偏微分方程PDE求解专题总结研究

2052972 张斯然

目录

1 总述

偏微分方程的求解是物理学中非常重要的问题，各种各样的物理问题经过理论分析化简后，亦或是在真实应用场景中，往往就是要解决偏微分方程的求解。经典的牛顿力学方程，分析力学方程，麦克斯韦方程组以及薛定谔方程都可以归属于偏微分方程（PDE）。在数学、物理及工程技术中应用最广泛的，是二阶偏微分方程，甚至已经习惯上把这些方程称为数学物理方程。

解决一个实际物理问题，需要根据物理定律建立数学模型，导出泛定方程，再根据由初值条件、边界条件等定解条件来进行求解。当然，在求解偏微分方程的解，必须研究三个问题，解的存在性，唯一性，以及稳定性。

本专题研究主要对目前本学期上课及自己了解的偏微分方程的各种解法进行整理，主要整理传统解析法，常见的数值计算方法，以及深度学习方法(重点放在后两种)来对偏微分方程进行求解计算，并在其中穿插一些作图的例子，代码全部由本人使用Python进行编写

2 传统解析方法

本节主要介绍较为经典传统的数学解析解法，但是适用范围很有限，大部分的结果很难给出数值表达式，这里仅做概述，介绍主要思想。本节主要包含分离变量法、积分变换法 这两种常用的方法。使用这种方法对于推导有一定要求，虽然方法不同，但两种方法目标是将偏微分方程降维为常微分方程，从而进行求解，在第三章选择补充介绍数值求解常微分方程方法，作为解析方法的补充

2.1 分离变量法

核心：偏微分方程化常微分方程

基本思路：先求出分离变量形式的且满足边界条件的特解，然后根据叠加原理给出解的线性叠加，由定解条件确定待定系数

关键步骤：求解本征值问题。

依照不同的坐标系可以进行不同的坐标系的分离变量，其核心为拉普拉斯算符在不同坐标系的表达

1. 直角坐标系

$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$

2. 柱坐标系

$\nabla^2 = \frac{1}{\rho}\frac{\partial}{\partial \rho}(\rho\frac{\partial}{\partial \rho}) + \frac{1}{\rho^2}\frac{\partial^2}{\partial \varphi^2} + \frac{\partial^2}{\partial z^2}$

3. 球坐标系

一般球对称性较好的可以拆成径向和角向两个部分

$\nabla^2 = \frac{1}{R^2}\frac{\partial}{\partial R}(R^2\frac{\partial}{\partial R}) + \frac{1}{R^2\sin\theta}\frac{\partial}{\partial \theta}(\sin\theta\frac{\partial}{\partial \theta}) + \frac{1}{R^2\sin^2\theta}\frac{\partial^2}{\partial \varphi^2}$

2.1.1 Example1：氢原子的波函数求解

氢原子的波函数求解是量子力学中少有的可以精确求解的结果，也十分清晰的展示了分离变量法在其中的应用

在球坐标系下氢原子的定态薛定谔方程可以写为：

$-\frac{\hbar^2}{2m}[\frac{1}{r^2}\frac{\partial}{\partial r}(r^2\frac{\partial \psi}{\partial r} + \frac{1}{r^2}\frac{1}{\sin\theta}\frac{\partial}{\partial \theta}(\sin\theta\frac{\partial \psi}{\partial \theta}) + \frac{1}{r^2}\frac{1}{\sin^2\theta}\frac{\partial^2 \psi}{\partial \phi^2})] + V\psi = 0$

由于V球对称性，考虑使用分离变量法使得$\psi(r,\theta,\phi) = R(r)Y(\theta,\phi)$

代入原方程进行化简，化简形式为{}+{}=0，前一项仅与r有关，后一项仅与θ,ϕ有关，因此，每一项必须为一个常数

将原方程拆解为径向方程与角向方程

径向方程：$-\frac{\hbar^2}{2m}\frac{\mathrm{d}^2u}{\mathrm{d}r^2} + [-\frac{e^2}{4\pi\varepsilon_0}\frac{1}{r} + \frac{\hbar^2}{2m}\frac{l(l+1)}{r^2}]u = uE$(u=rR(r))

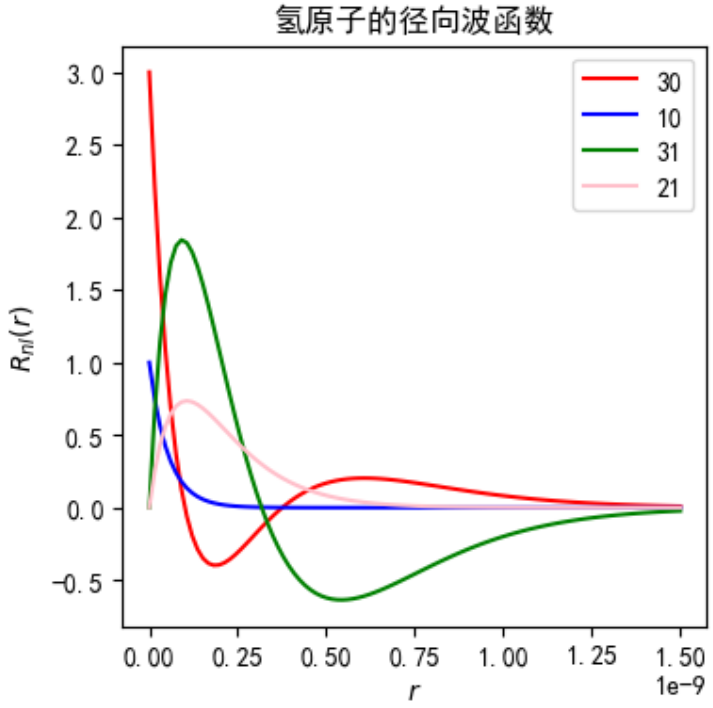
角向方程：$\sin\theta\frac{\partial}{\partial \theta}(\sin\theta\frac{\partial Y}{\partial \theta}) + \frac{\partial^2}{\partial \phi^2} = -l(l+1)\sin^2\theta Y$

