题目一 稀疏线性方程组的求解

一、题目内容

利用课堂上介绍的稀疏线性方程组求解方法编写通用程序，求解下述方程组。

$Ax = b$

其中：

$$A = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 2 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 2 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \end{pmatrix}$$

$$b = (4 \quad 4 \quad 4 \quad 4 \quad 4 \quad 4 \quad 4 \quad 4 \quad 5 \quad 5 \quad 5 \quad 5)^T$$

要求：

使用 MATLAB 或 Python 语言编写；

编写通用程序，原始数据通过参数传入程序中，不能把数据硬性写入代码中；

分为四个步骤，分别为：①因子表分解，②前代，③规格化，④回代；

要求采用稀疏计算方法，尽可能避免对零元素进行操作；

对于只需求解某个特定变量的情况能够处理，例如对于只需求解 x_3 的情况可以采用稀疏向量方法进一步减少计算量（注意，具体求解哪个变量也不能硬性规定，应该允许程序使用者动态指定）。

二、方案论证

1. 稀疏矢量和稀疏矩阵的储存

由于稀疏矩阵中非零元素较少，零元素较多，因此可以采用只存储非零元素的方法来进行压缩存储。对于一个用二维数组存储的稀疏矩阵 Amn，如果假设存储每个数

组元素需要 L 个字节，那么存储整个矩阵需要 m*n*L 个字节。但是，这些存储空间的大部分存放的是 0 元素，从而造成大量的空间浪费。为了节省存储空间，可以只存储其中的非 0 元素。大大减少了空间的存储。本文中采用 scipy 库的稀疏矩阵库 sparse 对稀疏矩阵进行读取信息和修改元素。

Scipy.sparse 中稀疏矩阵有多种储存格式，其中 coo_matrix() 是一种坐标形式的稀疏矩阵。采用三个数组 row、col 和 data 保存非零元素的信息，这三个数组的长度相同，row 保存元素的行，col 保存元素的列，data 保存元素的值，COO 的主要优点是灵活、简单，仅存储非零元素以及每个非零元素的坐标。但是 COO 不支持元素的存取和增删，一旦创建之后，除了将之转换成其它格式的矩阵，几乎无法对其做任何操作和矩阵运算。而 dok_matrix() 则是基于键值对的字典稀疏矩阵。本文中采用 coo_matrix()快速提取矩阵中的数据，包括行、列非零元的位置，采用 dok_matrix()对矩阵中的数据进行修改。

2. 稀疏矩阵的因子分解

对于 nxn 阶的矩阵 A 可以通过 LU 分解的方法将它分解成一个下三角矩阵和一个单位上三角矩阵的乘积，即 A = LU。LU 分解可以分成两步：(1) 按行规格化 (2) 消去运算或更新运算，若采用稀疏储存格式，则只取非零元 U(k),L(k) 进行计算，省去了非稀疏储存格式中对零元的大量判断，计算流程如下：

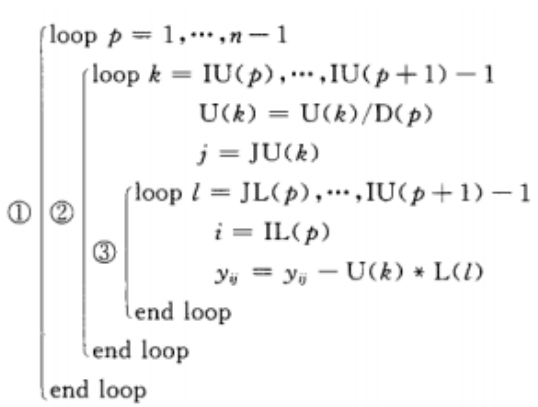


图 1 采用稀疏储存格式的计算流程

本文中采用另一种因子分解方式，即 LDU 分解，将 A 分解为单位下三角矩阵 L、对角线矩阵 D 和单位上三角矩阵 U 的乘积形式。

$$A = L\,D\,U$$

和 LU 分解相比，两者的 U 矩阵相同，D 矩阵是由 LU 分解中的矩阵 L 的对角线元素组成，这里的单位下三角矩阵和 LU 分解中的下三角矩阵不同，对角元素为 1，非对

