核科学与技术学院

数值计算方法及其应用
大作业

姓　　名：刘珩骞

班　　级：20191515

学　　号：2019151514

任课教师：王俊玲

哈尔滨工程大学

核科学与技术学院

2023 年 4 月 22 日

摘要

本次作业内容集数次小作业之和，因此对于算法原理实现部分不再赘述。本次报告内全部的基础数值代码均为从底层构建，总体上分为以下四个算法类：`FitSquares_polynomial`、`LinerInterpolation`、`CubicSplineFree`、`MatrixSolver`、`RK4_for_equations`分别进行最小二乘多项式拟合、分段线性插值、分段样条插值、线性方程求解与四阶龙格库塔微分方程组求解，这些代码将在附录中体现。具体的使用方式将在特定章节展开。对于ODE问题，为了验证正确性，将通过Modelica进行对比验证。

问题一：线性方程和非线性方程的解

对于反应堆堆芯^{135}Xe与^{135}I的平均浓度方程：

$$\begin{cases}\left(r_X\Sigma_f-\sigma_a^X X\right)\Phi_0 n_r+\lambda_I I-\lambda_X X=0\\ r_I\Sigma_f\Phi_0 n_r-\lambda_I I=0\end{cases}$$

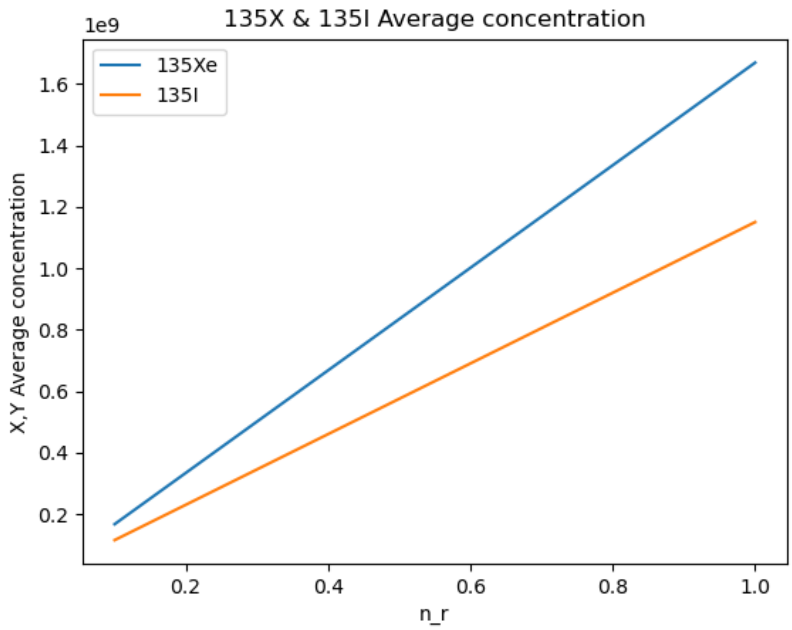
整理可以写为线性方程组$AX=b$的形式：

$$\begin{bmatrix} -\lambda_X + \sigma_a^X n_r & \lambda_I \\ 0 & -\lambda_I \end{bmatrix} \begin{bmatrix} X \\ I \end{bmatrix} = \begin{bmatrix} -r_X \Sigma_f \phi_0 n_r \\ -r_I \Sigma_f \phi_0 n_r \end{bmatrix}$$

由于n_r是给定的，为此求解在每个给定的n_r下的线性方程组即可。这里将通过LU分解进行求解。最后的求解结果为：

n_r	**0.1**	**0.2**	**0.3**	**0.4**	**0.5**	**0.6**	**0.7**	**0.8**	**0.9**	**1.0**
$X(10^8)$	1.6992	3.3383	5.0075	6.6767	8.3459	10.015	11.684	13.353	15.023	16.692
$I(10^8)$	1.1502	2.3004	3.4507	4.6009	5.7511	6.9014	8.0516	9.2018	10.352	11.502

平均浓度变化曲线为



python算法实现

```
import numpy as np
import matplotlib.pyplot as plt
import MartrixSolverLU
'''
反应堆堆芯中135X和135I的平均浓度曲线
(r_X * Sigma_f - sigma_a_X * X) * Phi_0 * n_r + lambda_I * I - lambda_X * X = 0
r_I * Sigma_f * Phi_0 * n_r - lambda_I * I = 0

线性方程组AX = b为:
    | -lambda_X + sigma_a_X * n_r , lambda_I    | | X |    | -r_X * Sigma_f * Phi_0 * n_r
|
    |                                           | |   | = |
|
    | 0                           , -lambda_I   | | I |    | -r_I * Sigma_f * Phi_0 * n_r
|

'''
```

```python
# 反应堆基本参数
P0 = 3000                   # MW, 反应堆功率
D = 316                     # cm, 堆芯直径
h = 355                     # cm, 堆芯高度
r_I = 0.059                 # 135I的裂变产额
r_X = 0.003                 # 135Xe的裂变产额
Sigma_f = 0.3358            # cm2, 裂变截面
sigma_a_X = 3.5E-18         # cm2, 135Xe吸收截面
lambda_I = 2.9E-5           # 1/s, 135I衰变常数
lambda_X = 2.1E-5           # 1/s, 135Xe衰变常数
Eff = 3.2E-11               # MWs,裂变效率
n_r = np.arange(0.1,1+0.1,0.1) # 中子的相对密度

# 计算堆芯体积
V = 2 * np.pi * (0.5 * D)**2 * h

# 计算初始平均中子注量率
Phi_0 = P0/(Eff*V)

# 构造系数矩阵
A = lambda n_r: np.array([[-lambda_X + sigma_a_X * Phi_0 * n_r, lambda_I],
                          [0, -lambda_I]])
# 构造右端项
b = lambda n_r: np.array([-r_X * Sigma_f * Phi_0 * n_r,
                          -r_I * Sigma_f * Phi_0 * n_r])
# 求解线性方程组
X = np.zeros((len(n_r),2))
Xs = np.zeros((len(n_r),2))
for i in range(len(n_r)):
    #X[i] = np.linalg.solve(A(n_r[i]),b(n_r[i]))
    X[i] = MartrixSolverLU.MartrixSolver(A(n_r[i]),b(n_r[i]))
print("X average concentration: \n",X[:,0])
print("I average concentration: \n",X[:,1])

# 绘制曲线
plt.figure()
plt.plot(n_r,X[:,0],label='135Xe')
plt.plot(n_r,X[:,1],label='135I')
plt.xlabel('n_r')
plt.ylabel('X,Y Average concentration')
plt.title('135X & 135I Average concentration')
plt.legend()
plt.show()
```

问题二：水位模型参数的处理

为了便于之后计算，首先我们需要将参数图标转化为方便我们使用的数值类型，这里使用`class Parameter()`对参数进行声明。使用时通过`Parameter.G1`的方式即可。

```python
class Parameter():
    #功率水平
    p = np.array([0.1,0.2,0.3,0.5,1]) #10%,...100%
    G1 = np.array([0.0031,0.0035,0.0035,0.0035,0.0035])
    G2 = np.array([0.402,0.339,0.256,0.188,0.131])
    G3 = np.array([0.166,0.207,0.143,0.055,0.028])
    tau_0 = np.array([10,10.4,8.0,6.4,4.7])
    tau = np.array([19.7,12.5,10.3,13.3,6.6])
    xi = np.array([0.65,1.6,1.6,0.62,1.68])
    beta = np.array([-0.08,0.44,0.47,0.20,0.20])
```

第一问：最小二乘多项式拟合法

对多项式的最小二乘法拟合采用`FitSquares_polynomial`方法，需要把x、y写成[x,y]形式。二次最小二乘法获得系数为

	1	x	x^2
G1系数	0.00306072	0.00169755	-0.00127227
G2系数	0.4818558	-0.85450954	0.50445399
G3系数	0.23947977	-0.4211376	0.20574827