角向方程可以使用类似的方法进行变量的分离

最终可以得到氢原子归一化波函数$\psi_{nlm} = \sqrt{(\frac{2}{na})^3\frac{(n-l-1)!}{2n[(n+l)!]^3}}e^{-r/na}(\frac{2r}{na})^l[L_{n-l-1}^{2l+1}(2r/na)]Y_l^m(\theta,\phi)$

$Y_l^m(\theta,\phi) = \varepsilon\sqrt{\frac{(2l+1)(l-|m|)!}{4\pi(l+|m|)!}}e^{im\phi}P_l^m(\cos\theta)$

```python
#这里画出氢原子的径向波函数进行展示
import matplotlib.pyplot as plt
import numpy as np
import scipy.special as special
def Rj(r,n,l):
    a0=0.529*10**(-10)
    p=r/a0/n
    v=special.assoc_laguerre(2*p,n-l-1,2*l+1)
    R=(2*p)**(l)*np.exp(-p)*v
    return R
fig=plt.figure(figsize=(4,4))
Rlist1=np.linspace(0.0000001,1.5,100)
Rlist=[i*10**(-9) for i in Rlist1]
ylist1=[]
ylist2=[]
ylist3=[]
ylist4=[]
for i in range(0,len(Rlist)):
    temp=Rj(Rlist[i],3,0)
    ylist1.append(temp)
    temp1=Rj(Rlist[i],1,0)
    ylist2.append(temp1)
    temp2=Rj(Rlist[i],3,1)
    ylist3.append(temp2)
    temp3=Rj(Rlist[i],2,1)
    ylist4.append(temp3)
plt.plot(Rlist,ylist1,c='r',label='30')
plt.plot(Rlist,ylist2,c='b',label='10')
plt.plot(Rlist,ylist3,c='g',label='31')
plt.plot(Rlist,ylist4,c='pink',label='21')
plt.legend()
plt.rcParams['font.sans-serif']=['Simhei']
plt.rcParams['axes.unicode_minus']=False
plt.title("氢原子的径向波函数")
plt.ylabel('$R_{nl}(r)$')
plt.xlabel('$r$')
```



2.1.2小结

分离变量法是最常用的求解常见偏微分方程并进行简化的方法

对于非齐次方程及非齐次边界条件的定解问题，一般都是先把非齐次边界条件的定解问题转为齐次，再用本征函数展开法进行求解。

2.2积分变换法

积分变换法也是目前求解数学物理方程中的一种重要方法，适用于求解无界区域以及半无界区域。核心思想是通过积分变换，减少自变量的个数，直到化为常微分方程。

常见的积分变换式有两种，一种是傅里叶变换，另外一种是拉格朗日变换。

傅里叶变换：

$F[f(x)]=\tilde{f}(k)=\int_{-\infty}^{\infty}f(x)e^{-ikx}dx$ 傅里叶正变换

$F^{-1}[\tilde{f}(k)]=f(x)=\frac{1}{2\pi}int_{-\infty}^{\infty}\tilde{f}(k)e^{ikx}dk$ 傅里叶逆变换

傅里叶变换要求：进行变换的函数在无穷区间内有定义，在任一有限区间满足狄利克雷条件，以及$\int_{-\infty}^{\infty}|f(t)|dt$存在

但是这样限制比较多，因此引入拉氏变换

$L[f(p)]=\bar{f}(p)=\int_{0}^{\infty}f(t)e^{-pt}dt$

$L^{-1}[\bar{f}(p)]=f(t)=\frac{1}{2\pi i}\int_{\sigma-i\infty}^{\sigma+i\infty}\bar{f}(p)e^{pt}dp$

在具体问题的求解过程中，常常使用傅里叶变换和拉式变换的性质来进行简化。整体思路通过变换将原函数的初值问题转为像函数，通过性质化简方程，求出像函数，最后通过逆变换求得原函数

在这里以三维泊松方程作为例子，同时熟悉画图的一些技巧。

2.2.1Example2：三维泊松方程定解问题

$\nabla^2 u=-\frac{1}{\varepsilon}\rho_f$

进行傅里叶变换

$F[\sum_{i=1}^{3}\frac{\partial^2 u(\mathbf{x})}{\partial x_i^2}]=-F^{-1}[-\frac{1}{\varepsilon}\rho_f]$

由傅里叶变换微分定理：（$F[f^{(n)}(x)]=(ik)^n F[f(x)]$）

有：$\sum_{i=1}^{3}(ik_i)^2\tilde{u}(k)=-\frac{1}{\varepsilon}\tilde{\rho}_f(k)$

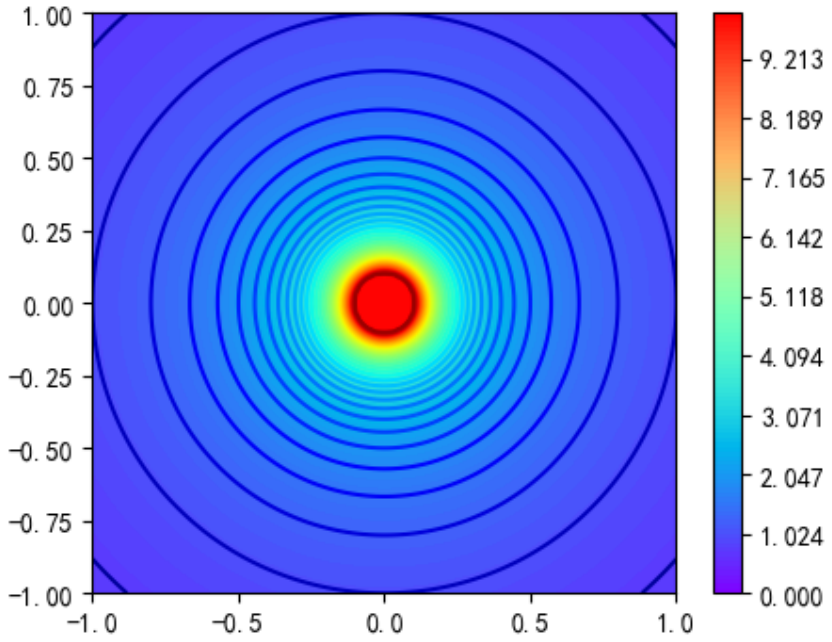
化简得到$\tilde{u}(k)=-\frac{1}{\varepsilon k^2}\tilde{\rho}_f(k)$