角线元素都除以相应列的对角线元素，即按列规格化，当 A 对称时，LU 互为转置。

3. 稀疏矩阵技术的图论描述

几个定义如下：

A 图，是和矩阵 A 有相同拓扑结构的网络图

有向 A 图，对给定的 A 图及节点编号，规定每条边的正方向都是由小号节点指向大号节点，由此形成的有向图。

赋权有向 A 图，是在有向 A 图上，将 A 的非对角非零元所对应的边称为互边，并将该边的权赋之以该非零元的值。将 A 的对角元素用在有向 A 图的接地边模拟，称之为自边，并赋之以该对角元素的值，这样得到的有向 A 图称为赋权有向 A 图。

可以以同样的方式以因子图、有向因子图、赋权有向因子图来描述因子表 U。

4. 因子分解的图论描述

因子分解过程是按节点号顺序由小到大依次进行的，每步计算中要消去下三角部分的非零非对角元，包括规格化运算和消去运算两步。

(1) 规格化运算

在赋权有向 A 图中，相当于对节点 p 发出的所有互边的边权加以修正，新的边权等于原边权除以节点 p 的自边边权，不改变互边的数目。

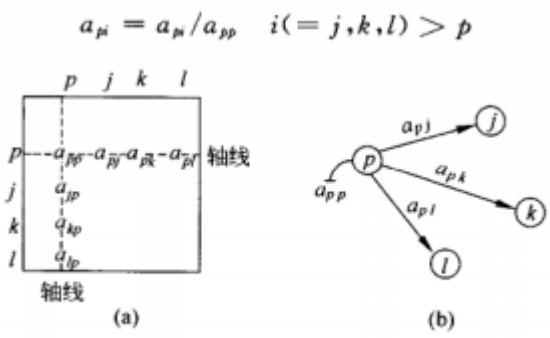


图 3.3 对第 p 行元素的规格化运算

图 2 对 p 行元素规格化运算的图论描述

(2) 消去运算

实际上是对于处于轴线非零元所在的行列上的元素进行消去运算，步骤如下。

a. 对于节点 p 发出的互边的收点，将该点上的自边边权减去该互边边权的平方乘以节点 p 上的自边边权

b. 对于节点 p 发出的所有互边，这些互边两两之间的所夹的互边边权应该减去两条相夹边之间的边权与节点 p 的自边边权三者乘积，若被夹节点无边时，应该视为有一条零边权值。