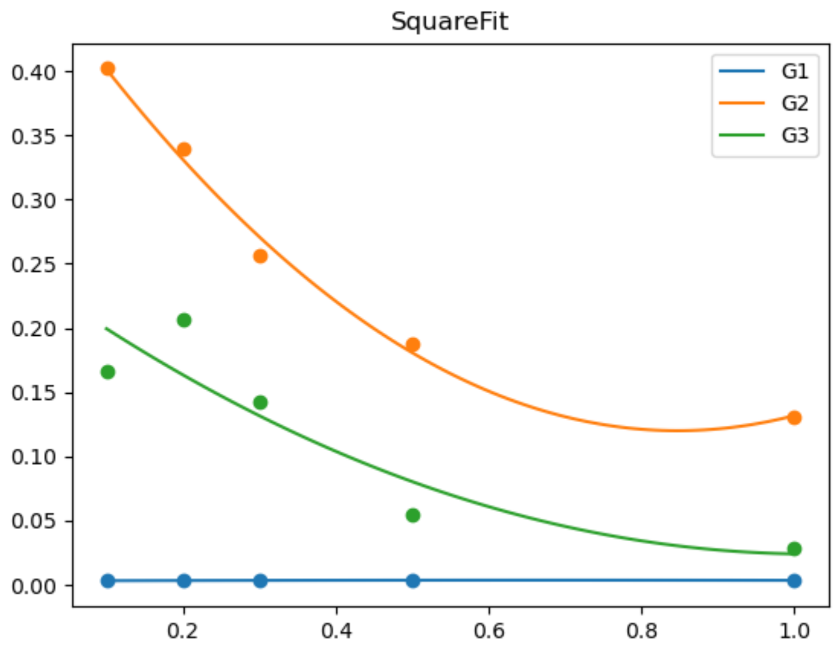
对应的误差为：（最小）

G1误差	**G2误差**	**G3误差**
1.96035E-10	3.031805E-7	1.5284642E-5

在功率$p = [0.15, 0.4, 0.8]$处对应的值为：

	0.15	**0.40**	**0.80**
G1近似	0.00328673	0.00353618	0.00360451
G2近似	0.36502958	0.22076462	0.12109872
G3近似	0.18093847	0.10394446	0.03424859

拟合曲线为：



第二问：分段线性插值与分段样条插值

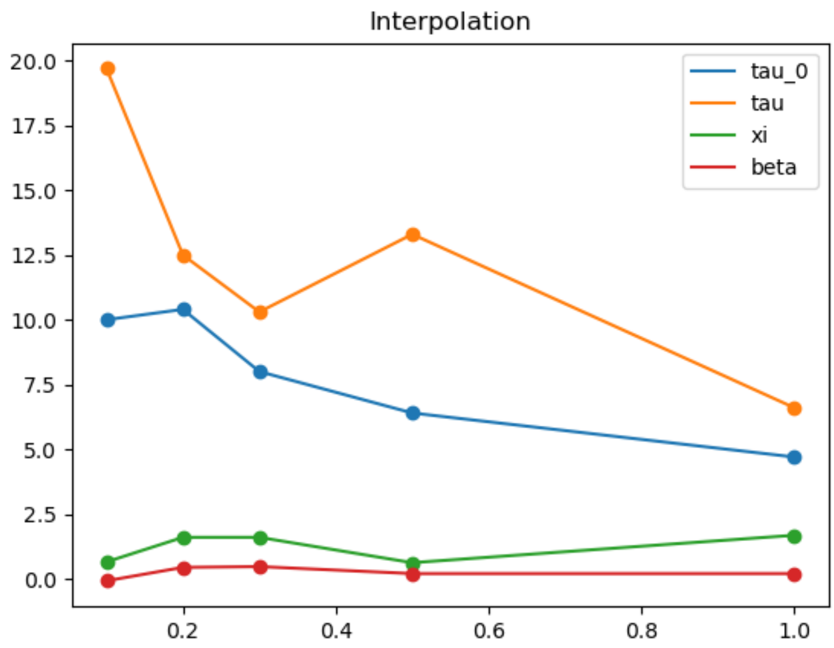
分段线性插值部分

分段线性插值非常简单，几乎都有现成的库。但是实现起来由于需要确定插值点在数据点中的位置，优化起来很困难。但是还是可以优化的，代码中包含了numpy.interp版本与自建`LinearInterpolation`版本。

在$p = [0.25, 0.75, 0.95]$处的插值为

	0.25	**0.75**	**0.95**
τ_0	9.2	5.55	4.87
τ	11.4	9.95	7.27
ξ	1.6	1.15	1.574
β	0.455	0.2	0.2

插值图像为



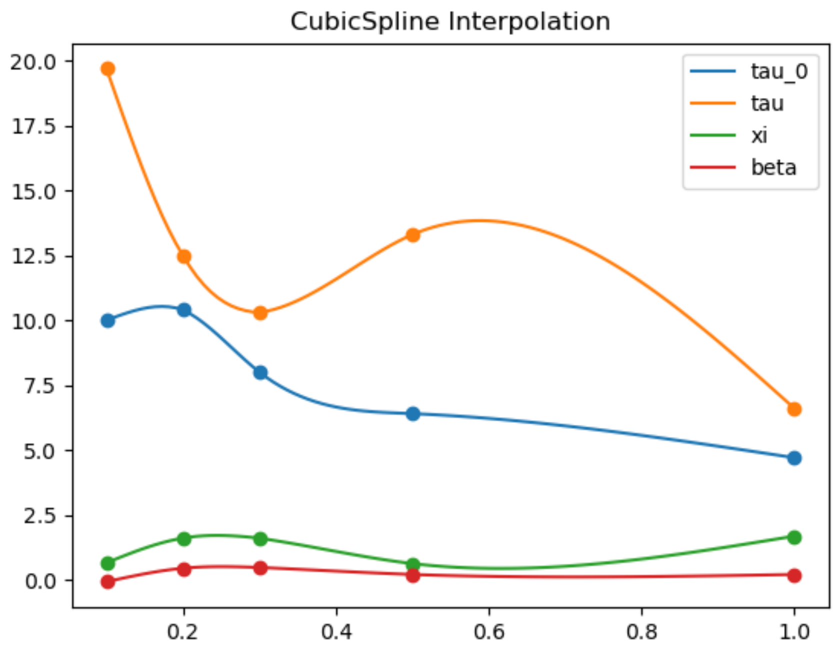
分段样条插值部分

分段样条部分采用的是三次自由样条插值，通过`CubicSplineFree`进行实现。

在$p=[0.25,0.75,0.95]$处的插值为

	0.25	0.75	0.95
τ_0	9.31875	5.753125	4.923625
τ	10.77875	12.3875	7.9135
ξ	1.704375	0.634375	1.437875
β	0.503125	0.10625	0.17525

