张量分析、Python符号计算与波数法公式的生成

于子叶

Cangye@hotmail.com

2017 年 4 月 20 日

1 数学问题

1.1 约束方程

对于物理场多场线性约束方程可以表示成如下的形式：

$$\mathcal{L} \cdot v(x, y, z) + \mathcal{M} \cdot f = 0 \tag{1}$$

其中$\mathcal{L}$代表二阶线性偏微分算子$\nabla\nabla, \nabla\times\nabla\times$等

在层状均匀介质的情况之下可以转换为以层状介质一维的情况：

$$\mathcal{D} \cdot m(z) + \mathcal{N} \cdot f = 0 \tag{2}$$

其中$\mathcal{D}$代表常微分算子$\frac{d}{dz}, \frac{d^2}{dz^2}$,求解方程过程中需要用一些降次和复态模分析的思想，这里不再具体阐述，见程序部分。

1.2 张量变换

在Python的sympy中并未定义张量在正交坐标系下的微分形式，这是不如Mathematica的部分，需要自己编写代码进行实现。这里阐述张量变换的方式数学原理。在空间中定义坐标变换：

$$z_i = z_i(x_1, x_2, \cdots, x_n) \tag{3}$$

定义在坐标变换上的(p,q)形张量表示为：

$$\mathcal{T}^{i_1,\cdots,i_p}_{j_1,\cdots,j_q} = \mathcal{T}^{k_1,\cdots,l_p}_{l_1,\cdots,l_q} \frac{\partial x_{i_1}}{\partial z_{k_1}} \cdots \frac{\partial x_{i_p}}{\partial z_{k_p}} \frac{\partial z_{l_1}}{\partial x_{j_1}} \cdots \frac{\partial z_{l_q}}{\partial x_{j_q}} \tag{4}$$

对于柱坐标变换举例来说：

$$x = r\ cos(\theta) y = r\ sin(\theta) z = z_1 \tag{5}$$

对于函数的梯度来说：

$$\nabla_{Cylindrical} f = (A^T)^{-1} \nabla_x f \tag{6}$$

变换后为：

$$\begin{pmatrix} \cos(\theta)\frac{\partial f(r,\theta,z)}{\partial x} - \frac{\sin(\theta)\partial f(r,\theta,z)/\partial\theta(r,\theta,z)}{r} \\ \sin(\theta)\frac{\partial f(r,\theta,z)}{\partial x} + \frac{\cos(\theta)\partial f(r,\theta,z)/\partial\theta}{r} \\ \frac{\partial f(r,\theta,z)}{\partial z}(r,\theta,z) \end{pmatrix} \tag{7}$$

这个分量是在原来的xyz坐标之下的分量，显然对于柱坐标需要的是一个旋转：

$$\begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{8}$$

相乘之后得到柱坐标之下梯度：

$$[\frac{\partial f(r,\theta,z)}{\partial r}, \frac{1}{r}\frac{\partial f(r,\theta,z)}{\partial r}, \frac{\partial f(r,\theta,z)}{\partial r}] \tag{9}$$

1.3 方程求解

对于均匀介质参数减少可以变成常微分方程，这里利用球谐函数对向量进行展开。

$$v^1 = \frac{1}{2\pi}\sum_{m=-\infty}^{\infty}\int_0^{\infty} v_t \cdot T_k^m(r,\theta) + v_s \cdot S_k^m(r,\theta) + v_r \cdot R_k^m(r,\theta) \tag{10}$$

其中

$$\begin{pmatrix} T_k^m(r,\theta) = k^{-1}\nabla \times e_z Y_k^m(r,\theta) \\ S_k^m(r,\theta) = k^{-1}\nabla Y_k^m(r,\theta) \\ T_k^m(r,\theta) = -e_z Y_k^m(r,\theta) \end{pmatrix} \tag{11}$$

