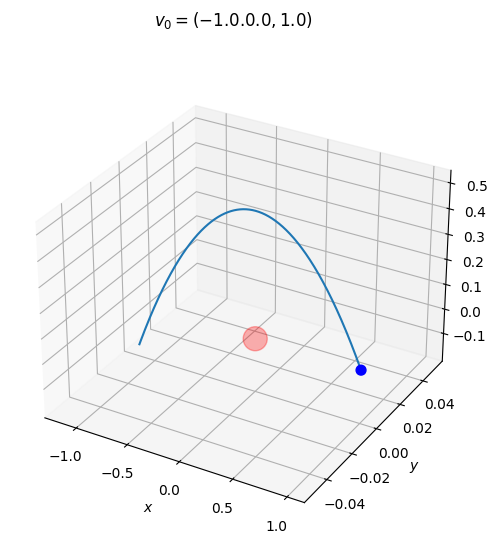
下图是椭圆轨迹的三维视图(左)与极坐标视图(右)



选取恰当的初始速度可以得到正圆轨迹，如下图所示



继续改变初速度从而改变初始能量还可以得到其他不同的轨迹，下图中左方图为抛物线轨迹，右图为双曲线轨迹



在此程序中解常微分方程用到的是欧拉法，因为此方法精度有限，所以需要设置足够多的离散点，也就是说Nt要取得足够大，才能实现对行星受力运动的精确模拟，这样画出来的图像才近似正确，若取的离散点的个数过少会发现行星的运动轨道向着中心收缩再逐渐稳定，这显然是不符合物理学规律的

为了提高运算精度，缩短程序运行时间，可以采用以下方法

1. 选用更加精确的求解常微分方程的方法，如改进欧拉法、龙格库塔法
2. 使用scipy模块的ode求解函数
3. 调节M，r0等参数缩短行星运行周期，从而提高离散点的相对数目

这个题目中在由直角坐标向极坐标转换时可能会想到使用反三角函数求解各个离散的角度，但Python中的反三角函数运算有定义域的限制，因此这样画出来的图像是不完整的，根据CSDN上的文章，将原来直角坐标中的x与y在复平面内表示，用复数的方法求解幅角与模长才能在极坐标中画出完整的运动轨迹

## 伽尔顿板实验

### 题目

假设有一个钉子组成的等间距的垂直阵列，小球从上部滚落下来，碰到一个钉子后将等概率向左或右运动。为了简便，再假定向左运动的小球将碰到下一排左边一格的钉子，向右运动的小球将碰到下一排右边一格的钉子。小球一直这样落下去，直到底部。用模拟法证明，即使小球都从一个位置下落，在底部也会形成正态分布。

### 理论分析

设小球下落的高度为下落的小球个数为，以横拍钉子间距作为小球下落高度的单位，那么在数值上与纵向排列的钉子数目相同，对于每一个小球，假设在下落的过程中有次在碰到钉子后向轴正向移动，另外次在碰到钉子后向轴负向移动那么小球正向移动次的概率为

可见满足二项式分布

而与总位移的关系为

那么也服从二项分布，只不过各概率所对应的数值不同

根据概率论的知识，当时服从正态分布

### 程序设计思路

利用随机数生成位移矩阵，矩阵的元素表示第个小球在碰到第个钉子后产生的位移,值得注意的是，实际的伽尔顿板上钉子的排列方式是下一排的钉子对准上一排钉子的空隙，整体呈三角形，所以小球每碰到一个钉子后的位移大小应是0.5个下落高度的单位而非1个，考虑到小球在碰到钉子后要么向左移动，要么向右移动，所以，位移矩阵的例子如下

这个位移矩阵描述的是五个小球经过六排钉子后的位移矩阵

对该矩阵的每一列求和就可以得到五个小球的最终位置坐标的列表

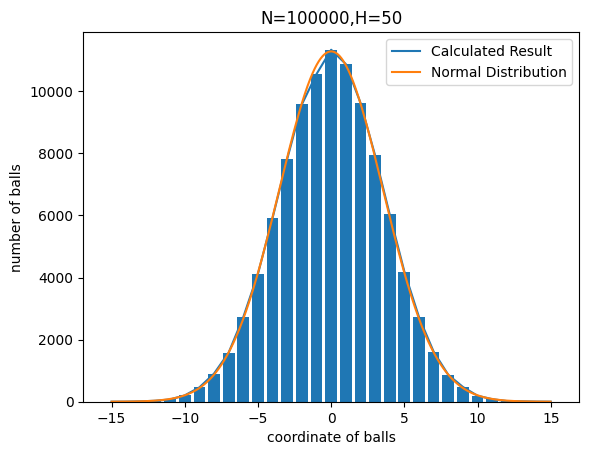
从该最终位置坐标列表中筛选出所有不重复的位置坐标并按大小排序,作为五个小球所有的位置坐标集合

以及每一坐标所对应的小球数目集合

根据与就可以画出各个坐标位置与小球数目对应关系的图像

为了便于得到小球的累计位移并统计处于不同位置的小球数目，考虑使用pandas模块，其中的sum()与unique()可以很好地满足上述需求

### 输出结果与程序分析

程序中的关键参数有

1. H：相当于二项分布中的样本数目，当它越大时二项分布就越接近正态分布
2. N：实验小球的数目，它越大所画出的曲线就越光滑

从结果图中可以发现实验结果与标准正态分布重合度较高，说明了模型的可靠性

## 带电圆环的电场和电势

### 题目

通过求解带电圆环周围的电场，理解数值计算积分的方法,并掌握数值求解积分函数，学会使用不同的方法显示最终结果。

### 理论分析

真空环境中，带电量为的点电荷在空间某点处产生的电场与电势可以用下面的积分描述

将电场写成分量的形式有：

将连续带电的物体离散化，看作由一个一个点电荷构成，那么根据上面点电荷的公式，空间某点处的电势与电场可表示为

其中记录构成带电体的点电荷而表证各个分量

这样就可以累加的方式求得电场电势

### 程序设计思路

可以直接利用for循环，套用上述公式进行累加求得(代码详见附录)，但考虑到Python运行循环的效率低下，所以采用矩阵承载数据，用空间换时间，提高程序运行效率，具体思路如下

首先计算点电荷在空间产生的电场

给定电场电势的计算范围，将这个空间离散化，然后得到在此范围每一点的横纵坐标矩阵，例如：

再给定源点坐标例如

这样就可以得到源点到各个场点的径矢所构成的矩阵与

由此可以得到源点到各个场点的距离矩阵（以下的平方开方运算是数组运算）

这样就可以求得电势在所考察空间中的矩阵(设，以下的除法运算也是数组计算，并保留一位小数)