插值图像为



python算法实现

从此题开始均通过`if __name__ == "__main__":`创建算法主分支，因为可以将其余部分进行cython优化运行。

```
import numpy as np
import matplotlib.pyplot as plt
import FitSquares as fs
import CubicSplineFree as csf
import LinearInterpolation as li
#表格内容
class Parameter():
    #功率水平
    p = np.array([0.1,0.2,0.3,0.5,1]) #10%,...100%
    G1 = np.array([0.0031,0.0035,0.0035,0.0035,0.0035])
    G2 = np.array([0.402,0.339,0.256,0.188,0.131])
    G3 = np.array([0.166,0.207,0.143,0.055,0.028])
    tau_0 = np.array([10,10.4,8.0,6.4,4.7])
    tau = np.array([19.7,12.5,10.3,13.3,6.6])
    xi = np.array([0.65,1.6,1.6,0.62,1.68])
    beta = np.array([-0.08,0.44,0.47,0.20,0.20])

"1. 最小二乘法拟合"
def SquareFit():
    G1 = np.c_[Parameter.p,Parameter.G1]
    G2 = np.c_[Parameter.p,Parameter.G2]
    G3 = np.c_[Parameter.p,Parameter.G3]

    fG1 = fs.FitSquares_polynomial(G1,3)
    fG2 = fs.FitSquares_polynomial(G2,3)
    fG3 = fs.FitSquares_polynomial(G3,3)
```

```python
#各项系数
aG1 = fG1.phiprod()[0]
aG2 = fG2.phiprod()[0]
aG3 = fG3.phiprod()[0]
print("G1系数: ",aG1)
print("G2系数: ",aG2)
print("G3系数: ",aG3)

#拟合误差
deltaG1 = fG1.delta()
deltaG2 = fG2.delta()
deltaG3 = fG3.delta()
print("G1误差: ",deltaG1)
print("G2误差: ",deltaG2)
print("G3误差: ",deltaG3)

#计算规定处近似值
p_c = np.array([0.15,0.4,0.8])
G1_assume = np.zeros(3)
G2_assume = np.zeros(3)
G3_assume = np.zeros(3)
for i in range(3):
    G1_assume[i] = fG1.num(p_c[i])
    G2_assume[i] = fG2.num(p_c[i])
    G3_assume[i] = fG3.num(p_c[i])
print("G1近似值: ",G1_assume)
print("G2近似值: ",G2_assume)
print("G3近似值: ",G3_assume)

#绘制拟合曲线
x = np.linspace(0.1,1,100)
y1 = np.zeros(100)
y2 = np.zeros(100)
y3 = np.zeros(100)
for i in range(100):
    y1[i] = fG1.num(x[i])
    y2[i] = fG2.num(x[i])
    y3[i] = fG3.num(x[i])
plt.plot(x,y1,label="G1")
plt.plot(x,y2,label="G2")
plt.plot(x,y3,label="G3")
plt.scatter(Parameter.p,Parameter.G1)
plt.scatter(Parameter.p,Parameter.G2)
plt.scatter(Parameter.p,Parameter.G3)
plt.title("SquareFit")
plt.legend()
plt.show()
```

```python
"2. 分段线性、分段样条插值"
tau_0 = np.c_[Parameter.p,Parameter.tau_0]
tau = np.c_[Parameter.p,Parameter.tau]
xi = np.c_[Parameter.p,Parameter.xi]
beta = np.c_[Parameter.p,Parameter.beta]

def LinearInterpolation():
    #分段线性插值
    #由于分段线性插值概念上比较简单，但是代码实现上比较复杂，因此直接调用库函数
    p_c = np.array([0.25,0.75,0.95])
    tau_0_interp = np.interp(p_c,tau_0[:,0],tau_0[:,1])
    tau_interp = np.interp(p_c,tau[:,0],tau[:,1])
    xi_interp = np.interp(p_c,xi[:,0],xi[:,1])
    beta_interp = np.interp(p_c,beta[:,0],beta[:,1])
    print("tau_0 近似值_分段线性插值: ",tau_0_interp)
    print("tau 近似值_分段线性插值: ",tau_interp)
    print("xi 近似值_分段线性插值: ",xi_interp)
    print("beta 近似值_分段线性插值: ",beta_interp)

    #绘制插值曲线
    x = np.linspace(0.1,1,100)
    plt.plot(x,np.interp(x,tau_0[:,0],tau_0[:,1]),label="tau_0")
    plt.plot(x,np.interp(x,tau[:,0],tau[:,1]),label = "tau")
    plt.plot(x,np.interp(x,xi[:,0],xi[:,1]),label = "xi")
    plt.plot(x,np.interp(x,beta[:,0],beta[:,1]),label = "beta")
    plt.scatter(Parameter.p,Parameter.tau_0)
    plt.scatter(Parameter.p,Parameter.tau)
    plt.scatter(Parameter.p,Parameter.xi)
    plt.scatter(Parameter.p,Parameter.beta)
    plt.legend()
    plt.title("Linear Interpolation")
    plt.show()

def LinI():
    #分段线性插值,通过构建LinearInterpolation类实现
    p_c = np.array([0.25,0.75,0.95])
    tau_0_assume_LI = np.zeros(3)
    tau_assume_LI = np.zeros(3)
    xi_assume_LI = np.zeros(3)
    beta_assume_LI = np.zeros(3)
    for i in range(3):
        tau_0_assume_LI[i] = li.LinearInterpolation(p_c[i],tau_0)
        tau_assume_LI[i] = li.LinearInterpolation(p_c[i],tau)
        xi_assume_LI[i] = li.LinearInterpolation(p_c[i],xi)
        beta_assume_LI[i] = li.LinearInterpolation(p_c[i],beta)
    print("tau_0 近似值_分段线性插值: ",tau_0_assume_LI)
    print("tau 近似值_分段线性插值: ",tau_assume_LI)
    print("xi 近似值_分段线性插值: ",xi_assume_LI)
    print("beta 近似值_分段线性插值: ",beta_assume_LI)
```

```python
    #绘制插值曲线
    x = np.linspace(0.1,1,100)
    y1 = np.zeros(100)
    y2 = np.zeros(100)
    y3 = np.zeros(100)
    y4 = np.zeros(100)
    for i in range(100):
        y1[i] = li.LinearInterpolation(x[i],tau_0)
        y2[i] = li.LinearInterpolation(x[i],tau)
        y3[i] = li.LinearInterpolation(x[i],xi)
        y4[i] = li.LinearInterpolation(x[i],beta)
    plt.plot(x,y1,label="tau_0")
    plt.plot(x,y2,label="tau")
    plt.plot(x,y3,label="xi")
    plt.plot(x,y4,label="beta")
    plt.scatter(Parameter.p,Parameter.tau_0)
    plt.scatter(Parameter.p,Parameter.tau)
    plt.scatter(Parameter.p,Parameter.xi)
    plt.scatter(Parameter.p,Parameter.beta)
    plt.legend()
    plt.title("Interpolation")
    plt.show()



def CubicSplineInterpolation():
    #分段样条插值
    fcbtau_0 = csf.CubicSplineFree(tau_0)
    fcbtau = csf.CubicSplineFree(tau)
    fcbxi = csf.CubicSplineFree(xi)
    fcbbeta = csf.CubicSplineFree(beta)

    #计算规定处近似值
    p_c = np.array([0.25,0.75,0.95])
    tau_0_assume_cb = np.zeros(3)
    tau_assume_cb = np.zeros(3)
    xi_assume_cb = np.zeros(3)
    beta_assume_cb = np.zeros(3)
    for i in range(3):
        tau_0_assume_cb[i] = fcbtau_0.num(p_c[i])
        tau_assume_cb[i] = fcbtau.num(p_c[i])
        xi_assume_cb[i] = fcbxi.num(p_c[i])
        beta_assume_cb[i] = fcbbeta.num(p_c[i])
    print("tau_0 近似值_分段样条插值: ",tau_0_assume_cb)
    print("tau   近似值_分段样条插值: ",tau_assume_cb)
    print("xi    近似值_分段样条插值: ",xi_assume_cb)
    print("beta  近似值_分段样条插值: ",beta_assume_cb)
```

```python
    #绘制插值曲线
    x = np.linspace(0.1,1,100)
    y1 = np.zeros(100)
    y2 = np.zeros(100)
    y3 = np.zeros(100)
    y4 = np.zeros(100)
    for i in range(100):
        y1[i] = fcbtau_0.num(x[i])
        y2[i] = fcbtau.num(x[i])
        y3[i] = fcbxi.num(x[i])
        y4[i] = fcbbeta.num(x[i])
    plt.plot(x,y1,label="tau_0")
    plt.plot(x,y2,label="tau")
    plt.plot(x,y3,label="xi")
    plt.plot(x,y4,label="beta")
    plt.scatter(Parameter.p,Parameter.tau_0)
    plt.scatter(Parameter.p,Parameter.tau)
    plt.scatter(Parameter.p,Parameter.xi)
    plt.scatter(Parameter.p,Parameter.beta)
    plt.legend()
    plt.title("CubicSpline Interpolation")
    plt.show()


if __name__ == "__main__":
    SquareFit()
    CubicSplineInterpolation()
    LinearInterpolation()
    LinI()
```