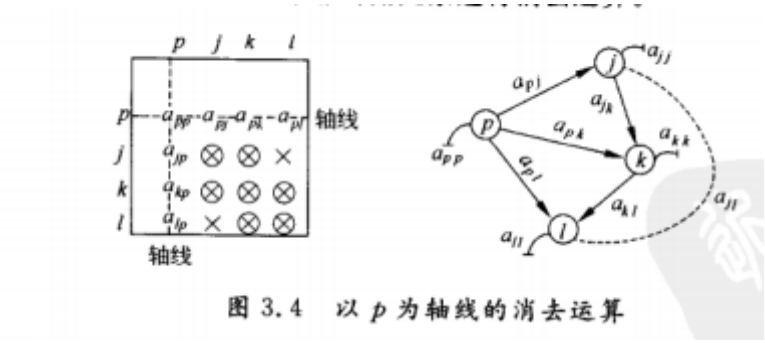


图 3.4 以 p 为轴线的消去运算

图 3 以 p 为轴线的消去运算的图论描述

5. 利用稀疏矩阵因子表求解稀疏线性代数方程组

对于 n 维线性代数方程组 Ax=b，如果系数矩阵 A 已经被分解为因子表，即 A=LDU 的形式，则可以通过引入中间矢量 y，z 来求解 x

$$Lz = b$$
$$Dy = z$$
$$Ux = y$$

其中求解 z 的过程是前代，求解 x 的过程是回代。

6. 前代回代过程计算流程的图论描述

线性代数方程组中独立矢量或解矢量的非零元可用赋权有向因子图上的节点的点位来描述，计算流程如下

(1)将独立矢量 b 的非零元赋值为赋权有向因子图上的点位

(2)扫描 i 从 1 到 n-1，修正节点 i 发出的边的收端节点 j 的点位

(3)对所有节点对点位规格化

(4)扫描 j 从 n 到 2 所有指向节点 j 的边的发端节点 i 修正点位。

在回代过程中，某点 i 的点位只受到该点发出的边的收点 j 的点位有向，这些收点 j 的点位又受到它们各自发出的边的收点 j'的点位影响。因此，如果要解矢量 x 中的特定元素，只需要从该节点沿图上箭头方向搜索直到根节点 n，就可以找到影响该节点的所有点位，只对这些节点进行回代即可。由于时间限制，本文中未涉及利用稀疏矢量法求解某个具体变量的部分。

本文中采用 python 为编程语言，PyCharm 为 IDE 进行开发。

三、过程论述

1. 理论思路

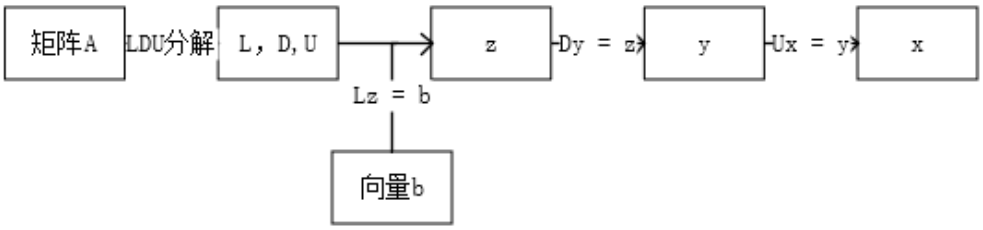


图 4 代码思路

2. 代码撰写说明

(1) my_slove(A,b)

传入参数 A：系数矩阵，b：独立矢量

返回值 x：解矢量

稀疏线性方程组求解方法, 分为四个步骤，分别为：①因子表分解，②前代，③规格化，④回代；代码见附录

(2) myLDU(A)

传入参数 A：待分解的矩阵，

返回值 L,D,U

撰写思路：

a. 首先获取 A 的上三角矩阵的三元组储存格式，方便遍历。

b. 按行遍历，获取该列中所有非对角非零元坐标，即该行对应的节点 p 发出的互边。使用该节点的对角元即自边边权对该互边进行修正。

c. 获取修改后 A 矩阵的对角矩阵 D，获取 A 的对角元为 1 的上三角矩阵 U。 获得 U 转置后的下三角矩阵 L.

(3) my_forward_substitution(b,L)

传入参数 b: 独立矢量 L 前乘矢量集

返回值 z: 前代后结果

撰写思路：

a. 首先获取 A 的上三角矩阵的三元组储存格式，方便遍历。

b. 在赋权有向因子图上，将每个节点的点位赋以独立矢量 b 中相应非零元的值

c. 按由小到大的顺序对列遍历，获取该列中所有非对角非零元坐标，即在赋权有向因子图上，该列对应的节点 p 发出的收端节点，对该收端节点进行修正。

d. 所有节点扫描完后，因子图上的点位就是前代后的结果，即返回值 z

(3) my_norm_substitution(z,D)

传入参数 z: 前代结果 D 对角矩阵

返回值 y: 规格化结果

撰写思路：

a. 将前代结束后节点 i 的点位 ei 除以赋权有向因子图上节点 i 的自边边权即可。

(4) my_backward_substitution(y,U)