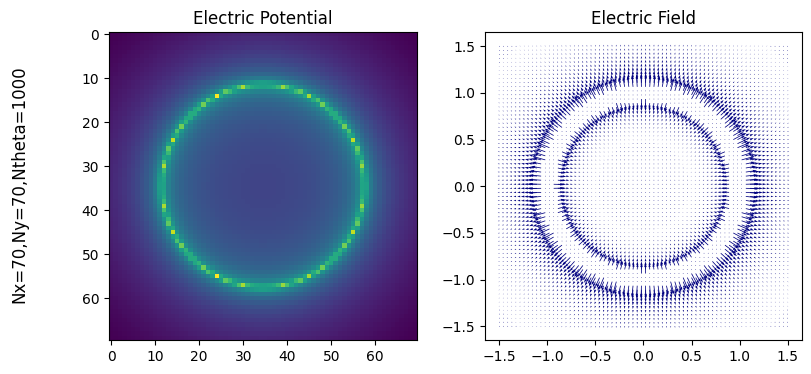
其中发散处()的位置正好点电荷所在的位置，对于电场各分量的矩阵求法与电势类似

对于连续带电物体，只要得到该带电体离散化后的空间坐标，以及与之相对应的电荷列表：

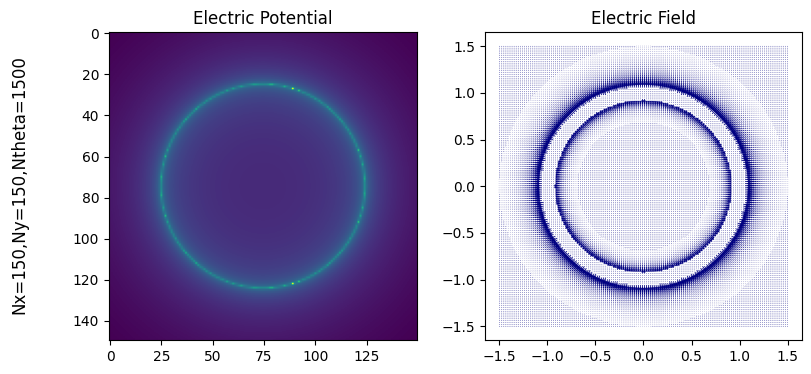
即可通过循环累加求得电场的各个分量与电势在考察空间的数值矩阵

在程序设计时，使用了大量的自定义函数，这样通过改变自定义函数的形式可以方便地得到不同带电体的电场电势分布，提高了程序的扩展性

### 输出结果与程序分析



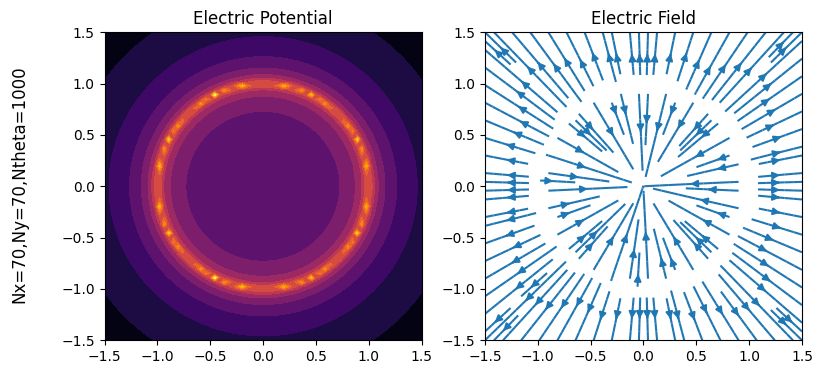
通过增大Nx，Ny，Ntheta可以增大图片的清晰度与电势图的亮度，可得下图



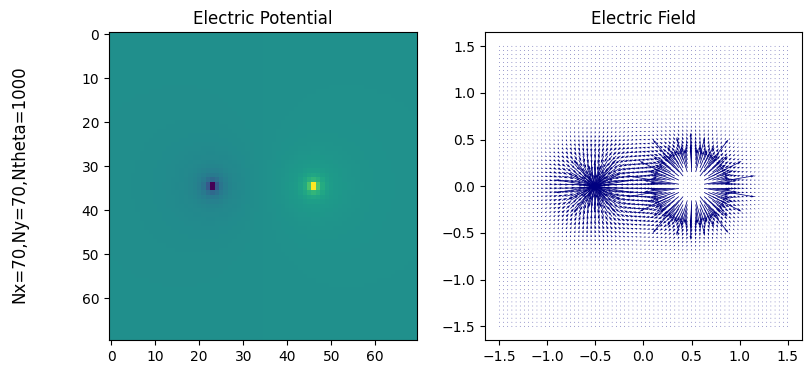
可以发现电势图更加清晰、明亮而箭头图更加密集

使用以下程序还可以画电势的等高线图以及电场的流线图

1. fig=plt.figure(figsize=(9,4))
2. ax1=fig.add\_subplot(1,2,1)
3. ax1.contourf(xlist,ylist,Phimat,15,cmap='inferno')
4. ax1.set\_title('Electric Potential')
5. ax2=fig.add\_subplot(1,2,2)
6. ax2.streamplot(xlist,ylist,Exmat,Eymat,density=(1.2,1.2))
7. ax2.set\_title('Electric Field')
8. plt.show()



使用不同带电体的函数可以画出不同带电体产生的电场与电势分布图，下面是两个带相反异号电的点电荷的电场电势分布图



上述程序主要采用矩阵的数据结构,使用一次循环，大大提升了运行效率，在与上述程序主要参数相同的情况下，附录程序运行时间为31.7s而上述程序运行时间仅为1.5s

此程序中所用到的这一个循环目前还是必须的，考虑到带电体的任意性，暂时还未想到可以不用循环实现多对多(带电体列表与点电荷函数)的匹配方式

另外参数std(standard)的设置至关重要，它表示需要剔除的点的索引，只有剔除适当的点用于画图的数据才不会过于发散，数据信息才能保留完全，输出的图像才正常、清晰

## 波动方程

### 题目

通过显示差分公式求解弦振动方程，掌握用数值方法求解偏微分方程的一般步骤，并进一步研究稳定条件。

### 理论分析

波动方程

初始条件

边界条件

边界条件与初始条件之间在交界处应满足条件

### 程序设计思路

将原方程时空离散化

令

将这个线性方程组写成矩阵的形式可以得到相邻时间间隔内的递推关系

是的矩阵

利用傅里叶变换改写原矩阵方程

矩阵的本征值与本征向量分别设为与,且有

其中

同理可分解边界条件

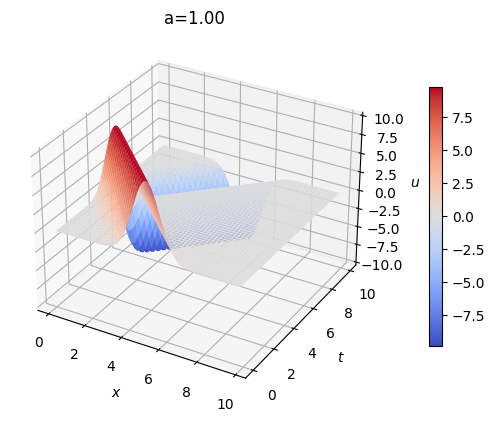
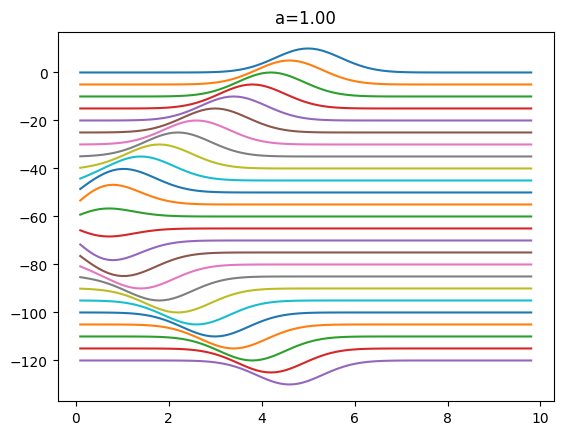
这样原来的矩阵方程中的向量全由矩阵的本征矢组成，因此可以替换为

中第个本征向量前的系数满足关系

求得了就可以知道

只需取前一部分的就可以实现较好的效果，这样就大大减小了计算量

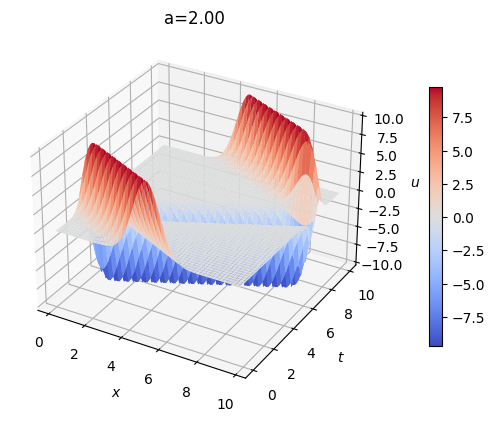
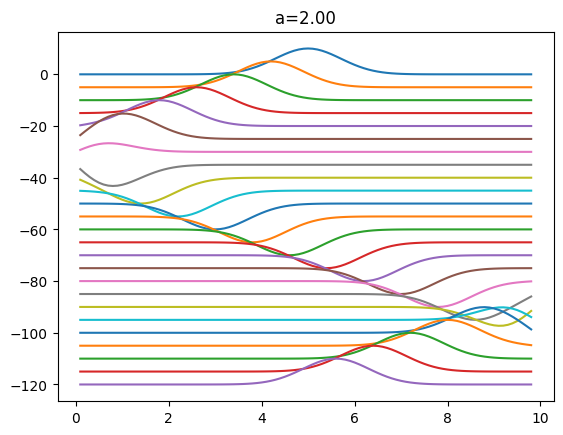
### 输出结果与程序分析



上方作图中最上方的曲线是时刻的波形，随着时间的推移，波向左移动然后被反弹回来，继续向右移动，可以发现反弹前的波与反弹后的波相位相乘，这正是半波损失现象

此程序在设计时需要特别注意参数alpha和beta的选取，若选地太大，方程就会指数爆炸，算出错误的结果，另外这两个参数与时空步长deltat与deltx紧密联系，在写程序时，为确保alpha与beta取恰当的值，采用先确定alpha与beta然后再由他们确定deltat与deltax的方式而非先确定deltat与deltax

当改变参数a的值时，波的运动速度也会发生改变，如下图所示，波经过了两次反射，而a=1时，波只经过了一次反射



考虑到此程序所计算的数据量并不是特别的大(运行时间不超过1s)，为简便起见，并没有采用理论分析中的傅里叶变换方法减小计算量

上述程序只适用于齐次方程、第一类边界条件的情况，对于非齐次方程、周期性边界条件或其他类型的边界条件，他们所对应的矩阵lambda不尽相同，因此难以做到兼容

## 传热方程

### 题目

通过显示差分公式求解热传导方程，掌握用数值方法求解偏微分方程的一般步骤，并进一步研究稳定条件。

### 理论分析

热传导方程

初始条件

边界条件

边界条件与初始条件之间在交界处应满足条件

### 程序设计思路

将原方程时空离散化

令

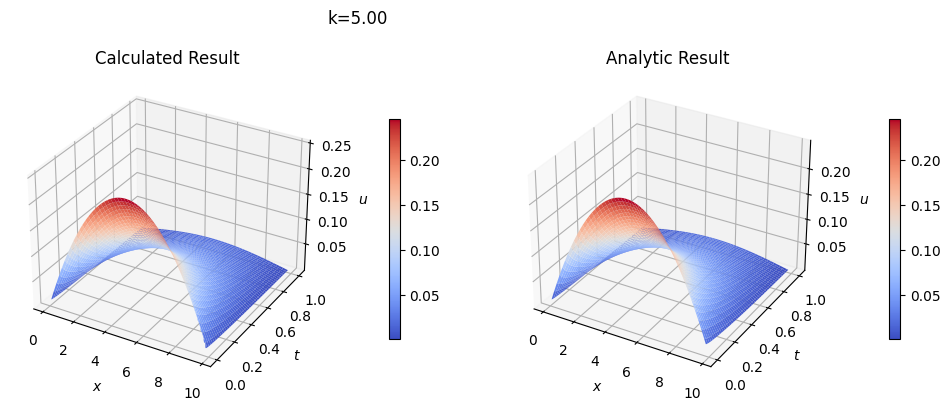
将这个线性方程组写成矩阵的形式可以得到相邻时间间隔内的递推关系

是的矩阵

一个例子(梁昆淼数学物理方法第四版第八章第一节课后习题2)