第三问：常微分方程组初值问题

蒸汽发生器水位控制系统的微分方程表示为：

$$\begin{aligned}
&\dot{x}_1 = G_1\left(q_e - q_v\right)\\
&\dot{x}_2 = -\frac{1}{\tau_0}x_2 + \frac{G_2}{\tau_0}q_v\\
&\dot{x}_3 = \frac{1}{\tau^2}x_4 - \frac{G_3\beta}{\tau}q_e\\
&\dot{x}_4 = -x_3 - \frac{2\xi}{\tau}x_4 + (2\xi\beta - 1)G_3q_e\\
&y(t) = x_1 + x_2 + x_3\\
&x_1(0) = x_2(0) = x_3(0) = x_4(0) = 0
\end{aligned}$$

主要求解前四个组成的微分方程组。通过`RK4_for_equations`即可求解。需要注意的是，这里$f(x,y)$表示的形式是$f(t,\mathbf{x})$，即$f(t,x_1,x_2,x_3,x_4)$。但是对于我们的求解器而言，我们只需要指定一下方程组的规模数即可。

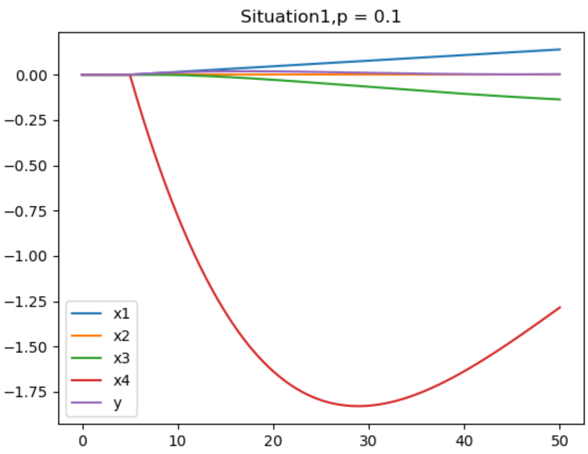
对于需要求解的三种情况，总仿真时间为$time: [0, 50]$，仿真总间隔数为$step: 1000$，每种情况遍历表中给定的五个功率点，求解器的使用方式为：

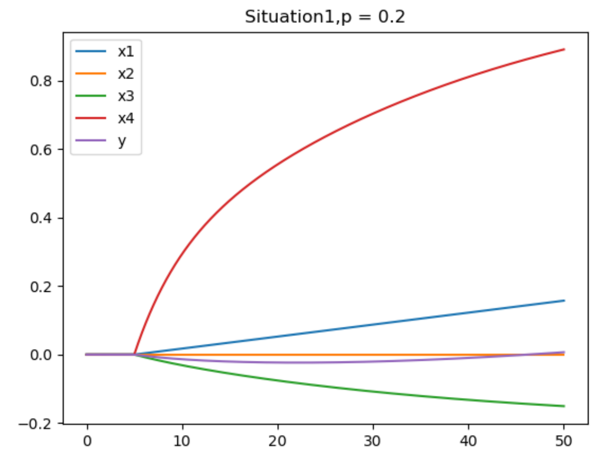
```
solver = ODE.RK4_for_equations(Func,N,start_time,end_time,totel_step,start_bound)
solver.slover()
```