之后再进行傅里叶逆变换进行化简：

最终可得：$u(x)=\frac{1}{4\pi\varepsilon}\int_{-\infty}^{\infty}\frac{\rho_f(x')}{|x-x'|}d^3(x')$

下面以单个点电荷的电场分布进行画图展示（2d）将常数忽略掉

```
import numpy as np
import matplotlib.pyplot as plt
x=np.linspace(-1,1,200)
y=np.linspace(-1,1,200)
X,Y=np.meshgrid(x,y)
r=np.sqrt(X**2+Y**2)
q=1
r1=0.1
temp=np.where(r<=r1)
V=q/r
V[temp[0],temp[1]]=q/r1
fig=plt.figure(figsize=(5,4))
height=plt.contour(X,Y,V,40,cmap='jet')#等高线绘制
#plt.clabel(height)
range1=np.linspace(0,10,128)
draw=plt.contourf(X,Y,V,levels=range1,cmap='rainbow')
fig.colorbar(draw)
```



3 常用数值解法以及示例

解析解一般针对方程较为简单，对称性较好的情况，但是对于实际问题，大多数时候很难求得解析解的形式，因此需要使用数值解法进行求解。数值求解精度高，在低维度的情况下速度也比较快，但是随着维数的增多，求解网格划分的增多，数值求解的时间会呈现几何倍数的增加。本节主要总结一下常用的数值方法与求解思路，从常微分方程出发，再到偏微分方程，逐次深入

3.1常微分方程数值解法

偏微分方程的解法很多与常微分方程相类似，因此，从常微分方程开始进行介绍。常微分方程的问题一般可以分为初值问题和边值问题

3.1.1Runge-Kutta法

Runge-Kutta法可由最基础的欧拉方法推导而来，这里不对具体的推导过程进行详述，一般显式Runge-Kutta形式：

$y_{n+1} = y_n + \sum_{i=1}^{r} w_i k_i$

$k_1 = hf(x_n, y_n)$

$K_i = hf(x_n + \alpha_i h, y_n + \sum_{j=1}^{i-1} \beta_{ij} kj$

可以通过对$y(x_{n+1})$泰勒展开求得不同阶-段的Runge-Kutta公式

给出标准的四阶-四段Runge-Kutta公式：

$y_{n+1}=y_n+\frac{1}{6}(k_1+2k_2+2k_3+k_4)$

$k_1=hf(x_n,y_n),k_2=hf(x_n+\frac{1}{2}h,y_n+\frac{1}{2}k_1),k_3=hf(x_n+\frac{1}{2}h,y_n+\frac{1}{2}k_2),k_4=hf(x_n+h,y_n+k_3)$

其具体算法为：给定正整数N，计算$h=\frac{b-a}{N},y(1)=y_0$便历n=1, 2, . . N, 执行求解
k_1,k_2,k_3,k_4,y_{n+1},返回x,y

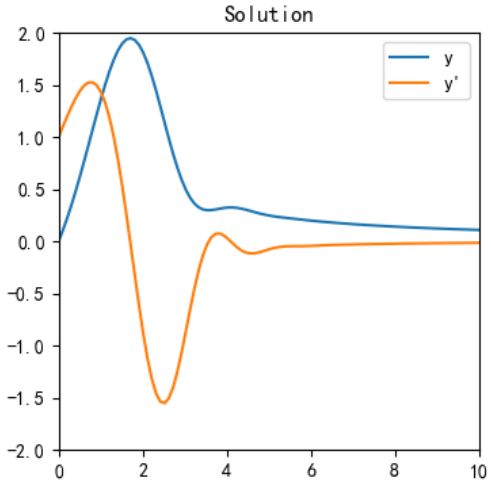
对于高阶常微分方程，将高阶微分方程转化为一阶微分方程组再进行求解

3.1.2Example3: 四阶Runge_Kutta求解高阶常微分方程

使用4阶Runge-Kutta求解一个二阶常微分方程，与一阶常微分方程不同，需要将方程先化为一阶常微分方程组

求解的常微分方程为：$y''=-xy'-x^2y+x+1,y(0)=0,y'(0)=1,0<x<10$

```
import numpy as np
import matplotlib.pyplot as plt
def Runge_kutta4(a,b,N,fun):
    h=(b-a)/N
    y1=np.array([[0],[1]])
    x=np.linspace(a,b,N)#定义域
    Y=np.zeros
    Y=[0*i for i in range(0,N+1)]
    Y[0]=y1
    for i in range(0,len(x)):
        k1=h*fun(x[i],Y[i])
        k2=h*fun(x[i]+0.5*h,Y[i]+0.5*k1)
        k3=h*fun(x[i]+0.5*h,Y[i]+0.5*k2)
        k4=h*fun(x[i]+h,Y[i]+k3)
        Y[i+1]=Y[i]+1/6*(k1+2*k2+2*k3+k4)
    return x,Y
def fun(xn,yn):
    yn1=np.matmul(np.array([[0,1],[-xn**2,-xn]]),yn)+np.array([[0],[xn+1]])
    return yn1
x,Y=Runge_kutta4(0,10,100,fun)
fig=plt.figure(figsize=(4,6))
y=[]
y1=[]
for i in Y:
    y.append(i[0])
    y1.append(i[1])
fig=plt.figure(figsize=(4,4))
y.pop()
y1.pop()
plt.plot(x,y,label='y')
plt.plot(x,y1,label='y\'')
plt.xlim(0,10)
plt.ylim(-2,2)
plt.legend()
plt.title("Solution")
```