解析解为：

### 输出结果与程序分析



计算结果与理论结果在同一时刻的温度都是两端低中间高，计算结果较为理想

热传导方程的程序同样也需要格外注意参数alpha与beta的设置，与波动方程不同的是，取alpha=beta=1并不能使方程收敛，虽然这对于波函数而言是在合适不过的值，只有当alpha与beta都小于1时原方程才收敛

类似于波动方程，参数k也影响传热速率，当k较大时三维图像会更快地衰减到零

同样地，上述程序只适用于常系数齐次方程、第一类边界条件、一维的情况

## 亥姆霍兹方程

### 题目

仿照二维Helmholtz方程求解示例编写求解三维Helmholtz方程代码

### 程序设计思路

#### 蛮力法求解

边界条件

若具有分离变量的形式:

同理

若已知即可分别求出再由之得

上述分析只适用于可以分离变量的、第一类边界条件、为常数的情况

作为一个例子，求解模式下谐振腔中磁场方向分量的分布，它满足的亥姆霍兹方程是

边界条件为()

解析解为：

在程序实现时为简便起见又不失程序要点,假设已知即

再结合原来的解析表达式(令)将第二类边界条件转换为第一类边界条件,这样满足亥姆霍兹方程与边界条件为

将此方程时空离散化得到(用符号代替)

令

将上式写成矩阵方程

是的矩阵

#### 超松弛法求解

蛮力法求解所使用的矩阵需要消耗大量的存储空间，特别是当维度提高到三维时，数据的体量很容易大到无法计算的地步，为解决维数爆炸的问题，我们转换思路，用时间换空间，进行求解，而为了尽可能多地减少迭代次数，采用超松弛算法，整个程序的设计基于元胞自动机，以元胞为关注点，建立元胞数据类型，通过改变该数据类型的属性与方法可以很方便地实现程序的扩展。

为了使求解的方程更加一般化，将亥姆霍兹方程泛化为下式

差分后得到

为方便化简，令

为加快迭代速度，使尽快地趋近于0，得到迭代表达式为：

理论告诉我们当满足下式时迭代效率最高

而根据上面的公式进行计算：

其中表示迭代次数

当用元胞自动机实现时，即为所考察的元胞在迭代次后的状态，根据上式可知，元胞通过与它相邻的上下左右前后共计六个元胞进行迭代，通过改变类的方法(迭代方式)可以很容易地得到按不同迭代方式形成的结果

当时，原方程变为泊松方程

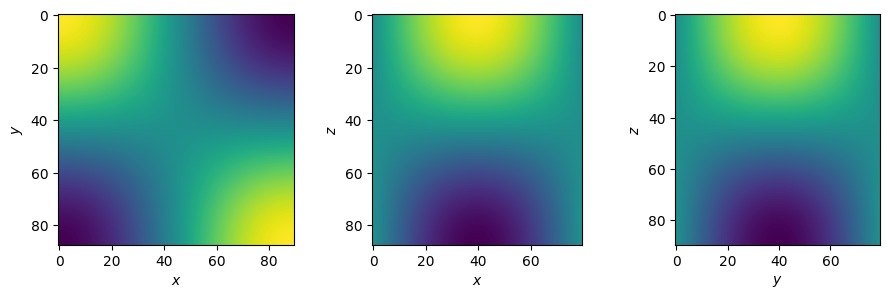
接下来的就以求解泊松方程为例，计算点电荷、电偶极子、带电正方体放在接地导体球中时导体球内的电势分布，特别地，对于带电量为的放在球心处的点电荷，球内电势的解析解为：

其中为考察点与球心的距离，为导体球的半径，通过计算数据与解析解相比较可以在一定程度上验证程序的正确性

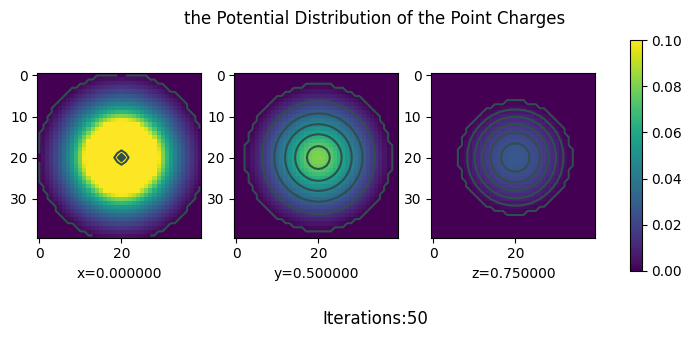
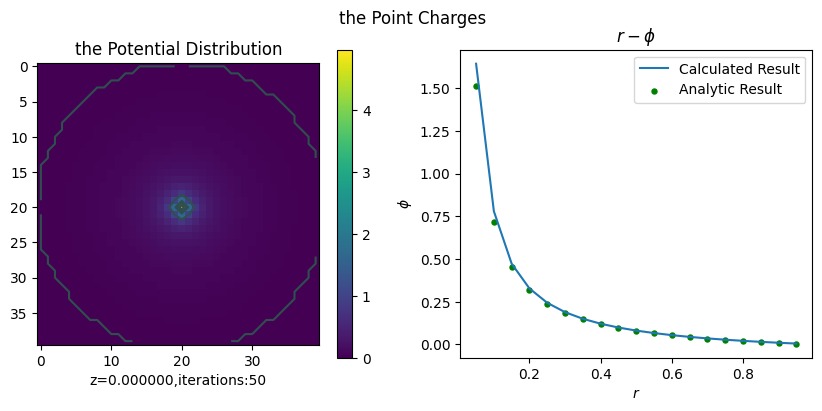
### 输出结果与程序分析

蛮力法求解结果

下图是用蛮力法求解经简化后的谐振腔模型

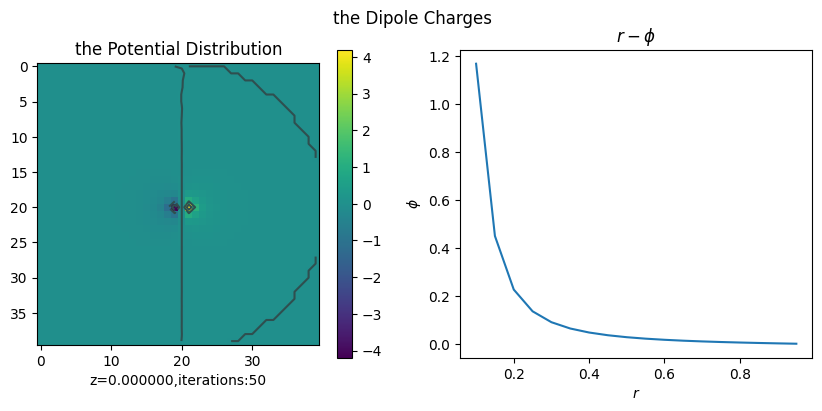


超松弛法求解结果

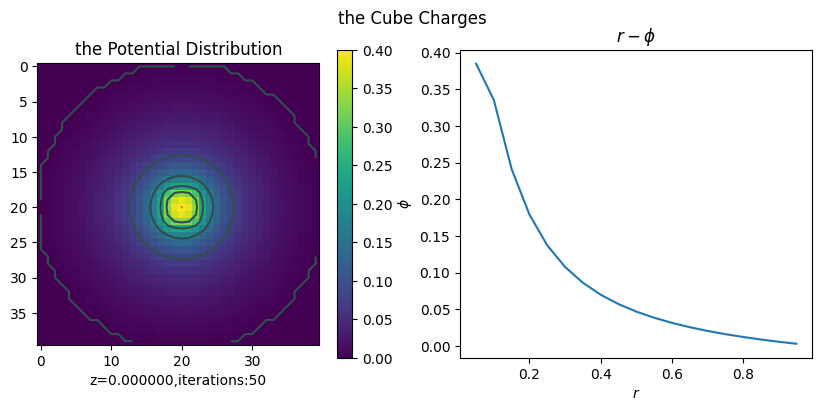


以上两幅图是用超松弛法求解接地导体中心内置点电荷时导体内部的电势分布图像，从上往下第一个图中左图是处的切面视图，右图是电势大小随半径的分布图像，折线图是数值计算结果，而绿色散点图是理论计算结果，可见数值计算值与理论值符合地很好，说明了数值计算模型的可靠性，从上往下第二个图是电势分布的三视图，可以发现随着考察面与球心的距离逐渐增大，电势分布的范围(最外层的等式面)在逐渐缩小，电势的强度(用颜色的深浅表示)也在逐渐变弱，这正是三维图像的特征。