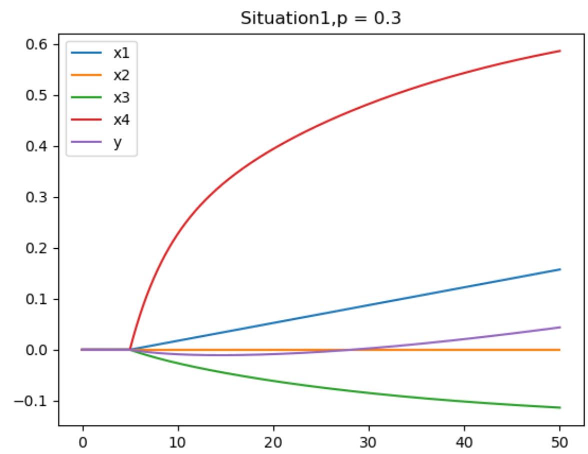
基于Python算法的求解

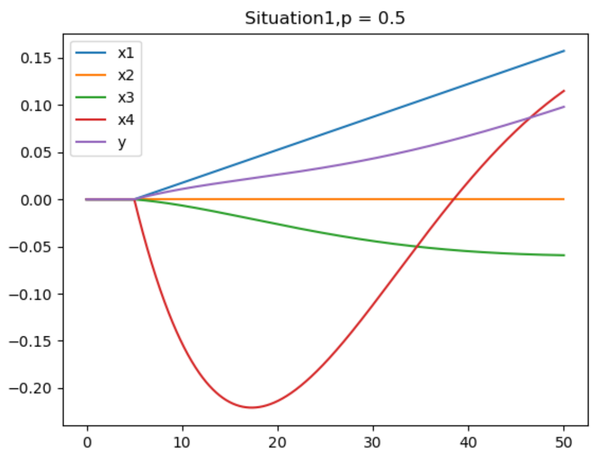
情况一：

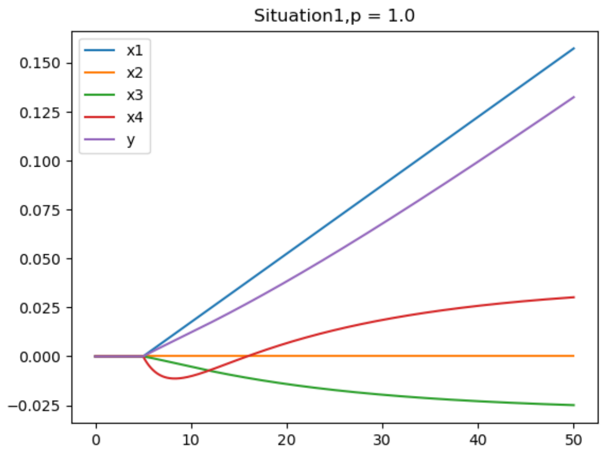
$x_1, x_2, x_3, x_4, y(t)$的解绘制的曲线为：





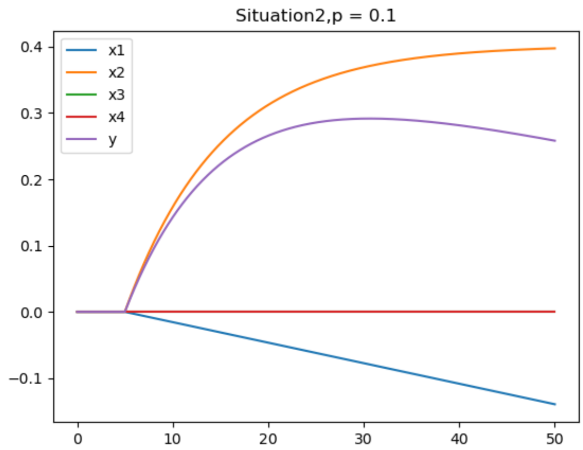


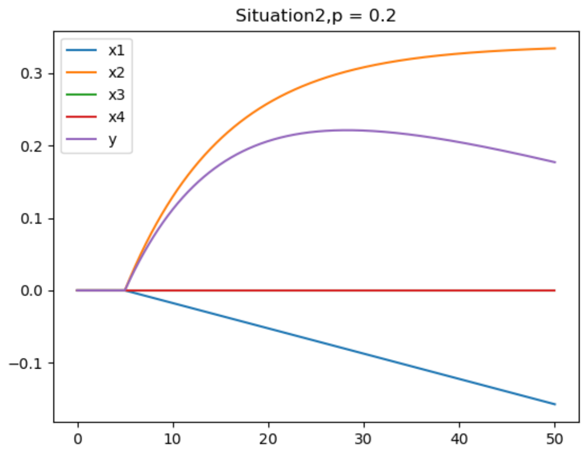


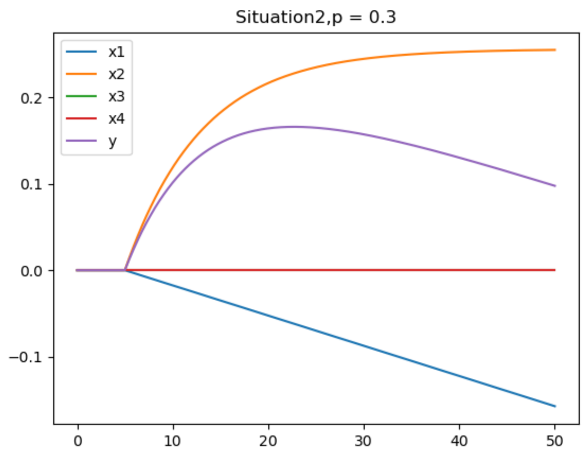


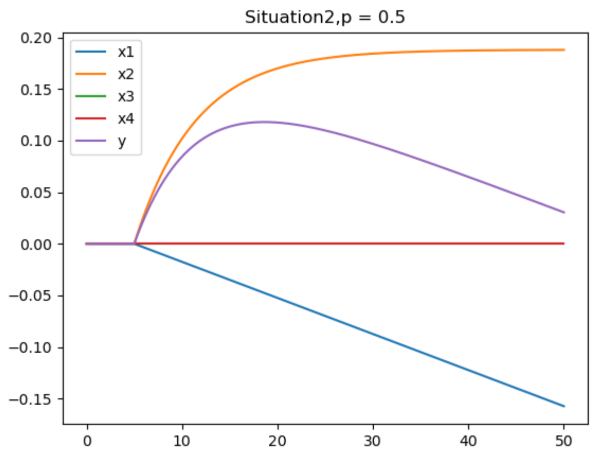
情况二：

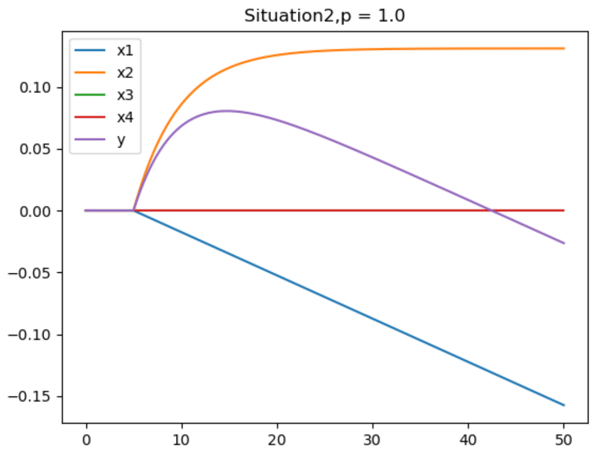
$x_1, x_2, x_3, x_4, y(t)$的解绘制的曲线为：





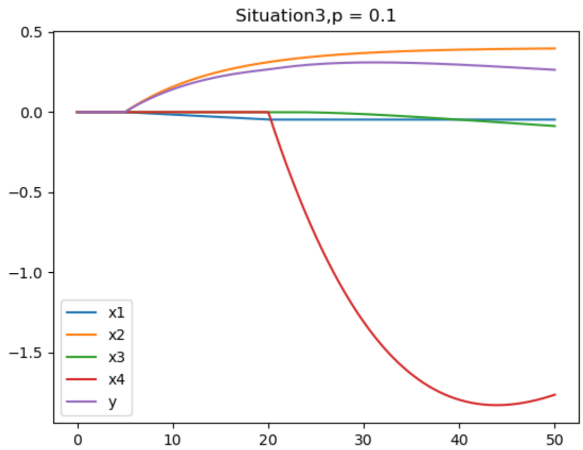


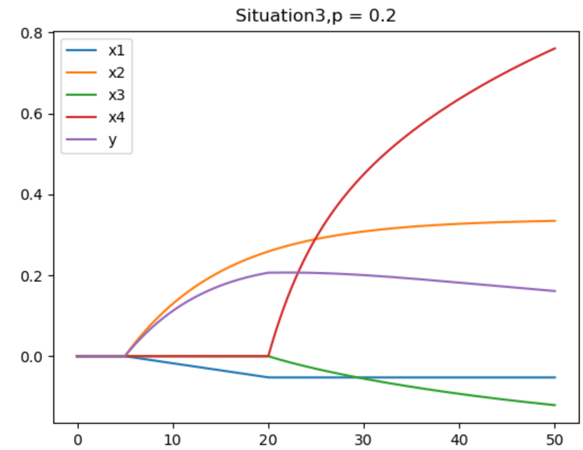