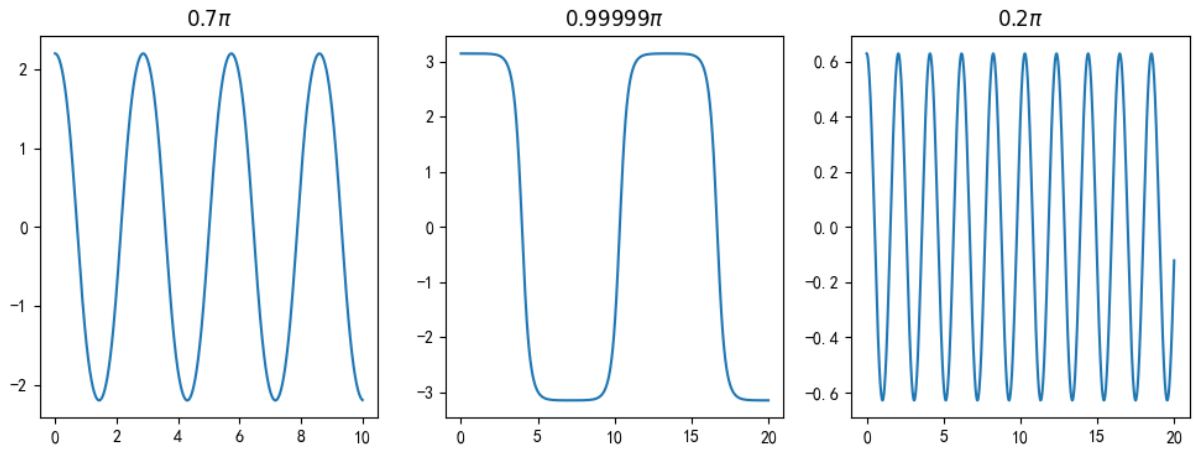
3.1.3 Example 4:scipy库函数求解常微分方程 以单摆为例(二阶非线性)

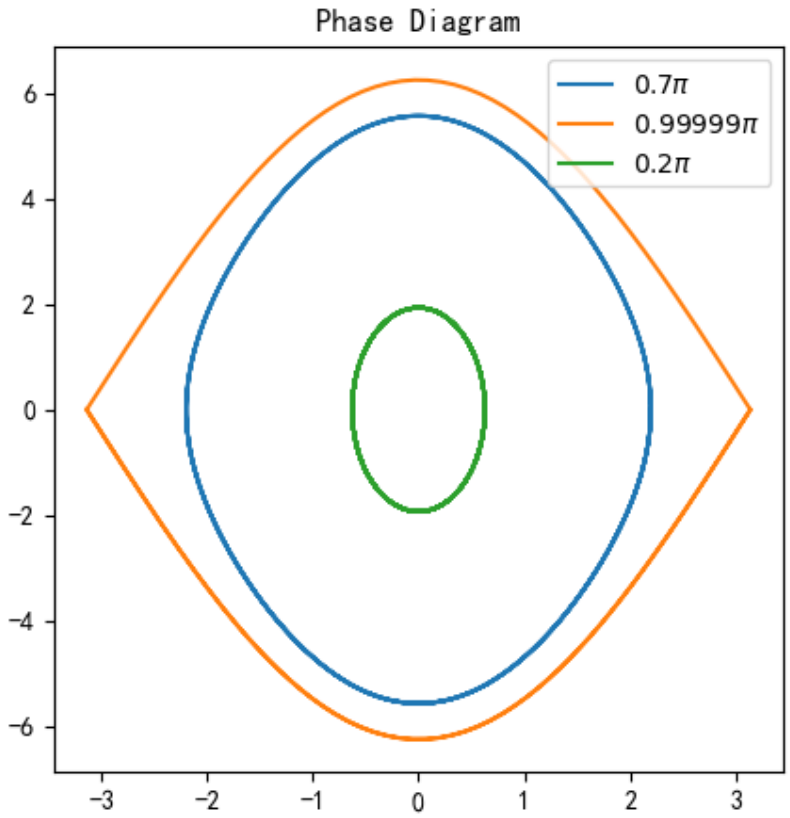
单摆方程可抽象简化为：$\ddot{\theta}+\frac{g}{L}\sin\theta=0$ 通过改变初始设置高度，可以得到不同的运动形式

这里使用scipy的odeint函数进行求解，展示求解结果并画出相图，整理相图的画法。

```
import scipy as sc
from scipy.integrate import odeint,solve_ivp  # 导入 scipy.integrate 模块
import numpy as np
import matplotlib.pyplot as plt
def fun(t,y):
    dy1 = y[1]# 一阶导
    dy2 = -g/l*np.sin(y[0]) # y[0]代表y
    return [dy1,dy2]
def Solve(a,t1):
    t = np.linspace(0,t1,1000)
    y0 = [a, 0] # 初值条件
    y = odeint(fun, y0, t, tfirst=True)
    return t,y
g=9.8
l=1
t,y1=Solve(np.pi*0.7,10)
fig=plt.figure(figsize=(12,4))
plt.subplot(1,3,1)
th1=[]
th11=[]
for i in y1:
    th1.append(i[0])
    th11.append(i[1])
plt.plot(t,th1)
plt.title("$0.7\pi$")
t,y2=Solve(np.pi*0.99999,20)
plt.subplot(1,3,2)
th2=[]
th22=[]
for i in y2:
    th2.append(i[0])
    th22.append(i[1])
plt.plot(t,th2)
plt.title("$0.99999\pi$")
t,y3=Solve(np.pi*0.2,20)
plt.subplot(1,3,3)
th3=[]
th33=[]
```

```
for i in y3:
    th3.append(i[0])
    th33.append(i[1])
plt.plot(t,th3)
plt.title("$0.2\pi$")
fig2=plt.figure(figsize=(5,5))
plt.plot(th1,th11,label='$0.7\pi$')
plt.plot(th2,th22,label='$0.99999\pi$')
plt.plot(th3,th33,label='$0.2\pi$')
plt.title("Phase Diagram")
plt.legend()
```