计算时迭代次数为50次，每十次迭代的收敛精度依次为：9.498888e-04、5.777043e-04、2.002309e-04、2.191077e-05、2.916483e-06，可见使用超弛豫法可使计算结果很快地收敛

通过更改参数可以得到电偶极子的电视分布图像，如下

可以发现电偶极子产生的电势比点电荷产生的电势衰减地更快，这与理论是相符的

通过数值计算方法，还可以计算任意带电体的电势分布，而这是使用严格的理论推导难以得出的，下图是位于中心的正方体形的电荷产生的电势

## **三个偏微分方程**

上面所列举的微分方程兼容性都不太强，下面给出一个兼容性较强的微分方程，它可以兼并一维的亥姆霍兹方程、热传导方程、以及薛定谔方程，理论分析与程序设计思路如下

令

令

选择合适的时空步长：

当为一常数时，原差分方程变为

或者令

得到：

为避免发散，选择合适的时空步长

当为一常数时，原差分方程变为

若令得到亥姆霍兹方程

若令得到没有热源的热传导方程

若令得到薛定谔方程

## 附录

### 的计算

#### 用Python计算π的主程序

1. import Pi\_Func
2. l=50
3. Leibniz\_list=Pi\_Func.Leibniz(l)
4. Euler\_list=Pi\_Func.Euler(l)
5. Ramanujan\_list=Pi\_Func.ramanujan\_by\_igfasouza(l)
6. #输出结果
7. print(
8. '莱布尼茨公式计算结果:%f,精确度为%.0e'
9. %(Leibniz\_list[l-1],abs(Leibniz\_list[l-1]-Leibniz\_list[l-2]))
10. )
11. print(
12. '欧拉公式计算结果:%f,精确度为%.0e'
13. %(Euler\_list[l-1],abs(Euler\_list[l-1]-Euler\_list[l-2]))
14. )
15. print(
16. '拉马努金公式计算结果:%f,精确度为%.1e'
17. %(Ramanujan\_list[l-1],abs(Ramanujan\_list[l-1]-Ramanujan\_list[l-2]))
18. )
19. #作图
20. import matplotlib.pyplot as plt
21. fig,axes=plt.subplots(1,3,figsize=(15,4))
22. fig.supylabel('$\pi$',x=0.07)
23. fig.supxlabel('$N$',y=-0.07)
24. axes[0].plot(Leibniz\_list)
25. axes[0].set\_title('Leibniz Formula')
26. axes[1].plot(Euler\_list)
27. axes[1].set\_title('Euler Formula')
28. axes[2].plot(Ramanujan\_list)
29. axes[2].set\_title('Ramanujan Formula')
30. plt.show()

#### Python计算的其他作图程序

#delta\_pi随迭代次数N的变化图像

1. import numpy as np
2. Nlist=np.arange(l)
3. Delta\_Leibniz\_analy=np.divide(4,2\*Nlist+1)
4. Delta\_Leibniz=abs(Leibniz\_list[1:]-Leibniz\_list[:-1])
5. Delta\_Euler=abs(Euler\_list[1:]-Euler\_list[:-1])
6. Delta\_Ramanujan=np.zeros(l-1)
7. for i in range(l-1):
8. Delta\_Ramanujan[i]=format(abs(Ramanujan\_list[i+1]-Ramanujan\_list[i]))
9. fig,axes=plt.subplots(1,3,figsize=(15,4))
10. fig.supylabel('$\Delta\pi$',x=0.07)
11. fig.supxlabel('$N$',y=-0.07)
12. axes[0].plot(Delta\_Leibniz,label='calculated value')
13. axes[0].scatter(Nlist[1:],Delta\_Leibniz\_analy[1:],s=5,label='analytic value',c='y')
14. axes[0].set\_title('Leibniz Formula')
15. axes[0].legend()
16. axes[1].plot(Delta\_Euler)
17. axes[1].set\_title('Euler Formula')
18. axes[2].plot(Delta\_Ramanujan)
19. axes[2].set\_title('Ramanujan Formula')
20. plt.show()

#lg\_delta\_pi的对数随迭代次数的对数变化的图像

1. start=0
2. list=np.arange(1,l)#从1开始迭代
3. lgn=np.log10(list[start:])
4. lgn\_Ramanujan=np.log10(list[start:39])
5. lg\_Delta\_Leibniz=np.log10(Delta\_Leibniz[start:])
6. lg\_Delta\_Euler=np.log10(Delta\_Euler[start:])
7. lg\_Delta\_Ramanujan=np.log10(Delta\_Ramanujan[start:39])
8. fig,axes=plt.subplots(1,3,figsize=(15,4))
9. fig.supylabel('$lg(\Delta\pi)$',x=0.07)
10. fig.supxlabel('$lgN$',y=-0.07)
11. axes[0].plot(lgn,lg\_Delta\_Leibniz)
12. axes[0].set\_title('Leibniz Formula')
13. axes[1].plot(lgn,lg\_Delta\_Euler)
14. axes[1].set\_title('Euler Formula')
15. axes[2].plot(lgn\_Ramanujan,lg\_Delta\_Ramanujan)
16. axes[2].set\_title('Ramanujan Formula')
17. plt.show()

#### 用Python计算π的函数文件

1. from math import factorial, sqrt#不用引入pow
2. import numpy as np
3. # 注意公式迭代的初始值
4. def Euler(l):
5. PiList=np.zeros(l)
6. s=0
7. for n in range(1,l+1):
8. an = 1/n\*\*2
9. s = s+an
10. PiList[n-1]=sqrt(s\*6)
11. return PiList
12. def Leibniz(l):
13. PiList=np.zeros(l)
14. s=0
15. for n in range(1,l+1):
16. an=an=1/(2\*n-1)\*pow(-1,n-1)
17. s=s+an
18. PiList[n-1]=4\*s
19. return PiList
20. def Ramanujan\_by\_myself(l):
21. PiList=np.zeros(l)
22. s=0
23. k=sqrt(8)/pow(99,2)
24. for n in range(0,l):
25. an=factorial(4\*n)\*(1103+26390\*n)/pow(factorial(n),4)/pow(396,4\*n)
26. s=s+an
27. PiList[n]=1/(k\*s)
28. return PiList
29. from math import factorial,sqrt,log10
30. from decimal import Decimal, getcontext
31. import mpmath as mp
32. mp.dps = 100000  # number of digits
33. def ramanujan\_by\_igfasouza(max\_step):
34. """ Computing an approximation of pi with a Ramanujan's formula."""
35. getcontext().prec = max\_step\*10  # trick to improve precision
36. PiList=[]
37. Sum = Decimal(0)
38. d\_1103 = Decimal(1103)
39. d\_26390 = Decimal(26390)
40. d\_396 = Decimal(396)
41. for k in range(max\_step):
42. Sum += ((Decimal(factorial(4 \* k))) \* (d\_1103 + d\_26390 \* Decimal(k))) / ( (Decimal(factorial(k)))\*\*4 \* (d\_396\*\*(4\*k)))
43. my\_pi\_multiple\_factor = Sum \* 2 \* Decimal(2).sqrt() / Decimal(9801)
44. my\_pi\_reciprocal  = my\_pi\_multiple\_factor\*\*(-1)
45. PiList.append(my\_pi\_reciprocal)
46. return PiList

#### 用C语言计算π的主程序

1. #include <stdio.h>
2. #include <math.h>
3. #include "pi\_Func.h"
4. int main()
5. {
6. double precision=1e-4;
7. double result\_Leibniz=pi\_Leibniz(precision);
8. double result\_Euler=Euler(precision);
9. double result\_Ramanujan=Ramanujan\_calc(precision);
10. printf(
11. "result\_Leibniz=%f\nresult\_Euler=%f\nresult\_Ramanujan=%lf"
12. ,result\_Leibniz,result\_Euler,result\_Ramanujan
13. );
14. return 0;
15. }

#### 用C语言算π的函数文件

1. /\*莱布尼茨公式是交错级数可以直接用an来判断精度,但欧拉公式与拉马努金公式都不能用an或sn+1-sn来判断,
2. 为简便起见,在这里用已知的标准数值3.1415926535进行判断\*/
3. double pi\_Leibniz(double precision)
4. {
5. double n,an=10,s=0,tmp=precision+1;
6. for(n=1;fabs(tmp)>precision;n++)
7. {
8. an=1/(2\*n-1)\*pow(-1,n-1);
9. s=s+an;
10. tmp=an\*4;
11. }
12. return 4\*s;
13. }
14. double Euler(double precision)
15. {
16. double an=0,n,s=0;
17. for(n=1;fabs(sqrt(s\*6)-3.1415926535)>precision;n++)
18. {
19. an=1/pow(n,2);
20. s=s+an;
21. }
22. return sqrt(s\*6);
23. }
24. double Ramanujan\_calc(double precision)
25. {
26. double fact(double n);//函数要提前申明
27. double n,k,s=0,an=0;
28. k=sqrt(8)/pow(99,2);
29. for(n=0;fabs(1/(k\*s)-3.1415926535)>precision;n++)
30. {
31. an=fact(4\*n)\*(1103+26390\*n)/pow(fact(n),4)/pow(396,4\*n);
32. s=s+an;
33. }
34. return 1/(k\*s);
35. }
36. double fact(double n)
37. {
38. double output=1;
39. int i;
40. for(i=1;i<=n;i++)
41. {
42. output=output\*i;
43. }
44. return output;
45. }