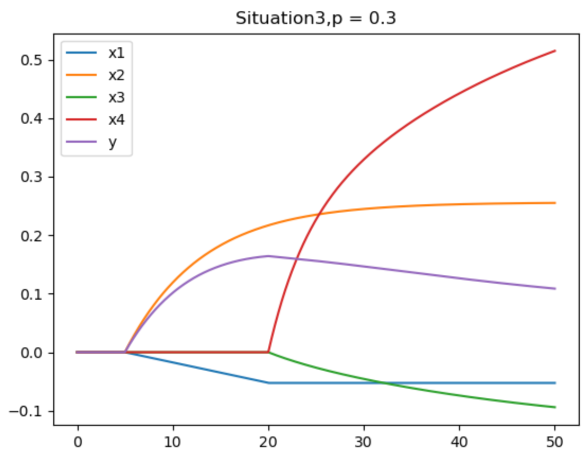


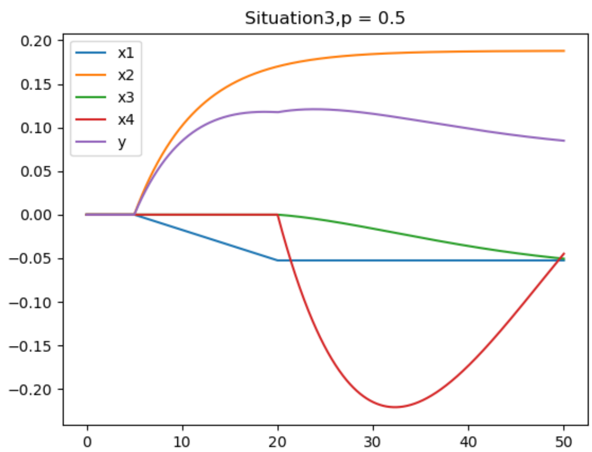
情况三：

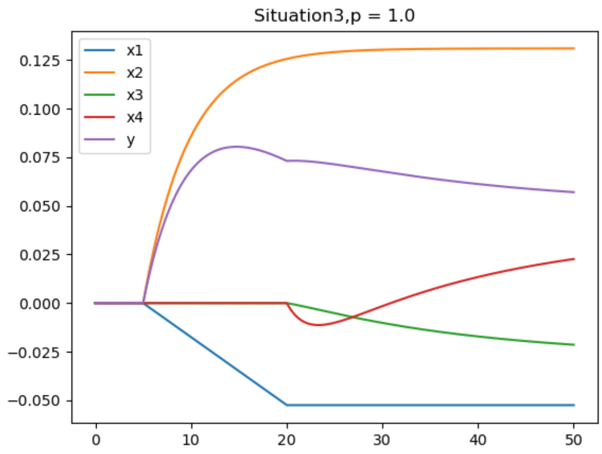
$x_1, x_2, x_3, x_4, y(t)$的解绘制的曲线为：







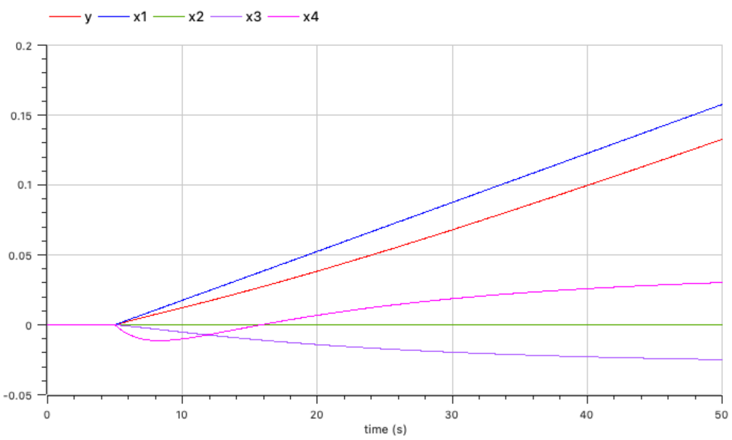


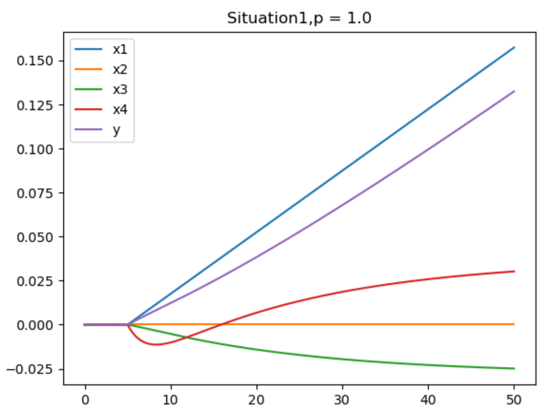


基于Modelica的部分验证

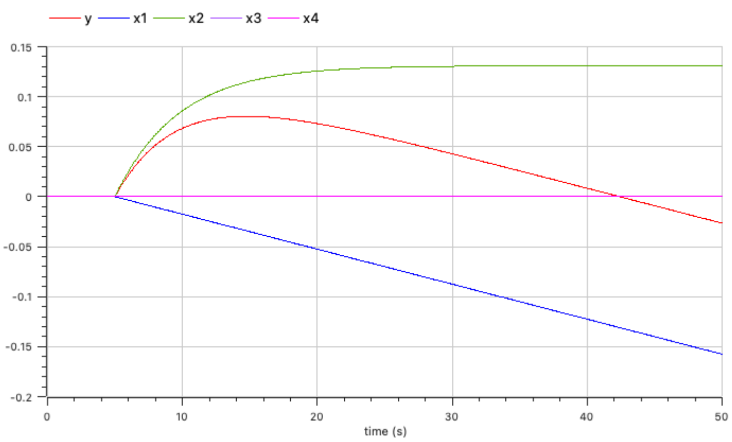
Modelica是一个面向物理方程的建模语言，长期实践下发现其非常适合求解ODE问题。此题即为一类非常适合Modelica求解的类型。Modelica提供了多种线性/非线性微分方程组求解器，如dassl，cvode等，也具备传统的RK、euler等求解器。为了保证求解的正确性，我们一并采用与python算法相同的设置。我们仅验证满功率运行下三种情况的解。Modlica的完整代码将在附录里介绍。