传入参数 y 规格化结果 L 回代矢量集

返回值 x: 回代后结果即最终解

撰写思路：

a. 首先获取 A 的上三角矩阵的三元组储存格式，方便遍历。

b. 在赋权有向因子图上，将每个节点的点位赋以独立矢量 y 中相应非零元的值

c. 按由大到小的顺序对列遍历，获取该列中所有非对角非零元坐标，即在赋权有向因子图上，该列对应的指向节点 p 的发端节点，对该发端节点的点位进行修正。也可以看作将赋权有向因子图所有边反向，然后按前代计算过程一样按箭头方向去修正收点点位。

d. 所有节点扫描完后，因子图上的点位就是回代后的结果，即最终解 x

四、结果分析

使用 numpy 的 lingal 线性代数模块进行验证，与本文中的代码结果进行比较，运行比较结果如下。

```
if __name__ == '__main__':
    A = np.array([[2,0,0,0,0,0,1,0,0,0,0,1],
                  [0,2,0,0,1,0,0,0,1,0,0,0],
                  [0,0,2,0,0,1,0,0,1,0,0,0],
                  [0,0,0,2,0,0,1,0,0,1,0,0],
                  [0,1,0,0,2,0,0,0,0,0,0,1],
                  [0,0,1,0,0,2,0,0,0,1,0,0],
                  [1,0,0,1,0,0,2,0,0,0,0,0],
                  [0,0,0,0,0,0,0,2,1,0,1,0],
                  [0,1,1,0,0,0,0,1,2,0,0,0],
                  [0,0,0,1,0,1,0,0,0,2,1,0],
                  [0,0,0,0,0,0,0,1,0,1,2,1],
                  [1,0,0,0,1,0,0,0,0,0,1,2]],dtype=np.float)
    b = np.array([4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5],dtype=float)
    x1 = my_slove(A,b)
    print('x1:',x1)
    print('x2:',np.linalg.solve(A,b))
    print('x3:', np.matmul(np.linalg.inv(A),b))
```

图 5 代码验证部分

```
my_LDU
C:\Users\lenovo\AppData\Local\Programs\Python\Python38\
x1: [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
x2: [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
x3: [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

图 6 代码验证结果

可以看出本稀疏线性方程组计算结果与 numpy 中 linalg 的解一致。

五、代码附录

```
import numpy as np
from    scipy.sparse import coo_matrix,dok_matrix
from itertools import combinations_with_replacement
import pandas as pd
```

```
def myLDU(A):
    # 首先获取A 的上三角三元组格式的
    A = np.triu(A)

    A = coo_matrix(A)
    rowInd = A.row
    # 获得矩阵的行数
    node_list = list(set(rowInd))
    for p in node_list[:-1]:
        A = coo_matrix(A,dtype=np.float)
        rowInd = A.row
        colInd = A.col
        A = dok_matrix(A,dtype=np.float)
        # 获取该行中所有的非零元列坐标
        col_index_list = []
        for row_index,i in enumerate(rowInd) :
            if i == p:
                # 根据行的索引找到对应的列
                col_index_list.append(colInd[row_index])
        # 进行规格化计算,由于已经取了对角元，对角元需要保留
        # 对节点 p 发出的所有互边的边权进行修正
        for j in (col_index_list[1:]):
            # 根据列索引找到对应的元素
            A[p,j] =    A[p,j]/ A[p,p]
        # 消去运算
        # 该节点发出的所有边的排列组合
        col_index_list.sort()
        i_j_list = list(combinations_with_replacement(col_index_list[1:], 2))
        for i_j in (i_j_list):
            i = i_j[0]
            j = i_j[1]
            A[i,j] = A[i,j] - A[p,i]*A[p,j]*A[p,p]
    D =np.diag(A.diagonal())
    dim = (A.shape[0])
    U = np.eye(dim,dtype=np.float)
    U += A-D
    L = U.T

    return L,D,U
# 引入中间矢量y 和z，Lz =b,Dy = z,Ux = y
def my_forward_substitution(b,L):
```