3.2偏微分方程数值解法

偏微分方程是物理问题中最为常见的方程，本节主要针对二阶偏微分方程进行常见方法的总结分析。
二阶偏微分方程的一般形式为 $Au_{xx}+Bu_{yy}+Cu_{zz}=f(x,y,z,u_x,u_y,u_z)$

根据$\Delta=B^2-4AC$将方程分为三类：$\Delta<0$为椭圆型，$\Delta=0$为抛物线型$\Delta>0$为双曲型

这在matlab中针对不同形式的方程进行求解。

本节主要使用python进行偏微分方程的求解研究

3.2.1有限差分法

这里给出有限差分法,这里以中心差分为例：

n阶中心差分公式：$f_i^{(n)}=\frac{f_{i+1}^{n-1}-f_{i-1}^{n-1}}{2h}$

例如二阶中心差分公式：$f_i''=\frac{f_{i+1}'-f_{i-1}'}{2h}=\frac{f_{i+2}-2f_i+f_{i-2}}{4h^2}$

对于偏微分方程:以稳定场方程为例

$\nabla^2u=u_{xx}+u_{yy}=\frac{u_{i,j}-2u_{i-1,j}+u_{i-2,j}}{h^2}+\frac{u_{i,j}-2u_{i,j-1}+u_{i,j-2}}{h^2}$

之后根据定解条件建立方程组

3.2.2Example 5 有限差分法解点电荷电势分布

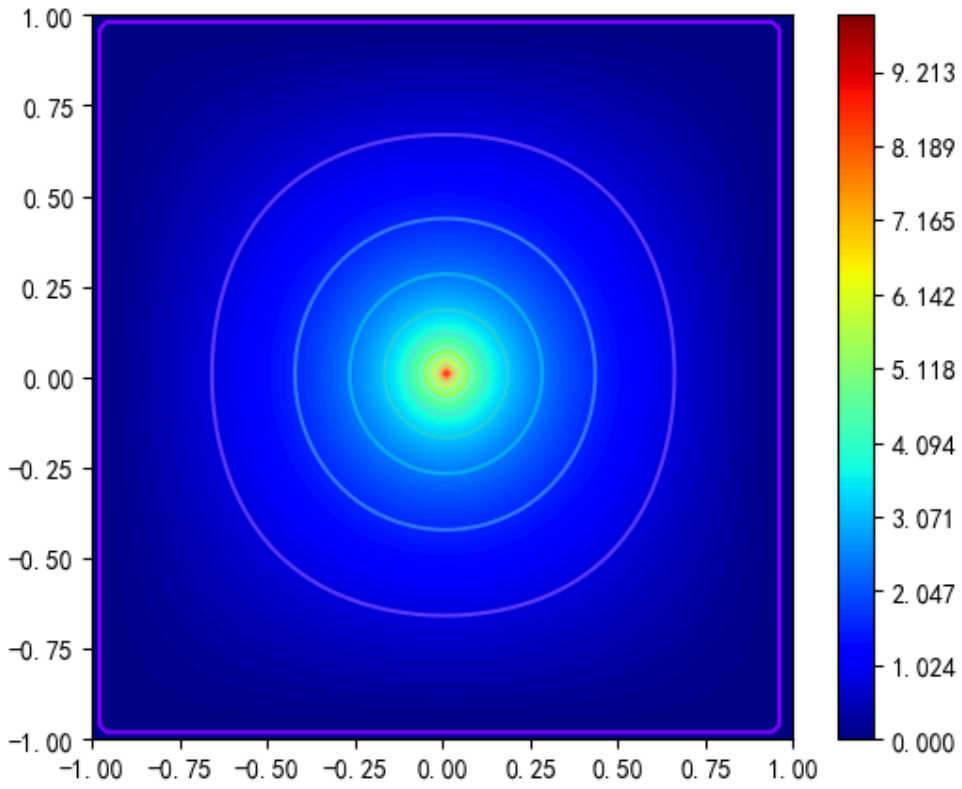
求解一个接地正方形边框内电势分布，正中心有一个点电荷。
使用松弛迭代法减少迭代次数

```
import numpy as np
import matplotlib.pyplot as plt
N1=102#划分网格数
N2=102#划分网格数
V1=np.zeros((N1,N2))
V1[int(N1/2)][int(N2/2)]=10#放置一个点电荷
V2=V1
itermax=500
maxj=1
iter1=0
#计算松弛因子
temp=(np.cos(np.pi/N1)+np.cos(np.pi/N2))/2
w=2/(1+np.sqrt(1-temp**2))
while maxj>1e-6:
    iter1=iter1+1
    if iter1==itermax:
        break
    maxj=0
    for i in range(2,N1-2):
```

```
        for j in range(2,N2-2):
            if i==int(N1/2) and j==int(N2/2):
                continue
            V2[i,j]=V1[i,j]
            if j+1==100:
                j=98
            V1[i,j] = w/4*(V1[i-1,j]+V1[i+1,j] + V1[i,j-1] + V1[i,j+1])+(1-w) * V1[i,
j]
            if abs(V2[i,j]-V1[i,j]>maxj):
                maxj=abs(V2[i,j]-V1[i,j])
            else:
                maxj=1
print(iter1)
```

```
500
```

```
x=np.linspace(-1,1,102)
y=np.linspace(-1,1,102)
X,Y=np.meshgrid(x,y)
fig=plt.figure(figsize=(6,5))
range1=np.linspace(0,10,128)
height=plt.contour(X,Y,V1,15,cmap='rainbow')#等高线绘制
draw=plt.contourf(X,Y,V1,levels=range1,cmap='jet')
fig.colorbar(draw)
```