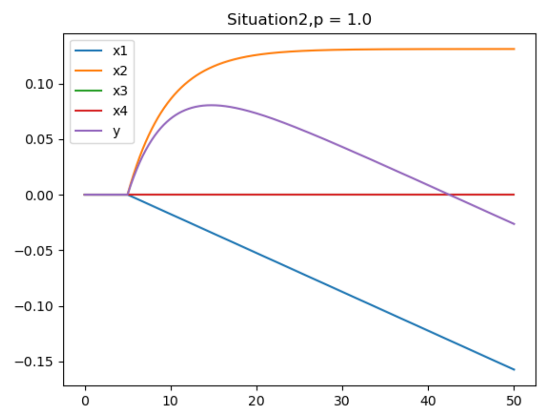
情况一，p=1.0



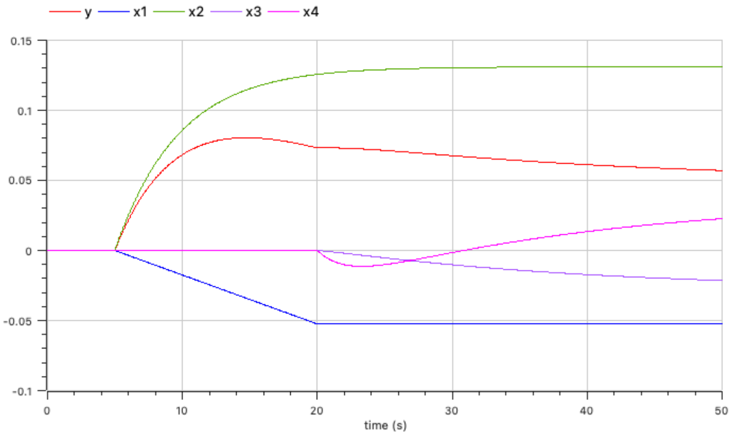


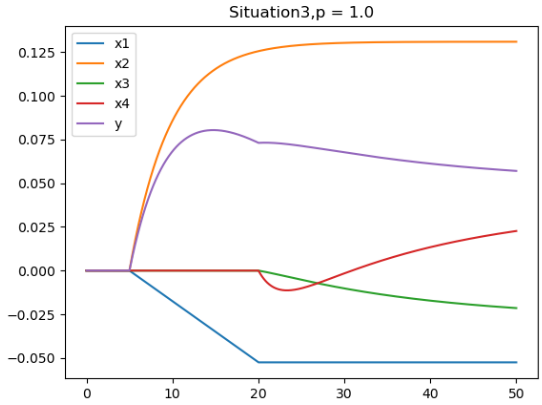
情况二，p=1.0





情况三，p=1.0





不难发现，自编写的求解器具备与Modelica相似的准确度。

Python算法实现

通过`if __name__ == "__main__":`创建算法主分支，因为可以将其余部分进行cython优化运行。

```
import numpy as np
import matplotlib.pyplot as plt
import ODE
import plotly.graph_objects as go

'''
SG水位控制系统微分方程为:
x1' = G1 (qe - qv)
x2' = -x2/tau0 + G2 * qv/ tau0
x3' = x4/tau**2 - G3 * beta * qe / tau
x4' = -x3 - 2*xi*x4/tau + (2*xi*beta - 1)*G3 * qe
y(t) = x1 + x2 + x3
在各个功率点分别求解
'''
#表格内容
class Parameter():
    #功率水平
    p = np.array([0.1,0.2,0.3,0.5,1]) #10%,...100%
    G1 = np.array([0.0031,0.0035,0.0035,0.0035,0.0035])
    G2 = np.array([0.402,0.339,0.256,0.188,0.131])
    G3 = np.array([0.166,0.207,0.143,0.055,0.028])
    tau_0 = np.array([10,10.4,8.0,6.4,4.7])
    tau = np.array([19.7,12.5,10.3,13.3,6.6])
    xi = np.array([0.65,1.6,1.6,0.62,1.68])
    beta = np.array([-0.08,0.44,0.47,0.20,0.20])

#构建微分方程程及方程组
class Functions():
    def __init__(self,G1,G2,G3,tau_0,tau,beta,xi,qe,qv):
```

```python
        self.G1 = G1
        self.G2 = G2
        self.G3 = G3
        self.tau_0 = tau_0
        self.tau = tau
        self.beta = beta
        self.xi = xi
        self.qe = qe
        self.qv = qv
    def f1(self,t,x):
        return self.G1*(self.qe(t) - self.qv(t))
    def f2(self,t,x):
        x2 = x[1]
        return -x2/self.tau_0 + self.G2 * self.qv(t)/self.tau_0
    def f3(self,t,x):
        x4 = x[3]
        return x4/self.tau**2 - self.G3*self.beta*self.qe(t)/self.tau
    def f4(self,t,x):
        x3 = x[2]
        x4 = x[3]
        return -x3-2*self.xi*x4/self.tau+(2*self.xi*self.beta-1)*self.G3*self.qe(t)

#设置初始值
x0 = np.array([0,0,0,0])
step = 1000

def Situation1():
    '''
    情况一:
    qv(t) = 0 , 0 <= t <= 50
            | 0 , 0<= t < 5
    qe(t) = {
            | 1 , 5<= t <= 50
    '''
    qv = lambda t: 0
    qe = lambda t: 1 if t>=5 else 0
    for i in range(0,5):
        Func = Functions(Parameter.G1[i],Parameter.G2[i],Parameter.G3[i]
                        ,Parameter.tau_0[i],Parameter.tau[i],Parameter.beta[i]
                        ,Parameter.xi[i],qe,qv)
        F = lambda t,x: np.array([Func.f1(t,x),Func.f2(t,x),Func.f3(t,x),Func.f4(t,x)])
        solver1 = ODE.RK4_for_equations(F,4,0,50,step,x0).slover()
        y = solver1[0,:] + solver1[1,:] + solver1[2,:]
        print('p = '+str(Parameter.p[i]))
        print(solver1)

        #绘图
        t = np.linspace(0,50,step)
        plt.plot(t,solver1[0,:],label='x1')
```

```
        plt.plot(t,solver1[1,:],label='x2')
        plt.plot(t,solver1[2,:],label='x3')
        plt.plot(t,solver1[3,:],label='x4')
        plt.plot(t,y,label='y')
        plt.title('Situation1,p = '+str(Parameter.p[i]))
        plt.legend()
        plt.show()
        '''
        #用plotly绘图
        fig = go.Figure()
        fig.add_trace(go.Scatter(x=t,y=solver1[0,:],name='x1'))
        fig.add_trace(go.Scatter(x=t,y=solver1[1,:],name='x2'))
        fig.add_trace(go.Scatter(x=t,y=solver1[2,:],name='x3'))
        fig.add_trace(go.Scatter(x=t,y=solver1[3,:],name='x4'))
        fig.show()
        '''
def Situation2():
    '''
    情况二:
    qe(t) = 0 , 0 <= t <= 50

            | 0 , 0<= t < 5
    qv(t) = {
            | 1 , 5<= t <= 50

    '''
    qe = lambda t: 0
    qv = lambda t: 1 if t>=5 else 0
    for i in range(0,5):
        Func = Functions(Parameter.G1[i],Parameter.G2[i],Parameter.G3[i]
                        ,Parameter.tau_0[i],Parameter.tau[i],Parameter.beta[i]
                        ,Parameter.xi[i],qe,qv)
        F = lambda t,x: np.array([Func.f1(t,x),Func.f2(t,x),Func.f3(t,x),Func.f4(t,x)])
        solver1 = ODE.RK4_for_equations(F,4,0,50,step,x0).slover()
        y = solver1[0,:] + solver1[1,:] + solver1[2,:]
        print('p = '+str(Parameter.p[i]))
        print(solver1)

        #绘图
        t = np.linspace(0,50,step)
        plt.plot(t,solver1[0,:],label='x1')
        plt.plot(t,solver1[1,:],label='x2')
        plt.plot(t,solver1[2,:],label='x3')
        plt.plot(t,solver1[3,:],label='x4')
        plt.plot(t,y,label='y')
        plt.legend()
        plt.title('Situation2,p = '+str(Parameter.p[i]))
        plt.show()
def Situation3():
```

```
    '''
    情况三:
            | 0 , 0<= t < 20
    qe(t) = {
            | 1 , 20<= t <= 50

            | 0 , 0<= t < 5
    qv(t) = {
            | 1 , 5<= t <= 50

    '''
    qe = lambda t: 1 if t>=20 else 0
    qv = lambda t: 1 if t>=5 else 0
    for i in range(0,5):
        Func = Functions(Parameter.G1[i],Parameter.G2[i],Parameter.G3[i]
                        ,Parameter.tau_0[i],Parameter.tau[i],Parameter.beta[i]
                        ,Parameter.xi[i],qe,qv)
        F = lambda t,x: np.array([Func.f1(t,x),Func.f2(t,x),Func.f3(t,x),Func.f4(t,x)])
        solver1 = ODE.RK4_for_equations(F,4,0,50,step,x0).slover()
        y = solver1[0,:] + solver1[1,:] + solver1[2,:]
        print('p = '+str(Parameter.p[i]))
        print(solver1)

        #绘图
        t = np.linspace(0,50,step)
        plt.plot(t,solver1[0,:],label='x1')
        plt.plot(t,solver1[1,:],label='x2')
        plt.plot(t,solver1[2,:],label='x3')
        plt.plot(t,solver1[3,:],label='x4')
        plt.plot(t,y,label='y')
        plt.legend()
        plt.title('Situation3,p = '+str(Parameter.p[i]))
        plt.show()


if __name__ == "__main__":
    #Situation1()
    #Situation2()
    Situation3()
```

附录

MatrixSolverLU.py

```
import numpy as np
def MartrixSolver(A, d):
    '通过LU分解求解线性方程组'
    n = len(A)
```

```
    U = np.zeros((n, n))
    L = np.zeros((n, n))

    for i in range(0, n):
        U[0, i] = A[0, i]
        L[i, i] = 1
        if i > 0:
            L[i, 0] = A[i, 0] / U[0, 0]
    # LU分解
    for r in range(1, n):
        for i in range(r, n):
            sum1 = 0
            sum2 = 0
            ii = i + 1
            for k in range(0, r):
                sum1 = sum1 + L[r, k] * U[k, i]
                if ii < n and r != n - 1:
                    sum2 = sum2 + L[ii, k] * U[k, r]
            U[r, i] = A[r, i] - sum1
            if ii < n and r != n - 1:
                L[ii, r] = (A[ii, r] - sum2) / U[r, r]
    # 求解y
    y = np.zeros(n)
    y[0] = d[0]

    for i in range(1, n):
        sumy = 0
        for k in range(0, i):
            sumy = sumy + L[i, k] * y[k]
        y[i] = d[i] - sumy
    # 求解x
    x = np.zeros(n)
    x[n - 1] = y[n - 1] / U[n - 1, n - 1]
    for i in range(n - 2, -1, -1):
        sumx = 0
        for k in range(i + 1, n):
            sumx = sumx + U[i, k] * x[k]
        x[i] = (y[i] - sumx) / U[i, i]

    return x
```

FitSquares.py

```
import numpy as np
import matplotlib.pyplot as plt
import time
class FitSquares_polynomial:
    def __init__(self,arr1,n):
```

```python
        self.arr1 = arr1
        self.arr1_x = arr1[:,0]
        self.arr1_y = arr1[:,1]
        self.lenth = len(arr1)
        self.n = n
        self.an = self.phiprod()[0]

    def phiprod(self):
        #确定总长度
        n = self.n
        #初始化G,d向量
        G = np.array([])
        d = np.array([])
        #计算并生成G,d向量
        for i in range(0,n):
            d = np.append(d,np.sum((self.arr1_y)*(self.arr1_x**i)))
            for j in range(0,n):
                #这里的G向量是有n个元素的行向量
                G = np.append(G,np.sum((self.arr1_x**i)*(self.arr1_x**j)))
        #通过.reshape方法将G向量转为n阶方阵
        G = G.reshape(n,n)
        #通过np求逆求解，待更新轮子解法
        #an = np.dot(np.linalg.inv(G), d)
        #通过自制LU求解器
        an = self.MartrixSolver(G,d)
        return an,G,d

    def num(self,x):
        num = 0
        for i in range(0,self.n):
            num = num+(self.an[i])*(x**i)
        return num

    def visualize(self,start,end,step,text):
        x = np.linspace(start,end,step)
        y = np.zeros(1)
        for i in x:
            y = np.append(y,self.num(i))
        y = y[1:]
        plt.figure()
        plt.scatter(self.arr1_x, self.arr1_y, c='red')
        if text is True:
            for j in range(0,self.lenth):
                plt.text(self.arr1_x[j],self.arr1_y[j],(self.arr1_x[j],self.arr1_y[j]))
        plt.plot(x,y)
        plt.show()

    def delta(self):
        de = np.zeros(self.lenth)
```

```python
        for i in range(0,self.lenth):
            de[i] = (self.num(self.arr1_x[i])-self.arr1_y[i])**2
        return np.min(de)

    #LU分解
    def MartrixSolver(self,A, d):
        n = len(A)
        U = np.zeros((n, n))
        L = np.zeros((n, n))

        for i in range(0, n):
            U[0, i] = A[0, i]
            L[i, i] = 1
            if i > 0:
                L[i, 0] = A[i, 0] / U[0, 0]
        # LU分解
        for r in range(1, n):
            for i in range(r, n):
                sum1 = 0
                sum2 = 0
                ii = i + 1
                for k in range(0, r):
                    sum1 = sum1 + L[r, k] * U[k, i]
                    if ii < n and r != n - 1:
                        sum2 = sum2 + L[ii, k] * U[k, r]
                U[r, i] = A[r, i] - sum1
                if ii < n and r != n - 1:
                    L[ii, r] = (A[ii, r] - sum2) / U[r, r]
        # 求解y
        y = np.zeros(n)
        y[0] = d[0]

        for i in range(1, n):
            sumy = 0
            for k in range(0, i):
                sumy = sumy + L[i, k] * y[k]
            y[i] = d[i] - sumy
        # 求解x
        x = np.zeros(n)
        x[n - 1] = y[n - 1] / U[n - 1, n - 1]
        for i in range(n - 2, -1, -1):
            sumx = 0
            for k in range(i + 1, n):
                sumx = sumx + U[i, k] * x[k]
            x[i] = (y[i] - sumx) / U[i, i]

        return x
```