偏微分方程转换化简后对于转换后的常微分方程

$$\mathcal{D}\cdot m(z) + \mathcal{N}\cdot f = 0 \tag{12}$$

其求解方式在于将方程转换为线性不相关的方程。这个过程在于将矩阵$\mathcal{D}$转换为对角矩阵：

$$\mathcal{D} = \mathcal{E}^{-1}\cdot\mathcal{A}\cdot\mathcal{E}^{-1} \tag{13}$$

其$\mathcal{A}$为对角矩阵。

2 Python的sympy实现上述过程

2.1 Bessel函数的实现

主要作用在于定义Bessel函数的导数。

```

class BesselBase(Function):
    def __init__(self,*args):
        self.dn=1
    @property
    def order(self):
        return self.args[0]
    @property
```

```
    def argument(self):
        return self.args[1]
    @classmethod
    def eval(cls, nu, z):
        return
    def fdiff(self, argindex=2):
        if argindex != 2:
            raise ArgumentIndexError(self, argindex)
        if(self.order-m==0):
            a=(self.__class__(self.order+1, self.argument))
            return a
        elif(self.order-m==1):
            a=(self.__class__(self.order-1, self.argument))
            b=(self.__class__(self.order, self.argument))
            xx=self.argument
            return -((-m**2+xx**2)*a+xx*b)/xx**2
        elif(self.order-m==2):
            a=(self.__class__(self.order+1, self.argument))
            b=(self.__class__(self.order+1, self.argument))
            xx=self.argument
            return (a*(-3*m**2 + xx**2)-
              b*(xx - m**2*xx + xx**3 - 3*xx))/xx**3

        return (self.__class__(self.order + 1, self.argument))
    def _eval_conjugate(self):
        z = self.argument
        if (z.is_real and z.is_negative) is False:
            return self.__class__(self.order.conjugate(), z.conjugate())
    def _eval_expand_func(self, **hints):
        nu, z, f = self.order, self.argument, self.__class__
        if nu.is_real:
            if (nu - 1).is_positive:
                return (-self._a*self._b*f(nu - 2, z)._eval_expand_func() +
                        2*self._a*(nu - 1)*f(nu - 1, z)._eval_expand_func()/z)
            elif (nu + 1).is_negative:
                return (2*self._b*(nu + 1)*f(nu + 1, z)._eval_expand_func()/z -
                        self._a*self._b*f(nu + 2, z)._eval_expand_func())
        return self
    def _eval_simplify(self, ratio, measure):
        from sympy.simplify.simplify import besselsimp
        return besselsimp(self)
```

2.2 微分函数定义

定义微分的方法：

```
class MyTensorMethod():
    def __init__(self, syms):
        self.symb=syms
    def grad(self,tens):
        self.coord(self.symb[0],self.symb[1],self.symb[2])
        retens = Matrix(tens.diff(self.symb[0]))
        ct = 1
        for sym in self.symb[1:]:
```

```
            retens = retens.row_insert(ct, tens.diff(sym))
            ct += 1
        retens = self.invA*retens
        retens = simplify(transpose(self.rot.inv())*retens)
        #reeye=ss.Matrix().
        return retens.transpose()
    def grad_2d(self,tens):
        self.coord(self.symb[0],self.symb[1],self.symb[2])
        retens = Matrix(tens.diff(self.symb[0]))
        ct = 1
        for sym in self.symb[1:]:
            retens = retens.row_insert(ct, tens.diff(sym))
            ct += 1
        retens = self.invA*retens
        retens = simplify(transpose(self.rot.inv())*retens)
        reeye=sp.zeros(3, 3)
        ssx=tens
        reeye[0,1]=-ssx[0,1]/self.symb[0]
        reeye[1,1]=ssx[0,0]/self.symb[0]
        return retens.transpose()+reeye
    def coord(self,x1,x2,x3):
        self.transmatrix=sp.Matrix([[x1*sp.cos(x2),x1*sp.sin(x2),x3]])
        self.A = sp.Matrix(self.transmatrix.diff(x1))
        self.A = self.A.row_insert(1, self.transmatrix.diff(x2))
        self.A = self.A.row_insert(2, self.transmatrix.diff(x3))
        self.invA = sp.simplify(self.A.inv())
        self.rot=sp.Matrix([[ sp.cos(x2), sp.sin(x2), 0],
                            [-sp.sin(x2), sp.cos(x2), 0],
                            [          0,          0, 1]])
    def curl(self,tens):
        ssx=tens
        ts = Matrix(tens.copy().diff(self.symb[0]))
        ct = 1
        for sym in self.symb[1:]:
            ts = ts.row_insert(ct, tens.copy().diff(sym))
            ts = ts.row_insert(ct, tens.copy().diff(sym))
            ct += 1
        re = Matrix([[-ts[2, 1]+ts[1, 2]/self.symb[0]]])
        re=re.row_insert(1, Matrix([[ts[2,0]-ts[0,2]]]))
        re=re.row_insert(2, Matrix([[-ts[1,0]/self.symb[0]+ts[0,1]+ssx[1]/self.symb[0]]]))
        return re.transpose()
    def div(self,tens):
        ts = Matrix(tens.copy().diff(self.symb[0]))
        ct = 1
        for sym in self.symb[1:]:
            ts = ts.row_insert(ct, tens.copy().diff(sym))
            ct += 1
        return ts[0,0]+ts[1,1]/self.symb[0]+ts[2,2]+tens[0]/self.symb[0]
    def div_2d(self,tens):
        ts = Matrix(tens.diff(self.symb[0]))
        ct = 1
        for sym in self.symb[1:]:
            ts = ts.row_insert(ct, tens.diff(sym))
            ct += 1
        ois=Matrix([[1][1/self.symb[0]][1]])
```

```
        tsi=ts*ois

        vectx=ois[0,0]+(tens[0,0]-tens[1,1])/self.symb[0]
        vecty=ois[1,0]+(tens[0,1]+tens[1,0])/self.symb[0]
        vectz=ois[2,0]+(tens[2,0])/self.symb[0]
        return Matrix([[vectx,vecty,vectz]])
```