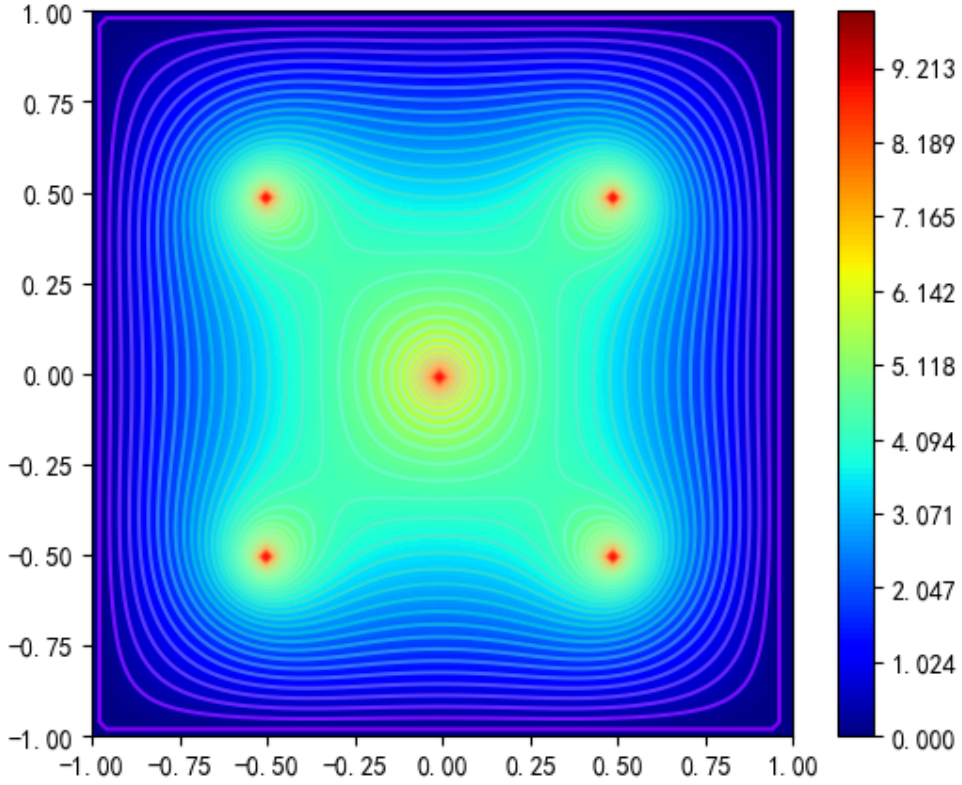
尝试放置多个点电荷

```

```

```
import numpy as np
import matplotlib.pyplot as plt
N1=102#划分网格数
N2=102#划分网格数
V1=np.zeros((N1,N2))
V1[25][25]=10#放置一个点电荷
V1[25][75]=10#放置一个点电荷
V1[75][25]=10#放置一个点电荷
V1[75][75]=10#放置一个点电荷
V1[50][50]=10#放置一个点电荷
V2=V1
itermax=500
maxj=1
iter1=0
#计算松弛因子
temp=(np.cos(np.pi/N1)+np.cos(np.pi/N2))/2
w=2/(1+np.sqrt(1-temp**2))
while maxj>1e-6:
    iter1=iter1+1
    if iter1==itermax:
        break
    maxj=0
    for i in range(2,N1-2):
        for j in range(2,N2-2):
            if i==25 and j==25:
                continue
            elif i==25 and j==75:
                continue
            elif i==75 and j==25:
                continue
            elif i==75 and j==75:
                continue
            elif i==50 and j==50:
                continue
            V2[i,j]=V1[i,j]
            if j+1==100:
                j=98
            V1[i,j] = w/4*(V1[i-1,j]+V1[i+1,j] + V1[i,j-1] + V1[i,j+1])+(1-w) * V1[i,
j]
            if abs(V2[i,j]-V1[i,j]>maxj):
                maxj=abs(V2[i,j]-V1[i,j])
            else:
                maxj=1
print(iter1)
```

```
x=np.linspace(-1,1,102)
y=np.linspace(-1,1,102)
X,Y=np.meshgrid(x,y)
fig=plt.figure(figsize=(6,5))
range1=np.linspace(0,10,128)
height=plt.contour(X,Y,V1,40,cmap='rainbow')#等高线绘制
draw=plt.contourf(X,Y,V1,levels=range1,cmap='jet')
fig.colorbar(draw)
```



4 深度学习方法

4.1 核心思想

常见的低维度偏微分方程的求解使用数值求解可以获得较好的精度，可对于高维微分方程，由于网格划分时会引起维数灾难，因此，很难对于高维的偏微分方程进行求解，但是，随着AI技术的发展，使用神经网络算法求解偏微分方程也成为了研究的领域之一。

这里主要介绍一种最为简单基础的使用神经网络求解偏微分方程的算法(DGM)，是这个领域较早的方法之一，其后续思想的发展与PINN有很高的相似性。

文章利用神经网络求解微分方程的思想，总共分为三步：

1.构建深度网络模型，选择合适的求解精度、网络深度、宽度。

2.设计适当的损失函数。

3.生成数据，离散损失函数，通过优化离散的损失函数来训练神经网络参数。

其中，在生成数据时通常采取概率分布来采样。在随机采样的空间点使用随机梯度下降训练深度神经网络以满足微分算子、初始条件和边界条件（即满足初始方程）。通过随机采样空间点，避免了形成网格的需要（这在高维中是不可行的），将PDE问题转化为机器学习问题。