LinearInterpolation.py

```
def LinearInterpolation(x,arr1):
    '线性插值'
    n = len(arr1)
    for i in range(0, n):
        if arr1[i, 0] == x:
            return arr1[i, 1]
        elif arr1[i, 0] > x:
            return arr1[i - 1, 1] + (x - arr1[i - 1, 0]) * (arr1[i, 1] - arr1[i - 1,
1]) / (arr1[i, 0] - arr1[i - 1, 0])
```

CubicSplineFree.py

```
import numpy as np
import matplotlib.pyplot as plt
class CubicSplineFree:
    def __init__(self,arr1):
        self.arr1 = arr1
        self.arr1_x = arr1[:,0]
        self.arr1_y = arr1[:,1]
        self.lenth = len(arr1)
    #hn为x之间的间隔
    def hn(self):
        hnn = np.array([])
        for i in range(0,self.lenth-1):
            hnn =np.append(hnn,self.arr1_x[i+1]-self.arr1_x[i])
        return hnn

    def mu(self):
        mu = np.zeros(1)
        hn = self.hn()
        for i in range(1,len(hn)):
            mu = np.append(mu,hn[i-1]/(hn[i-1]+hn[i]))
        return mu

    def lam(self):
        lam = np.zeros(1)
        hn = self.hn()
        for i in range(1,len(hn)):
            lam = np.append(lam,hn[i]/(hn[i-1]+hn[i]))
        return lam
    #fm为余项，定义与牛顿插值相同
    def fm(self,i):
        return (self.arr1_y[i]-self.arr1_y[i+1])/(self.arr1_x[i]-self.arr1_x[i+1])\
              -(self.arr1_y[i]-self.arr1_y[i-1])/(self.arr1_x[i]-self.arr1_x[i-1])

    def dn(self):
```

```
        dn = np.zeros(1)
        hn = self.hn()
        for i in range(1,len(hn)):
            dn = np.append(dn,6*self.fm(i)/(hn[i-1]+hn[i]))
        return dn

    def TDMA(self,a, b, c, d):
        try:
            n = len(d)  #确定长度以生成矩阵
            # 通过输入的三对角向量a,b,c以生成矩阵A
            A = np.array([[0] * n] * n, dtype='float64')
            for i in range(n):
                A[i, i] = b[i]
                if i > 0:
                    A[i, i - 1] = a[i]
                if i < n - 1:
                    A[i, i + 1] = c[i]
            # 初始化代计算矩阵
            c_1 = np.array([0] * n)
            d_1 = np.array([0] * n)
            for i in range(n):
                if not i:
                    c_1[i] = c[i] / b[i]
                    d_1[i] = d[i] / b[i]
                else:
                    c_1[i] = c[i] / (b[i] - c_1[i - 1] * a[i])
                    d_1[i] = (d[i] - d_1[i - 1] * a[i]) / (b[i] - c_1[i - 1] * a[i])
            # x: Ax=d的解
            x = np.array([0] * n)
            for i in range(n - 1, -1, -1):
                if i == n - 1:
                    x[i] = d_1[i]
                else:
                    x[i] = d_1[i] - c_1[i] * x[i + 1]
            #x = np.array([round(_, 4) for _ in x])
            return x
        except Exception as e:
            return e

    def Mn(self):
        a = np.append(self.mu(),0)
        c = np.append(self.lam(),0)
        b = 2*np.ones(self.lenth)
        d = np.append(self.dn(),0)
        Mn = self.TDMA(a,b,c,d)
        return Mn

    def zone(self,x):
        if x < np.min(self.arr1_x): zone = 0
```

```python
        if x > np.max(self.arr1_x): zone = self.lenth-2
        for i in range(0,self.lenth-1):
            if x-self.arr1_x[i]>=0 and x-self.arr1_x[i+1]<=0:
                zone = i
        return zone

    def num(self,x):
        j = self.zone(x) #zone函数的作用为确定输入量x处于的区间
        M = self.Mn()
        h = self.hn()
        S = M[j]*((self.arr1_x[j+1]-x)**3)/(6*h[j]) \
            + M[j+1]*((x-self.arr1_x[j])**3)/(6*h[j]) \
            + (self.arr1_y[j]-(M[j]*(h[j]**2))/6)*(self.arr1_x[j+1]-x)/h[j] \
            + (self.arr1_y[j+1]-M[j+1]*h[j]**2/6)*(x-self.arr1_x[j])/h[j]
        return S

    def visualize(self,start,end,step,text):
        x = np.linspace(start,end,step)
        y = np.zeros(1)
        for i in x:
            y = np.append(y,self.num(i))
        y = y[1:]
        plt.figure()
        plt.scatter(self.arr1_x, self.arr1_y, c='red')
        if text is True:
            for j in range(0,self.lenth):
                plt.text(self.arr1_x[j],self.arr1_y[j],(self.arr1_x[j],self.arr1_y[j]))
        plt.plot(x,y)
        plt.show()
```

ODE.py

```python
import numpy as np
class RK4_for_equations:
    def __init__(self, F,n,min,max,totstep,begin):
        self.F = F
        self.n = n #指定方程组的个数
        self.min = min
        self.max = max
        self.step = (self.max-self.min)/totstep
        self.begin = begin
        self.totstep = totstep

    def slover(self):
        f = self.F
        x = 0
        y = self.begin
        yn = np.zeros((self.n,1))
```

```
        yn[:,0] = y
        step = self.step
        flag = 1
        for i in range(1,self.totstep):
            K1 = f(x,y)
            #print("K1:",K1)
            K2 = f(x+0.5*step,y+0.5*step*K1)
            #print("K2:",K2)
            K3 = f(x+0.5*step,y+0.5*step*K2)
            #print("K3:",K3)
            K4 = f(x+step,y+step*K3)
            #print("K4:",K4)
            y = y + (step/6)*(K1+2*K2+2*K3+K4)
            yn = np.append(yn,y.reshape(self.n,1),axis=1)
            flag = flag + 1
            x = i * step
        return yn
```

Homework.mo

```
model Homework
parameter Real G1 =   0.0035;
parameter Real G2 =   0.131;
parameter Real G3 =   0.028;
parameter Real tau0 = 4.7;
parameter Real tau =  6.6;
parameter Real xi =   1.68;
parameter Real beta = 0.2;
Real qe;
Real qv;
Real x1;
Real x2;
Real x3;
Real x4;
Real y;

function qvt
  input Real t;
  output Real qv;
  algorithm
    if t < 5 then
    qv := 0;
  else
    qv:= 1;
  end if;
end qvt;

function qet
```

```
  input Real t;
  output Real qe;
  algorithm
    if t < 20 then
    qe := 0;
  else
    qe:= 1;
  end if;
end qet;

initial equation
x1 = 0;
x2 = 0;
x3 = 0;
x4 = 0;

equation
qv = qvt(time);
qe = qet(time);
der(x1) = G1 * (qe-qv);
der(x2) = -x2/tau0 + G2*qv/tau0;
der(x3) = x4/tau^2 - G3*beta*qe/tau;
der(x4) = -x3 - 2*xi*x4/tau + (2*xi*beta - 1)*G3*qe;
y = x1 + x2 + x3;
end Homework;
```