```
    L = coo_matrix(L,dtype=np.float)
    rowInd = L.row
    colInd = L.col
    # 获取需要的数据后把A 变为可以修改的矩阵
    L = dok_matrix(L,dtype=float)
    # 获得矩阵的行数
    node_list = list(set(rowInd))
    node_list.sort()
    z = b
    z = z.astype(float)

    for j in    node_list[:-1]:
        if z[j]!=0:
            # 获取要被消去的行的非零元的列坐标
            row_index_list = []
            for col_index,p in enumerate(colInd) :
                if p == j:
                    # 根据列的索引找到对应的行
                    row_index_list.append(rowInd[col_index])
            row_index_list.sort()
            for    i in (row_index_list[1:]):
                # 根据列索引找到对应的元素
                    z[i] = z[i] - L[i,j]*z[j]
    return z

# 规则化运算
def my_norm_substitution(z,D):
    y = z/D.diagonal()
    y = y.astype(float)
    return y
def my_backward_substitution(y,U):
    U = coo_matrix(U,dtype=np.float)
    rowInd = U.row
    # 获取需要的数据后把A 变为可以修改的矩阵
    U = dok_matrix(U,dtype=np.float)
    # 获得矩阵的行数
    node_list = list(set(rowInd))
    node_list.sort(reverse=True)
    x = y
    x = x.astype(float)
    for j in node_list[:-1]:
        if x[j] != 0:
```

```
                U = coo_matrix(U)
                rowInd = U.row
                colInd = U.col
                U = dok_matrix(U)
                # 扫描该列非零元的行号
                row_index_list = []
                for col_index, p in enumerate(colInd):
                    if p == j:
                        # 根据行的索引找到对应的行
                        row_index_list.append(rowInd[col_index])
                row_index_list.sort(reverse=True)

                for i in (row_index_list[1:]):
                    # 根据列索引找到对应的元素
                    x[i] = x[i] - U[i, j] * x[j]