核心在于将物理约束加入神经网络的训练过程，即损失函数的设计上，将损失函数定义

为：

$$J(f)=\left\|\frac{\partial f}{\partial t}(t,x,\theta)+Lf(t\cdot x\cdot\theta)\right\|^2_{[0,T]\times\Omega,v_1}+\|f(t,x,\theta)-g(t,x)\|^2_{[0,T]\times\partial\omega,w_2}+\|f(0,x,\theta)-u_0(x)\|^2_{\omega,v_3}$$

第一部分代表原微分方程，第二部分代表边界条件，第三部分代表初始条件

4.2 Example6：热传导方程

方程的具体形式：

$$\frac{\partial u}{\partial t}=\frac{1}{2}\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right),\quad 0<x,y<\pi,t>0$$
$$u(x,y,0)=\sin x\sin y,\quad 0\le x,y\le\pi$$
$$u(0,y,t)=u(\pi,y,t)=0,\quad 0\le y\le\pi,t\ge 0$$
$$u(x,0,t)=u(x,\pi,t)=0,\quad 0\le y\le\pi,t\ge 0$$

```
import time
import torch
import torch.nn as nn
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import math
from torch.autograd import Variable
import torch.nn.functional as F
import torch.optim as optim
```

```
#网络架构
class zsrDGM_net(nn.Module):
    def __init__(self,numl,numn):
        super(zsrDGM_net, self).__init__()
        self.input_layer = nn.Linear(3, numn)#前面的数字代表几个输入
        self.hidden_layers = nn.ModuleList([nn.Linear(numn, numn) for i in
range(numl)])
        self.output_layer = nn.Linear(numn, 1)
    def forward(self, x):
        o = self.act(self.input_layer(x))
        for i, li in enumerate(self.hidden_layers):
            o = self.act(li(o))
        out = self.output_layer(o)
        return out
    def act(self, x):
        return x * torch.tanh(x)
```

```
#偏微分方程相关参数设计
class PDE():
    def __init__(self, net,t,size):
        self.net=net
        self.t=t
        self.size=size
    def sample(self):
        size=self.size
        sp = torch.cat((torch.full([N1*size, 1], 0) + torch.rand([N1*size, 1]) *
np.pi,torch.full([N1*size, 1], 0) + torch.rand([N1*size, 1]) *
np.pi,torch.rand([N1*size, 1]) * self.t), dim=1)
```

```
        sp_x = torch.full([N2*size, 1], 0) + torch.rand([N2*size, 1]) * np.pi
        sp_y = torch.full([N2*size, 1], 0) + torch.rand([N2*size, 1]) * np.pi
        sp_initial = torch.cat((sp_x,sp_y,torch.zeros(N2*size, 1)), dim=1)
        x_boundary_left = torch.cat(( torch.full([N3*size, 1], 0),torch.rand([N3*size,
1])*np.pi,torch.rand([N3*size, 1])*self.t), dim=1)
        x_boundary_right = torch.cat(( torch.full([N3*size, 1],
np.pi),torch.rand([N3*size, 1])*np.pi,torch.rand([N3*size, 1])*self.t), dim=1)
        y_boundary_left = torch.cat(( torch.rand([N3*size,
1])*np.pi,torch.full([N3*size, 1], 0),torch.rand([N3*size, 1])*self.t), dim=1)
        y_boundary_right = torch.cat(( torch.rand([N3*size,
1])*np.pi,torch.full([N3*size, 1], np.pi),torch.rand([N3*size, 1])*self.t), dim=1)

        return sp, sp_initial, sp_x,sp_y,x_boundary_left,
x_boundary_right,y_boundary_left, y_boundary_right
    def loss_func(self):
        size=self.size
        sp_train, sp_initial, sp_x,sp_y,x_boundary_left,
x_boundary_right,y_boundary_left, y_boundary_right = self.sample()
        sp = Variable(sp_train, requires_grad=True)
        d = torch.autograd.grad(net(sp), sp, grad_outputs=torch.ones_like(net(sp)),
create_graph=True)
        dx = d[0][:, 0].unsqueeze(-1)
        dy = d[0][:, 1].unsqueeze(-1)
        dt = d[0][:, 2].unsqueeze(-1)
        dxx = torch.autograd.grad(dx, sp, grad_outputs=torch.ones_like(dx),
create_graph=True)[0][:, 0].unsqueeze(-1)
        dyy = torch.autograd.grad(dy, sp, grad_outputs=torch.ones_like(dy),
create_graph=True)[0][:, 1].unsqueeze(-1)
        loss_fn = nn.MSELoss(reduction='mean')
        loss1 = loss_fn(dt, 1/2*(dxx+dyy))
        loss2 = loss_fn(net(sp_initial),
torch.zeros([N2*size,1])+np.sin(sp_x)*np.sin(sp_y))
        loss3 = loss_fn(net(x_boundary_left),
torch.zeros([N3*size,1])+torch.full([N3*size, 1], 0))
        loss4 =
loss_fn(net(x_boundary_right),torch.zeros([N3*size,1])+torch.full([N3*size, 1], 0))
        loss5 = loss_fn(net(y_boundary_left),
torch.zeros([N3*size,1])+torch.full([N3*size, 1], 0))
        loss6 =
loss_fn(net(y_boundary_right),torch.zeros([N3*size,1])+torch.full([N3*size, 1], 0))
        loss = loss1 + loss2 + loss3 + loss4 +loss5 +loss6
        #print(loss1,loss2,loss3,loss4)
        return loss
```