### 一种利用拉马努金公式计算高精度圆周率的程序实现

1. from math import factorial,sqrt,log10
2. from decimal import Decimal, getcontext
3. import mpmath as mp
4. mp.dps = 100000  # number of digits
5. len(str(mp.pi))
6. mpmath\_pi = Decimal(str(mp.pi))
7. def ramanujan(max\_step):
8. """ Computing an approximation of pi with a Ramanujan's formula."""
9. my\_pi = Decimal(0)
10. d\_1103 = Decimal(1103)
11. d\_26390 = Decimal(26390)
12. d\_396 = Decimal(396)
13. for k in range(max\_step):
14. my\_pi += ((Decimal(factorial(4 \* k))) \* (d\_1103 + d\_26390 \* Decimal(k))) / ( (Decimal(factorial(k)))\*\*4 \* (d\_396\*\*(4\*k)))
15. my\_pi = my\_pi \* 2 \* Decimal(2).sqrt() / Decimal(9801)
16. my\_pi = my\_pi\*\*(-1)
17. return my\_pi
18. getcontext().prec = 2000  # trick to improve precision
19. my\_pi = ramanujan(200)
20. print(my\_pi)
21. accuracy = 100\*abs(mpmath\_pi - my\_pi)/mpmath\_pi
22. print("Accuracy % with mpmath\_pi: {:.4g}".format(accuracy))
23. print("Accuracy % with mpmath\_pi:%".format(accuracy))

的计算结果

3.1415926535897932384626433832795028841971693993751058209749445923078164062862089986280348253421170679821480865132823066470938446095505822317253594081284811174502841027019385211055596446229489549303819644288109756659334461284756482337867831652712019091456485669234603486104543266482133936072602491412737245870066063155881748815209209628292540917153643678925903600113305305488204665213841469519415116094330572703657595919530921861173819326117931051185480744623799627495673518857527248912279381830119491298336733624406566430860213949463952247371907021798609437027705392171762931767523846748184676694051320005681271452635608277857713427577896091736371787214684409012249534301465495853710507922796892589235420199561121290219608640344181598136297747713099605187072113499999983729780499510597317328160963185950244594553469083026425223082533446850352619311881710100031378387528865875332083814206171776691473035982534904287554687311595628638823537875937519577818577805321712268066130019278766111959092164201989380952572010654858632788659361533818279682303019520353018529689957736225994138912497217752834791315155748572424541506959508295331168617278558890750983817546374649393192550604009277016711390098488240128583616035637076601047101819429555961989467678374494482553797747268471040475346462080466842590694912933136770289891521047521620569660240580381501935112533824300355876402474964732639141992726042699227967823547816360093417216412199245863150302861829745557067498385054945885869269956909272107975093029553211653449872027559602364806654991198818347977535663698074265425278625518184175746728909777727940092593679577412880114072520662452344498309284389026310583057432459663517241665671357564259409181627161036619398380266804323123256475817186916514676524019320976885611753635255011894984447618075599914209997043044425803288826839697424724950586712681051712136647565220069328838660790206792014356556715545181106432983263815336472301873472006826126207435983354628153181952126467499415321188698738323941189429

### 天体运动

1. #相当于MATLAB中的clear,用于清空变量区
2. # [-1.4,0.2,0.1]
3. import numpy as np
4. #TODO 初始化
5. G,M,m=1,1,0.01
6. k=G\*M\*m
7. Nt,tN=int(1e5),10
8. # r0,v0=[1,0,0],[-2.5,0.7,0.2]#要和维数(dim/dimention)保持一致
9. r0,v0=[1,0,0],[0,0.7,0]
10. tlist=np.linspace(0,tN,Nt)
11. deltat=tlist[1]-tlist[0]
12. r=np.zeros((Nt,3))
13. v=np.zeros((Nt,3))
14. r[0,:]=r0
15. v[0,:]=v0
16. # v[0,:],v[1,:]=v0,v0
17. #TODO 计算数据
18. for i in range(Nt-1):
19. norm=np.linalg.norm(r[i,:])
20. for j in range(3):
21. q=r[i,j]
22. v[i+1,j]=-k\*q\*deltat/norm\*\*3/m+v[i,j]#下一时刻的速度分量,注意负号表示吸引力
23. deltaq=(v[i+1,j]+v[i,j])/2\*deltat
24. r[i+1,j]=q+deltaq#下一时刻的坐标分量
25. #结果分析
26. norm=np.sqrt(r\*\*2@np.ones(3))
27. a,b=np.max(norm),np.min(norm)
28. v0norm,r0norm=np.linalg.norm(v0),np.linalg.norm(r0)
29. L=np.linalg.norm(np.cross(r0,v0))
30. E=0.5\*m\*v0norm\*\*2-k/r0norm
31. print(f'能量为{E},角动量为{L}')
32. if E<0:
33. print('轨迹为椭圆')
34. print(f'近日点={a},远日点={b}')
35. elif E==0:
36. print('轨迹为抛物线')
37. else:
38. print('轨迹为双曲线')
39. #TODO 作图
40. import matplotlib.pyplot as plt
41. fig=plt.figure(figsize=(6,6))#设置大小
42. ax=fig.add\_subplot(projection="3d")
43. ax.set\_xlabel('$x$')
44. ax.set\_ylabel('$y$')
45. ax.set\_zlabel('$z$')
46. ax.plot(r[:,0],r[:,1],r[:,2])
47. ax.scatter([0,1],[0,0],[0,0],s=[300,50],c=['red','blue'])#中心天体
48. plt.show()

极坐标作图程序

1. #极坐标作图
2. complex\_list=r[:,0]+1j\*r[:,1]
3. theta\_list=np.angle(complex\_list)
4. rho\_list=np.abs(complex\_list)
5. fig=plt.figure(figsize=(4,5))
6. ax=fig.add\_subplot(projection="polar")
7. ax.plot(theta\_list,rho\_list)
8. ax.scatter([0,0],[0,1],s=[300,50],c=['red','blue'])
9. fig.suptitle('$v\_0=(%.1f.%.1f,%.1f)$'%(v0[0],v0[1],v0[2]))
10. plt.show()

### 伽尔顿板实验

1. import numpy as np
2. import pandas as pd
3. H,N=50,100000
4. # H,N=int(1e2),int(1e5)#H是下落高度,相当于是纵向钉子的个数,N是下落的小球个数(或者说是实验次数)
5. MatSize=[H,N]
6. MatDeltaX=np.random.choice([-0.5,0.5],MatSize)#构造位移矩阵
7. MatDeltaX=pd.DataFrame(MatDeltaX)
8. XEndList=MatDeltaX.sum()#利用pandas模块可以很方便地算出每列位移之和
9. IndexList=XEndList.unique()#小球在底部的位置坐标,pandas模块可以方便地使用unique函数
10. IndexList.sort()#将位置坐标排序
11. XUniqueList=XEndList.value\_counts()#计算每一位置坐标所对应的小球个数
12. XUniqueList.sort\_index(inplace=True)#将得到的数目列表排序
13. #理论值
14. x0,xN=min(IndexList),max(IndexList)
15. deltax=0.01
16. Analy\_Xlist=np.arange(x0,xN,deltax)
17. scale=np.sqrt(H/4)
18. from scipy.stats import norm
19. Analy\_Ylist=norm.pdf(Analy\_Xlist,scale=scale)\*N
20. #TODO 作图
21. import matplotlib.pyplot as plt
22. fig=plt.figure()
23. ax=fig.add\_subplot()
24. ax.plot(IndexList,XUniqueList,label='Calculated Result')
25. ax.bar(IndexList,XUniqueList)
26. ax.plot(Analy\_Xlist,Analy\_Ylist,label='Normal Distribution')
27. ax.set\_title('N=%d,H=%d'%(N,H))
28. ax.set\_xlabel('coordinate of balls')
29. ax.set\_ylabel('number of balls')
30. ax.legend()
31. plt.show()