        return x

def my_slove(A,b):
        L,D,U = myLDU(A)
        z = my_forward_substitution(b, L)
        y = my_norm_substitution(z,D)
        x = my_backward_substitution(y, U)
        return x
if __name__ == '__main__':
        A = np.array([[2,0,0,0,0,0,1,0,0,0,0,1],
                      [0,2,0,0,1,0,0,0,1,0,0,0],
                      [0,0,2,0,0,1,0,0,1,0,0,0],
                      [0,0,0,2,0,0,1,0,0,1,0,0],
                      [0,1,0,0,2,0,0,0,0,0,0,1],
                      [0,0,1,0,0,2,0,0,0,1,0,0],
                      [1,0,0,1,0,0,2,0,0,0,0,0],
                      [0,0,0,0,0,0,0,2,1,0,1,0],
                      [0,1,1,0,0,0,0,1,2,0,0,0],
                      [0,0,0,1,0,1,0,0,0,2,1,0],
                      [0,0,0,0,0,0,0,0,1,0,1,2,1],
                      [1,0,0,0,1,0,0,0,0,0,1,2]],dtype=np.float)
        b = np.array([4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5],dtype=float)
        x1 = my_slove(A,b)
        print('x1:',x1)
        print('x2:',np.linalg.solve(A,b))
        print('x3:', np.matmul(np.linalg.inv(A),b))
```

题目二 微分方程求解方法

一、题目内容

利用课堂上介绍的微分方程求解方法编写程序求解下述问题：

$$\frac{dx}{dt} = -20x$$

要求：

使用 MATLAB 或 Python 语言编写；

分别编写①欧拉法、②改进欧拉法的计算程序；

步长分别取 $\Delta t = 1/n$，其中 $n = 100, 20, 15, 10$；

自行学习图形化输出方法，将仿真计算结果输出成图像；

程序完成后撰写编写思路的文档，课程结束后和代码一起提交作为课程分数依据。

编写隐式梯形法计算程序求解这一微分方程，步长选择与上面相同。

二、方案论证

1. 前向欧拉法及基本概念

对于微分方程

$$\frac{\mathrm{d}y}{\mathrm{d}t} = f(y,t)$$

设初值为$y|_{t=t_0} = y(t_0)$，若取计算步长为$h = t_n - t_{n-1}(n = 1,2,\cdots)$，则 t1 时刻的 y 的精确值应该为$y(t_1) = y(t_0) + \int_{t_0}^{t_1} f(y,t)\mathrm{d}t$，实际计算时，对积分项做近似计算，从而形成了各种不同的数值解法。

例如取$y_1 \approx y_0 + f(y_0,t_0)h$，即认为$[y_0,y_1]$区间内，y 的导数为定长，亦即近似取 y(t)为直线，斜率为y_0'，即该区间始点的导数。这种微分方程的数值解法称为向前欧拉法。

还可取$y_1 \approx y_0 + f(y_1,t_1)h$，则此时虽假定 y(t)在$[y_0,y_1]$近似为直线，但取$y_1' = f(y_1,t_1)$即区间终点的倒数为直线的斜率，此即为向后欧拉法，向后欧拉法计算中，由于等式两边均含有计算区间终点的函数值 y1，故 y1 的计算要通过解方程取得，一般称有此特点的数值积分方法为隐式接法。相反地，前向欧拉法则不需要解方程，即可直接计算 y1 值，这类数值积分方法称为显式解法。

也可以取

$$y_1 \approx y_0 + \frac{1}{2}[f(y_0, t_0) + f(y_1, t_1)]h$$

这种数值积分法是认为$y' = f(y,t)$为一直线，并用梯形面积$\frac{1}{2}(y_0' + y_1')h$来近似实际的积分面积$\int_{t_0}^{t_1} f(y,t)\mathrm{d}t$，所以称之为梯形积分法，这也是一种隐式解法。

2. 改进欧拉法

改进欧拉法$t_n \sim t_{n+1}$时步的数值积分是分预报和校正两步进行的，其通用计算公式对于方程$\frac{\mathrm{d}y}{\mathrm{d}t} = f(y,t)$

预报时：

$$y_{n+1}^{(0)} = y_n + f(y_n, t_n)h$$

校正时：

$$y_{n+1}^{(1)} = y_n + \frac{1}{2}\left[f(y_n, t_n) + f\left(y_{n+1}^{(0)}, t_{n+1}\right)\right]h$$

因此，可以改写为

$$y_{n+1} = y_{n+1}^{(1)} = y_{n+1}^{(0)} + \frac{1}{2}\left[f\left(y_{n+1}^{(0)}, t_{n+1}\right) - f(y_n, t_n)\right]h$$

3. 隐式梯形积分法

对于$\frac{\mathrm{d}y}{\mathrm{d}t} = f(y,t)$，若在$[t_n, t_{n+1}]$区段近似认为$y' = f(y,t)$为直线（参见下图），

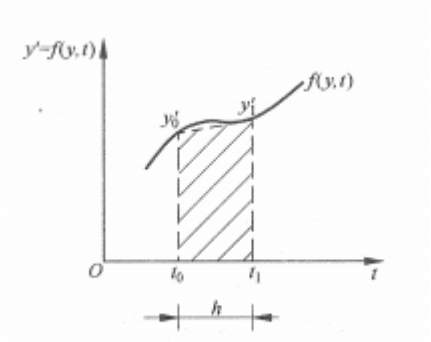


图 8-6　隐式梯形积分法示意图

图 7 隐式梯形法积分示意图

并以该直线下的梯形面积$\frac{h}{2}(y_n' + y_{n+1}')$近似于$\int_0^{2+1} f(y,t)\mathrm{d}t$的真值，则通用的计算公式为

$$\begin{aligned} y_{n+1} &\approx y_n + \frac{h}{2}(y_n' + y_{n+1}') \\ &= y_n + \frac{h}{2}[f(y_n, t_n) + f(y_{n+1}, t_{n+1})] \end{aligned}$$

这种解法属于单步、隐式解法，y_{n+1}要通过求解方程才能得到。本文中只分析典型微分方程$\frac{\mathrm{d}x}{\mathrm{d}t}=\lambda \mathrm{x}$。

结合上述两个方程，得到递推公式

$$x_{(k+1)}=x_{(k)}+\frac{\Delta t}{2}\left(f\left(x_{(k)}\right)+f\left(x_{(k+1)}\right)\right)=x_{(k)}+\frac{\Delta t}{2}\left(\lambda x_{(k)}+\lambda x_{(k+1)}\right)$$

整理得

$$x_{(k+1)}=\frac{1+\frac{\lambda\Delta t}{2}}{1-\frac{\lambda\Delta t}{2}}x_{(k)}$$

三、过程论述

1. 理论思路：

见部分二分析，将公式及迭代部分转化为代码。微分方程求解可抽象为三部分：初始化、微分方程函数储存、逐步求解，下面是详细分析

(1) 初始化：设置步长、求解范围、初值

(2) 微分方程函数储存：储存$\frac{\mathrm{d}y}{\mathrm{d}t}=f(y,t)$中的$f(y,t)$

(3) 逐步求解：根据步长生成离散时间序列 t，，对于$t_0,t_1,\dots t_n$,根据不同的求解方法逐步求出相应的系统状态矢量值$\boldsymbol{y}_0,\boldsymbol{y}_1,\dots,\boldsymbol{y}_n$

2. 代码展示(以欧拉法为例)