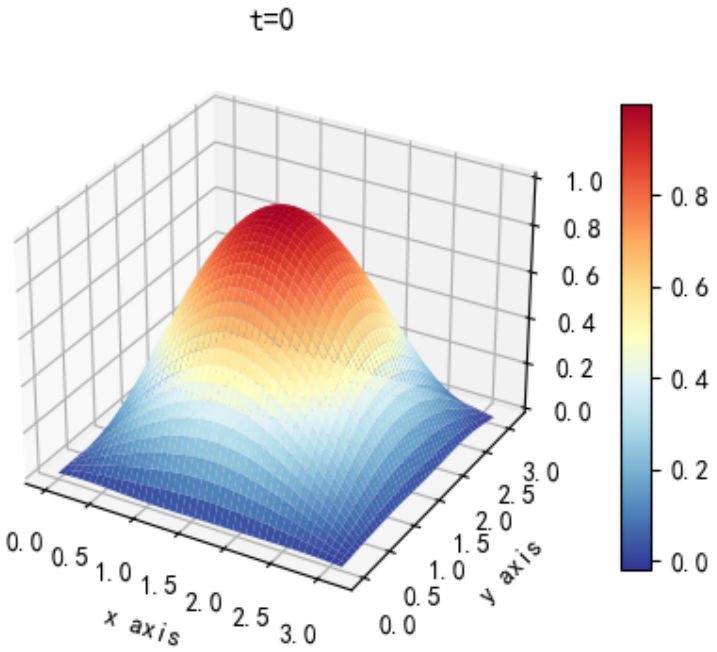
```
class Train():
    def __init__(self, net, eq, BATCH_SIZE):
        self.errors = []
        self.BATCH_SIZE = BATCH_SIZE
        self.net = net
        self.model = eq
    def train(self, epoch, lr):
        optimizer = optim.Adam(self.net.parameters(), lr)
        avg_loss = 0
        print('epoch start')
        for e in range(epoch):
            optimizer.zero_grad()
            loss = self.model.loss_func()
            avg_loss = avg_loss + float(loss.item())
            loss.backward()
            optimizer.step()
```

```
            if e % 100 == 99:
                loss = avg_loss/100
                print("Epoch {} - lr {} -  loss: {}".format(e, lr, loss))
                avg_loss = 0
                error = self.model.loss_func()
                self.errors.append(error.detach())
    def get_errors(self):
        return self.errors
```

```
#模型的训练
net = zsrDGM_net(numl=6, numn=150)#6层150个神经元
t=2.3
N1=4#对微分方程采样程度，值越大，样本越多
N2=1#初始条件的采样比值
N3=2#边界条件的采样比值
size=2**11
equation = PDE(net,t,size)
train = Train(net, equation, BATCH_SIZE=size)
train.train(epoch=1000, lr=0.001)
torch.save(net, 'test5.pkl')
errors = train.get_errors()
```

```
x = [[i/100] for i in range(0,314)]
y = torch.tensor([[i/100] for i in range(0,314)])
Z=[]
for i in x:
    u = torch.cat((torch.full([314, 1],i[0]),y,torch.full([314, 1],0)), dim=1)
    net=torch.load('test5.pkl')
    usolve=net(u)
    usolve=usolve.detach().numpy()
    Z.append(usolve)
```

```
xd=np.array([[i/100] for i in range(0,314)])
yd=np.array([[i/100] for i in range(0,314)])
_X, _Y = np.meshgrid(xd, yd, indexing='ij')
True_Z_surface1 = np.reshape(Z, (xd.shape[0], yd.shape[0]))
fig = plt.figure(figsize=(8, 4), facecolor='white') #创建图片
sub = fig.add_subplot(111, projection='3d')# 添加子图,
sub.set_zlim([0, 1])
#ax = fig.gca(projection='3d')
surf=sub.plot_surface(_X, _Y, True_Z_surface1, cmap=plt.cm.RdYlBu_r, edgecolor='blue',
linewidth=0.0003, antialiased=True)
cb = fig.colorbar(surf, shrink=0.8, aspect=15) #设置颜色棒
sub.set_xlabel(r"x axis")
sub.set_ylabel(r"y axis")
sub.set_zlabel(r"u axis")
plt.title("t=0")
plt.show()
```



求解误差小于0.001，具体形式见误差.mp4

总结

本课题主要对偏微分方程以及常微分方程的问题进行了总结归纳，从数学理论出发的解析法到有数学推导基础的数值求解方法，再到目前较新的但是缺乏严谨理论依据的神经网络方法进行了总结、研究与整理。

对比三种方法我们可以看出：

1、解析方法无疑是可以严格对方程进行求解的，但是实际应用中，应用面最窄，绝大部分的微分方程都难以解析求解。

2、以数学作为基础传统的数值方法在求解上拥有很好的精度，与较为完善的理论依据，应用目前最为广泛，但对于高维度的方程很难进行求解。

3、AI方法可以对高维问题进行求解，通过调整各项参数虽然能够提高精度，也几乎达不到数值求解的精度。在对于精度要求很高的实际情形中很难应用。且缺乏具体的理论依据证明求解的可靠性，其理论基础是神经网络万能逼近定理得来，对于调参的方法也缺乏具体方向性的指导，几层多少个神经元能达到最好的效果需要靠尝试。且现在很多方法仅仅适用于部分种类偏微分方程，其泛用性较差。但是对于高维度的问题，数值计算难以使用的领域，神经网络求解展现优势。

参考文献

[1]汪德新，数学物理方法

[2]Justin Sirignano, Konstantinos Spiliopoulos, DGM: A deep learning algorithm for solving partial differential equations,Journal of Computational Physics,Volume 375,2018,Pages 1339-1364,ISSN 0021-9991,

[3]李维国，现代数值计算