### 带电圆环的电场与电势

#### 本文所用的程序

1. import numpy as np
2. Nx,Ny=70,70
3. xlist=np.linspace(-1.5,1.5,Nx)
4. ylist=np.linspace(-1.5,1.5,Ny)
5. Xmat,Ymat=np.meshgrid(xlist,ylist)
6. epsilon0=1
7. k=1/(4\*np.pi\*epsilon0)
8. def standard(x):
9. '''扣点'''
10. index=xlist<(R-x)
11. tmp=max(xlist[index])
12. index=xlist==tmp
13. std=Phimat[index,int(Ny/2)]
14. return std
15. def point(r0,q):
16. '''点电荷在全空间产生的场和势'''
17. Rx=Xmat-r0[0]
18. Ry=Ymat-r0[1]
19. Rnorm\_mat=k\*np.sqrt(Rx\*\*2+Ry\*\*2)#有无更好的求模的方式
20. Phimat=k\*np.divide(q,Rnorm\_mat)
21. Exmat=k\*np.divide(Rx\*q,Rnorm\_mat\*\*3)
22. Eymat=k\*np.divide(Ry\*q,Rnorm\_mat\*\*3)
23. return Phimat,Exmat,Eymat
24. def ring(R=1,Q=10,Ntheta=100):
25. '''圆环'''
26. thetalist=np.linspace(0,2\*np.pi,Ntheta)
27. x0list=np.cos(thetalist)\*R#圆环的中心在原点
28. y0list=np.sin(thetalist)\*R
29. delta\_L=R\*(thetalist[1]-thetalist[0])
30. Lamda=Q/(2\*np.pi\*R)
31. deltaq=Lamda\*delta\_L
32. qlist=deltaq\*np.ones(len(x0list))
33. return x0list,y0list,qlist
34. def dipole():
35. '''电偶极子'''
36. x0list=np.array([-0.5,0.5])
37. y0list=np.array([0,0])
38. qlist=np.array([-10,10])
39. return x0list,y0list,qlist
40. def EandPhi(x0list,y0list,qlist):
41. '''给定带电体在全空间产生的场和势'''
42. Phimat=np.zeros((Nx,Nx))
43. Exmat=np.zeros((Nx,Nx))
44. Eymat=np.zeros((Nx,Nx))
45. for n in range(len(x0list)):
46. r0n=np.array((x0list[n],y0list[n]))
47. Phimatn,Exmatn,Eymatn=point(r0n,qlist[n])
48. Phimat=Phimat+Phimatn
49. Exmat=Exmat+Exmatn
50. Eymat=Eymat+Eymatn
51. return Phimat,Exmat,Eymat
52. R,Q=1,10
53. x0list,y0list,qlist=ring(R=R,Q=Q,Ntheta=1000)#带电圆环
54. # x0list,y0list,qlist=dipole()#电偶极子
55. Phimat,Exmat,Eymat=EandPhi(x0list,y0list,qlist)
56. #清洗数据
57. std=standard(0.05)#匹配带电圆环
58. # std=70#匹配电偶极子
59. index=abs(Phimat)>std
60. Xmat[index],Ymat[index]=0,0
61. Exmat[index],Eymat[index]=0,0
62. #作图
63. import matplotlib.pyplot as plt
64. fig=plt.figure(figsize=(9,4))
65. ax1=fig.add\_subplot(1,2,1)
66. ax1.imshow(Phimat,cmap='inferno')
67. ax1.set\_title('Electric Potential')
68. ax2=fig.add\_subplot(1,2,2)
69. ax2.quiver(xlist,ylist,Exmat,Eymat,color='navy')
70. ax2.set\_title('Electric Field')
71. fig.supylabel('Nx=%d,Ny=%d,Ntheta=%d'%(Nx,Ny,Ntheta))
72. plt.show()

#### 另一种程序实现

1. import numpy as np
2. epsilon0=1
3. k=1/(4\*np.pi\*epsilon0)
4. def point(r0,r,q):
5. '''点电荷在空间某点产生的计算电势
6. 任何维度通用'''
7. dim=len(r0)#获取维数
8. R=r-r0#只有np.array创造的向量才能进行加减运算,直接用[]创建的是列表,会报错
9. norm=np.linalg.norm(R)
10. potential=k\*q/norm
11. E=np.zeros(dim)
12. for n in range(dim):
13. E[n]=k\*q\*R[n]/norm\*\*3
14. return potential,E
15. #圆环在空间某点产生的电势
16. R,Q=1,10
17. Ntheta=1000
18. thetalist=np.linspace(0,2\*np.pi,Ntheta)
19. def ring(r):
20. '''可以改变电荷分布方程来表现任何电荷分布在空间中的势与场'''
21. x0list=np.cos(thetalist)\*R#圆环的中心在原点
22. y0list=np.sin(thetalist)\*R
23. delta\_L=R\*(thetalist[1]-thetalist[0])
24. Lamda=Q/(2\*np.pi\*R)
25. deltaq=Lamda\*delta\_L
26. N=len(x0list)
27. potential=0
28. E=np.zeros(2)
29. for n in range(N):
30. r0n=np.array((x0list[n],y0list[n]))
31. phin,En=point(r0n,r,deltaq)
32. potential=potential+phin
33. E=E+En
34. return potential,E
35. #圆环在全空间产生的电势
36. Nx,Ny=70,70
37. xlist=np.linspace(-1.5,1.5,Nx)
38. ylist=np.linspace(-1.5,1.5,Ny)
39. phimat=np.zeros((Nx,Ny))
40. Exmat=np.zeros((Nx,Ny))
41. Eymat=np.zeros((Nx,Ny))
42. for i in range(Ny):
43. yi=ylist[i]
44. for j in range(Nx):
45. xj=xlist[j]
46. r=np.array((xj,yi))
47. phir,Er=ring(r)
48. phimat[i,j]=phir
49. Exmat[i,j],Eymat[i,j]=Er[0],Er[1]
50. X,Y=np.meshgrid(xlist,ylist)
51. #清洗数据
52. std,\_\_=ring([1.05,0])#使用"\_"忽略此处的返回值
53. index=phimat>std
54. X[index],Y[index]=0,0
55. Exmat[index],Eymat[index]=0,0
56. #作图
57. import matplotlib.pyplot as plt
58. fig=plt.figure(figsize=(9,4))
59. ax1=fig.add\_subplot(1,2,1)
60. ax1.imshow(phimat)
61. # plt.colorbar()
62. ax2=fig.add\_subplot(1,2,2)
63. ax2.quiver(X,Y,Exmat,Eymat,color='navy')
64. plt.show()

### 波动方程

#### 主程序

1. import numpy as np
2. #初始化数据
3. a=1#波的移动速度
4. xN,tN=10,10
5. x0,t0=0,0
6. deltax=0.1
7. deltat=np.sqrt(deltax\*\*2/a\*\*2)#这样alpha始终为1,不会发生指数爆炸,a可取任意值
8. xlist=np.arange(x0,xN,deltax)
9. tlist=np.arange(t0,tN,deltat)
10. Nx,Nt=len(xlist),len(tlist)
11. #设置参数
12. vdata=deltax/deltat
13. alpha=a\*\*2/vdata\*\*2
14. beta=2\*(1-alpha)
15. Lamda=np.zeros((Nx-2,Nx-2))
16. def Boundary(t):
17. '''边界条件函数'''
18. f=0
19. g=0
20. return f,g
21. def Initial(x):
22. '''初始条件函数'''
23. fun=lambda x:np.e\*\*(-x\*\*2)\*10
24. w=fun(x-5)
25. h=fun(x-5+a\*deltat)#第二个时刻的加减a\*deltat决定向左或向右移动
26. return w,h
27. #创建条纹矩阵Lamda
28. for i in range(Nx-3):
29. Lamda[i,i],Lamda[i,i+1],Lamda[i+1,i]=beta,alpha,alpha
30. Lamda[Nx-3,Nx-3]=beta
31. #设置Omega矩阵
32. Omegamat=np.zeros((Nx-2,Nt))
33. Omegamat[0,:],Omegamat[-1,:]=Boundary(tlist)
34. #计算主函数U矩阵
35. Umat=np.zeros((Nx-2,Nt))
36. Umat[:,0],Umat[:,1]=Initial(xlist[1:Nx-1])
37. for j in range(2,Nt):
38. Umat[:,j]=Lamda@Umat[:,j-1]+alpha\*Omegamat[:,j-1]-Umat[:,j-2]
39. import matplotlib.pyplot as plt
40. interval=4
41. for i in range(0,Nt//interval):
42. plt.plot(xlist[1:Nx-1],Umat[:,interval\*i]-i\*5)#每隔interval个点画一次
43. plt.title('a=%.2f'%a)
44. plt.show()

#### 三维作图程序

1. X,T=np.meshgrid(xlist[1:Nx-1],tlist)
2. from matplotlib import cm
3. fig=plt.figure()
4. ax=fig.add\_subplot(1,1,1,projection='3d')
5. surf=ax.plot\_surface(X.T,T.T,Umat,cmap=cm.coolwarm)
6. fig.colorbar(surf,fraction=0.04,pad=0.1,shrink=0.7)
7. ax.set\_title('a=%.2f'%a)
8. ax.set\_xlabel('$x$')
9. ax.set\_ylabel('$t$')
10. ax.set\_zlabel('$u$')
11. plt.show()