2.3 计算过程定义

定义波数法计算过程

```
class Formula():
    def get_vect(self):
        k=symbols("k")
        bl=mybsl(m,self.cod[0]*k)
        Y=exp(I*m*self.cod[1])*bl
        Y=exp(I*m*self.cod[1])*bl
        Y=exp(I*m*self.cod[1])*bl

        T=self.ms.curl(Matrix([[0,0,Y.copy()/k]]))
        S=self.ms.grad(Matrix([[Y.copy()]]))/k
        R=Matrix([[0,0,-Y.copy()]])
        vt=Function("vt")(self.cod[2])
        vs=Function("vs")(self.cod[2])
        vr=Function("vr")(self.cod[2])
        self.v=[vt,vs,vr]
        vect=T*self.v[0]+S*self.v[1]+R*self.v[2]
        return vect
    def __init__(self):
        x1,x2,x3,k,r=symbols("r,o,z,k,r")
        self.cod=[x1,x2,x3]
        self.syms=self.cod
        self.ms=MyTensorMethod(self.cod)
    def get_formlua(self,fom):
        k=symbols("k")
        vect=self.get_vect()
        defi=sp.simplify(fom/exp(I*m*self.cod[1])*k*self.cod[0])
        func=[]
        func.append(simplify(defi[0].diff(mybsl(m,self.cod[0]*k))/I))
        func.append(simplify(defi[1].diff(mybsl(m,self.cod[0]*k))/I))
        func.append(simplify(defi[2].diff(mybsl(m,self.cod[0]*k))/k/self.cod[0]))
        nm=len(vect)
        nm2=len(vect)*2
        mat=sp.zeros(len(vect)*2,len(vect)*2)
        for itry in range(nm):
            for itrx in range(nm):
                mat[itry,itrx]=func[itry].diff(self.v[itrx])
        for itry in range(nm):
            for itrx in range(nm):
                mat[itry,itrx]=mat[itry,itrx]+func[itry].diff(self.v[itrx].diff(self.cod[2]))
        for itry in range(3,nm2-1):
            for itrx in range(3,nm2-1):
                mat[itry,itrx]=mat[itry,itrx]+func[itry-3].diff(self.v[itrx-3].diff(self.cod[2]).
                    diff(self.cod[2]))
```

```
        egv=simplify(mat.eigenvects())
        mtE=Matrix(egv[0][2][0].transpose())
        for itr in range(1,nm2-1):
            mtE=mtE.row_insert(itr,egv[itr][2][0].transpose())
        file=open("formula.txt","w")
        file.write(latex(simplify(mat)))
        file.write("\n\n")
        file.write(latex(simplify(mtE)))
        pprint(simplify(mat))
        pprint(simplify(mtE))
    def get_method(self):
        return self.ms
```

2.4 定义公式和计算结果

为了简便将公式定义为：

$$\nabla \times \nabla \times v + \omega f = 0 \tag{14}$$

这部分代码为:

```
omega=symbols("\omega")
test=Formula()
vect=test.get_vect()
ms=test.get_method()
#Define the formula
formula=ms.curl(ms.curl(vect))+vect*omega
test.get_formlua(formula)
```

输出的结果为：

$$\begin{pmatrix} m\left(\omega + k^2\right) & 0 & 0 \\ 0 & 0 & 0 \\ 0 & \omega m & -km \\ 0 & 0 & 0 \\ 0 & -k & -\omega - k^2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ -m & 0 & 0 \\ 0 & 0 & 0 \\ 0 & -m & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{15}$$