### 传热方程

#### 主程序

1. import numpy as np
2. #初始化数据
3. a=1
4. xN,tN=10,1
5. x0,t0=0,0
6. deltax=0.1
7. alpha\_target=0.4
8. deltat=alpha\_target\*(deltax/a)\*\*2#这样alpha为0.5,beta=-1不会发生指数爆炸,a可取任意值
9. xlist=np.arange(x0,xN,deltax)
10. tlist=np.arange(t0,tN,deltat)
11. Nx,Nt=len(xlist),len(tlist)
12. #设置参数与定义函数
13. gamma=deltax\*\*2/deltat
14. alpha=a\*\*2/gamma
15. beta=1-2\*alpha
16. Lamda=np.zeros((Nx-2,Nx-2))
17. def Boundary(t):
18. '''边界条件函数'''
19. f=0
20. g=0
21. return f,g
22. def Initial(x):
23. '''初始条件函数'''
24. fun=lambda x:x\*(xN-x)/xN\*\*2
25. w=fun(x)
26. return w
27. #创建条纹矩阵Lamda
28. for i in range(Nx-3):
29. Lamda[i,i],Lamda[i,i+1],Lamda[i+1,i]=beta,alpha,alpha
30. Lamda[Nx-3,Nx-3]=beta
31. #设置Omega矩阵
32. Omegamat=np.zeros((Nx-2,Nt))
33. Omegamat[0,:],Omegamat[-1,:]=Boundary(tlist)
34. #计算主函数U矩阵
35. Umat=np.zeros((Nx-2,Nt))
36. Umat[:,0]=Initial(xlist[1:Nx-1])
37. for j in range(1,Nt):
38. Umat[:,j]=Lamda@Umat[:,j-1]+alpha\*Omegamat[:,j-1]
39. index=Umat[:,j]<0
40. if np.sum(Umat[index,j])<0:
41. #检验Umat矩阵中是否存在负数,若存在就立即停止计算
42. break

#### 理论值的计算

1. from math import pi
2. K=50#取级数解析解的前50项
3. def analy\_u(x,t):
4. '''解析解表达式'''#需要用np.sin才能实现对矩阵运算
5. sum=np.zeros((Nx-2,Nt))
6. for k in range(K):
7. sum=sum+8/(pi\*(2\*k+1))\*\*3\*np.e\*\*(-((2\*k+1)\*pi\*a/xN)\*\*2\*t)\*np.sin((2\*k+1)\*pi\*x/xN)
8. return sum
9. T,X=np.meshgrid(tlist,xlist[1:Nx-1])
10. analy\_Umat=analy\_u(X,T)

#### 二维作图程序

1. width=3
2. tj=5
3. calcu\_result=Umat[:,tj]
4. analy\_result=analy\_Umat[:,tj]
5. calcu\_result=np.tile(calcu\_result,(width,1))
6. analy\_result=np.tile(analy\_result,(width,1))
7. import matplotlib.pyplot as plt
8. fig=plt.figure()
9. ax1=fig.add\_subplot(2,1,1)
10. imshow\_calcu=ax1.imshow(calcu\_result)
11. fig.colorbar(imshow\_calcu,orientation='horizontal',label='The Colorbar of Calculated Result')
12. ax1.set\_title('Calculated Result')
13. ax2=fig.add\_subplot(2,1,2)
14. imshow\_analy=ax2.imshow(analy\_result)
15. fig.colorbar(imshow\_analy,orientation='horizontal',label='The Colorbar of Analytic Result')
16. ax2.set\_title('Analytic Result')
17. plt.show()

#### 三维作图

1. from matplotlib import cm
2. fig=plt.figure(figsize=(12,4))
3. fig.suptitle('k=%.2f'%(a))
4. ax1=fig.add\_subplot(1,2,1,projection='3d')
5. fig.tight\_layout()#子图见紧密排列
6. calcu\_surf=ax1.plot\_surface(X,T,Umat,cmap=cm.coolwarm)
7. fig.colorbar(calcu\_surf,fraction=0.04,pad=0.1,shrink=0.7)
8. ax1.set\_title('Calculated Result')
9. ax1.set\_xlabel('$x$')
10. ax1.set\_ylabel('$t$')
11. ax1.set\_zlabel('$u$')
12. ax2=fig.add\_subplot(1,2,2,projection='3d')
13. analy\_surf=ax2.plot\_surface(X,T,analy\_Umat,cmap=cm.coolwarm)
14. fig.colorbar(analy\_surf,fraction=0.04,pad=0.1,shrink=0.7)
15. ax2.set\_xlabel('$x$')
16. ax2.set\_ylabel('$t$')
17. ax2.set\_zlabel('$u$')
18. ax2.set\_title('Analytic Result')
19. plt.tight\_layout()
20. plt.show()

### 亥姆霍兹方程之蛮力法求解

#### 主程序

1. import numpy as np
2. from math import cos,sin,pi,sqrt
3. xN,Nx=1,90
4. yN,Ny=1,90
5. zN,Nz=1,80
6. xlist=np.linspace(0,xN,Nx)
7. deltax=xlist[1]-xlist[0]
8. # k=sqrt(3)/deltax
9. m,n,p=1,1,1#谐振腔的模式
10. k=pi\*m/xN
11. # k=0
12. def Boundary(y,z):
13. '''边界条件函数
14. 其中f(y,z)是x=0处的边界条件
15. g(y,z)是x=xN处的边界条件'''
16. f=lambda y,z:cos(n\*pi\*y/yN)\*sin(p\*pi\*z/zN)
17. g=lambda y,z:(-1)\*\*m\*cos(pi\*y/yN)\*sin(pi\*z/zN)
18. return f(y,z),g(y,z)
19. #构造Lamda矩阵
20. Lambda=np.zeros((Nx-2,Nx-2))
21. alpha=(k\*deltax)\*\*2-2
22. for i in range(Nx-3):
23. Lambda[i,i]=alpha
24. Lambda[i+1,i],Lambda[i,i+1]=1,1
25. Lambda[Nx-3,Nx-3]=alpha
26. #填充U矩阵
27. ylist=np.linspace(0,yN,Ny)
28. zlist=np.linspace(0,zN,Nz)
29. Omega=np.zeros(Nx-2)
30. Umat=np.zeros((Nx-2,Ny,Nz))
31. for j in range(Ny):
32. for k in range(Nz):
33. Omega[0],Omega[Nx-3]=Boundary(ylist[j],zlist[k])
34. Umat[:,j,k]=np.linalg.solve(Lambda,-Omega)

#### 作图程序

1. import matplotlib.pyplot as plt
2. index=10
3. fig=plt.figure(figsize=(11,3))
4. ax1=fig.add\_subplot(1,3,1)
5. ax1.imshow(Umat[:,:,index])
6. ax1.set\_xlabel('$x$')
7. ax1.set\_ylabel('$y$')
8. ax2=fig.add\_subplot(1,3,2)
9. ax2.imshow(Umat[:,index,:])
10. ax2.set\_xlabel('$x$')
11. ax2.set\_ylabel('$z$')
12. ax3=fig.add\_subplot(1,3,3)
13. ax3.imshow(Umat[index,:,:])
14. ax3.set\_xlabel('$y$')
15. ax3.set\_ylabel('$z$')
16. plt.show()

### 亥姆霍兹方程之超松弛法求解

#### 函数文件Helmholtz.py

1. import numpy as np
2. class cell:
3. '''创建元胞数据类型'''
4. def \_\_init\_\_(self,value=0):
5. self.value=value
6. self.is\_iterate=True
7. self.neighbors=[]
8. def set\_value(self,value=0):
9. self.value=value
10. def set\_is\_iterate(self,bool=True):
11. self.is\_iterate=bool
12. def set\_neighbors(self,NeighborsList):
13. self.neighbors=NeighborsList
14. def iterate(self,x,y,z,deltaq,beta=1,f=0,g=0):
15. if self.is\_iterate:
16. sum=0
17. for neighbor in self.neighbors:
18. sum+=neighbor.value
19. # new\_value=(sum-deltaq\*\*2\*f)/(len(self.neighbors)-deltaq\*\*2\*g)#f,g传值
20. new\_value=(sum-deltaq\*\*2\*f(x,y,z))/(len(self.neighbors)-deltaq\*\*2\*g(x,y,z))#f,g传函数
21. delta\_value=new\_value-self.value
22. value=self.value+beta\*delta\_value
23. self.set\_value(value)
24. def Initial(xlist,ylist,zlist,set\_Boundary):
25. '''初始化元胞(添加邻近元胞列表)并设置边界条件'''
26. Nx,Ny,Nz=len(xlist),len(ylist),len(zlist)
27. #创建元胞网格点
28. UMatCell=cellList=[[[cell() for irow in range(Nx)]
29. for icol in range(Ny)]
30. for ihei in range(Nz)]#只能使用列表推导式
31. UMatCell=np.array(UMatCell)#转换为ndarray数据格式方便用UMatCell[i,j,k]形式的索引
32. for i in range(Nx):
33. for j in range(Ny):
34. for k in range(Nz):
35. set\_Boundary(xlist[i],ylist[j],zlist[k],UMatCell[i,j,k])#设置边界条件
36. # 暂时采用取余的办法规避索引溢出的问题
37. NeighborsList=[UMatCell[(i-1)%Nx,j,k],UMatCell[(i+1)%Nx,j,k],
38. UMatCell[i,(j-1)%Ny,k],UMatCell[i,(j+1)%Ny,k],
39. UMatCell[i,j,(k-1)%Nz],UMatCell[i,j,(k+1)%Nz],]
40. UMatCell[i,j,k].set\_neighbors(NeighborsList)
41. return UMatCell
42. def Iteration(xlist,ylist,zlist,tN,UMatCell,f,g,beta=1):
43. '''迭代'''
44. deltaq=xlist[1]-xlist[0]#x,y,z三个方向的离散化程度相同
45. Nx,Ny,Nz=len(xlist),len(ylist),len(zlist)
46. shape=np.shape(UMatCell)
47. UMatValue=np.zeros(shape)#不能使用empty\_like或zeros\_like
48. UMatValue\_tmp=np.zeros(shape)
49. for t in np.arange(tN):
50. if not (t+1)%10:
51. deltaU=abs((np.sum(UMatValue)-np.sum(UMatValue\_tmp))/Nx/Ny/Nz)
52. print('deltaU=%e'%deltaU)
53. for i in range(Nx):
54. for j in range(Ny):
55. for k in range(Nz):
56. x,y,z=xlist[i],ylist[j],zlist[k]
57. UMatValue\_tmp[i,j,k]=UMatValue[i,j,k]
58. # UMatCell[i,j,k].iterate(x,y,z,deltaq,beta,f(x,y,z),g(x,y,z))#f,g传值
59. UMatCell[i,j,k].iterate(x,y,z,deltaq,beta,f,g)#f,g传函数
60. UMatValue[i,j,k]=UMatCell[i,j,k].value
61. return UMatValue

#### 主程序

1. import numpy as np
2. from Helmholtz import Initial,Iteration
3. def f1(x,y,z):
4. '''放在(0,0,0)处的点电荷'''
5. if x==xlist[Nx//2] and y==ylist[Ny//2] and z==zlist[Nz//2]:
6. Q,epsilon=1,1
7. rho=Q/deltaq\*\*3
8. f=-rho/epsilon#注意q!=rho,应除以deltaq的三次方,注意负号
9. return f
10. else:
11. return 0
12. def f2(x,y,z):
13. '''电偶极子'''
14. condition1=x==xlist[Nx//2] and y==ylist[Ny//2+1] and z==zlist[Nz//2]
15. condition2=x==xlist[Ny//2] and y==ylist[Ny//2-1] and z==zlist[Nz//2]
16. if condition1 or condition2:
17. Q,epsilon=1,1
18. rho=Q/deltaq\*\*3
19. if condition1:
20. f=-rho/epsilon
21. else:
22. f=rho/epsilon
23. return f
24. else:
25. return 0
26. def f3(x,y,z):
27. '''中心在(0,0,0)的带电正方体
28. 特别地,当a=2时退化为点电荷'''
29. a=6#正方体边长,必须为整数
30. condition1=x>xlist[Nx//2-a//2] and x<xlist[Nx//2+a//2]
31. condition2=y>ylist[Ny//2-a//2] and y<ylist[Ny//2+a//2]
32. condition3=z>zlist[Nz//2-a//2] and z<zlist[Nz//2+a//2]
33. if condition1 and condition2 and condition3:
34. Q,epsilon=1,1
35. rho=Q/(a\*deltaq)\*\*3
36. f=-rho/epsilon
37. return f
38. else:
39. return 0
40. def g(x,y,z):
41. return 0
42. def set\_Boundary(x,y,z,cell):#cell参数不能改动
43. '''边界条件函数'''
44. R=np.sqrt(x\*\*2+y\*\*2+z\*\*2)
45. if R>xN:#xN在主程序中给出
46. cell.set\_value(0)
47. cell.set\_is\_iterate(False)
48. return
49. def analy():
50. Q=1
51. epsilon=1
52. Coefficient=1/4/np.pi/epsilon
53. result=Q\*Coefficient\*(np.divide(1,xlist[Nx//2:])-1/xN)
54. return result
55. #初始化参数
56. tN=50#迭代次数
57. x0,xN=-1,1
58. y0,yN=-1,1
59. z0,zN=-1,1
60. deltaq=0.05
61. #创建坐标值网格点,方便找到特定位置的索引从而设置边界条件
62. xlist=np.arange(x0,xN,deltaq)
63. ylist=np.arange(y0,yN,deltaq)
64. zlist=np.arange(z0,zN,deltaq)
65. Nx,Ny,Nz=len(xlist),len(ylist),len(zlist)
66. #对数据进行操作
67. beta=2/(1+np.pi/len(xlist))#beta=1.8#若β不等于1那么图像就不对称了，特别当β较大时
68. UMatCell=Initial(xlist=xlist,ylist=ylist,zlist=zlist,set\_Boundary=set\_Boundary)#元胞堆叠初始化
69. UMatValue=Iteration(xlist=xlist,ylist=ylist,zlist=zlist,tN=tN,UMatCell=UMatCell,f=f1,g=g,beta=beta)#迭代元胞

#### 点电荷的三视图作图程序

1. import matplotlib.pyplot as plt
2. from matplotlib import colors
3. norm=colors.Normalize(vmin=0,vmax=0.1)
4. fig,axes=plt.subplots(1,3,figsize=(9,3))
5. fig.suptitle('the Potential Distribution of the Point Charges')
6. fig.supxlabel('Iterations:%d'%tN,y=-0.08)
7. index\_x,index\_y,index\_z=Nx//2,Ny//2+10,Nz//2+15
8. draw0=axes[0].imshow(UMatValue[index\_x,:,:],norm=norm)
9. axes[0].contour(UMatValue[index\_x,:,:],5,colors='darkslategray')
10. axes[0].set\_xlabel('x=%f'%xlist[index\_x])
11. draw1=axes[1].imshow(UMatValue[:,index\_y,:],norm=norm)
12. axes[1].set\_xlabel('y=%f'%ylist[index\_y])
13. axes[1].contour(UMatValue[:,index\_y,:],5,colors='darkslategray')
14. draw2=axes[2].imshow(UMatValue[:,:,index\_z],norm=norm)
15. axes[2].contour(UMatValue[:,:,index\_z],5,colors='darkslategray')
16. axes[2].set\_xlabel('z=%f'%zlist[index\_z])
17. fig.colorbar(draw2,ax=[axes[0],axes[1],axes[2]])#共用一个图标
18. plt.show()

#### 点电荷作图分析程序

1. import matplotlib.pyplot as plt
2. fig,axes=plt.subplots(1,2,figsize=(10,4))
3. fig.suptitle('the Point Charges')
4. index=Nz//2
5. nep=axes[0].imshow(UMatValue[:,:,index])
6. axes[0].contour(UMatValue[:,:,index],5,colors='darkslategray')
7. axes[0].set\_title('the Potential Distribution')
8. axes[0].set\_xlabel('z=%f,iterations:%d'%(zlist[index],tN))
9. fig.colorbar(nep,ax=axes[0])
10. analy\_result=analy()
11. tmp=1
12. Range=xlist[Nx//2+tmp:]
13. axes[1].plot(Range,UMatValue[Nx//2+tmp:,Ny//2,Nz//2],label='Calculated Result')
14. axes[1].scatter(Range,analy\_result[tmp:],label='Analytic Result',s=13,c='g')
15. axes[1].set\_ylabel('$\phi$')
16. axes[1].set\_xlabel('$r$')
17. axes[1].set\_title('$r-\phi$')
18. plt.legend()
19. plt.show()

#### 电偶极子作图分析程序

1. import matplotlib.pyplot as plt
2. fig,axes=plt.subplots(1,2,figsize=(10,4))
3. fig.suptitle('the Dipole Charges')
4. index=Nz//2
5. nep=axes[0].imshow(UMatValue[:,:,index])
6. axes[0].set\_title('the Potential Distribution')
7. axes[0].set\_xlabel('z=%f,iterations:%d'%(zlist[index],tN))
8. axes[0].contour(UMatValue[:,:,index],5,colors='darkslategray')
9. fig.colorbar(nep,ax=axes[0])
10. tmp=2#从零处的值很大,需要截断
11. Range=xlist[Nx//2+tmp:]
12. axes[1].plot(Range,UMatValue[Nx//2,Ny//2+tmp:,Nz//2])#Uy方向才具有较好的对称性
13. axes[1].set\_ylabel('$\phi$')
14. axes[1].set\_xlabel('$r$')
15. axes[1].set\_title('$r-\phi$')
16. plt.show()

#### 带电正方体作图分析程序

1. import matplotlib.pyplot as plt
2. fig,axes=plt.subplots(1,2,figsize=(10,4))
3. fig.suptitle('the Cube Charges')
4. index=Nz//2
5. nep=axes[0].imshow(UMatValue[:,:,index])
6. axes[0].contour(UMatValue[:,:,index],5,colors='darkslategray')
7. axes[0].set\_title('the Potential Distribution')
8. axes[0].set\_xlabel('z=%f,iterations:%d'%(zlist[index],tN))
9. fig.colorbar(nep,ax=axes[0])
10. tmp=1
11. Range=xlist[Nx//2+tmp:]
12. axes[1].plot(Range,UMatValue[Nx//2+tmp:,Ny//2,Nz//2])
13. axes[1].set\_ylabel('$\phi$')
14. axes[1].set\_xlabel('$r$')
15. axes[1].set\_title('$r-\phi$')
16. plt.show()

## 参考

GitHub: <https://github.com/igfasouza/pi-day-jupyter-notebook/blob/master/PI.ipynb>

CSDN

Python官方文档