```
class Euler_Slove:
    # 初始化状态
    def __init__(self,x0,t_range,n,function_str):
        # 范围
        self.t_range = t_range
        # 步长
        self.n = n
        # 初值
        self.x0 = x0
        # 函数值 2
        self.function_str = function_str
        # 用于储存微分方程的解的列表
        self.dx_dt_list = []
        self.x_list = []
```

```
        # 求解微分方程
        self.slove()

    # 可以通过这里修改函数来修改所使用的微分方程
    def function(self,x):
        return eval(self.function_str)
    def slove(self):

        step = 1 / self.n
        t = np.arange(self.t_range[0],self.t_range[1],step)
        dx_dt_list=[]
        x_list=[]
        # 带入初值
        x = self.x0
        dx_dt = self.function(x)

        # 开始循环
        for i in t:
            # 记录上一时步的值
            x_list.append(x)
            dx_dt_list.append(dx_dt)
            # 更新斜率
            dx_dt = self.function(x)
            # 计算横坐标变换值
            x_delta = dx_dt*step
            # 曲线上下一个点
            x = x+x_delta
        self.dx_dt_list = dx_dt_list
        self.x_list = x_list
```

其中，改进欧拉法与欧拉法只有 for 循环内即求解方法不一样，改进欧拉法的 for 循环部分为

```
    # 开始循环
    for i in t:
        # 记录迭代前的值值
        x_list.append(x)
        dx_dt_list.append(dx_dt)
        # 更新斜率
        dx_dt_0 = self.function(x)
        # 按普通欧拉法计算横坐标变换值
        x_0= x+dx_dt*step
        # 计算欧拉法得到修正后的函数值
```

```
            dx_dt = self.function(x_0)
            # 修正后的x
            x = x + 1/2*(dx_dt_0+dx_dt)*step
        self.dx_dt_list = dx_dt_list
        self.x_list = x_list
```

3. 隐式梯形积分法求解

由于隐式梯形积分法需要解方程，因此只针对常见 dx/dt = λx 的情况，提前预设好方程求解的流程，因此传入的参数不一样，微分方程储存部分和求解部分如下：

```
class Implicit_Trapezoid_Method:
    # 初始化状态
    def __init__(self,x0,t_range,n,lamuda):
        self.lamuda = lamuda
    .....(与上同，略)
    # 可以通过这里修改函数来修改
    def function(self,x):
        return self.lamuda*x
    def slove(self):
        .....(与上同，略)
        # 开始循环
        for i in t:
            # 记录迭代前的值值
            x_list.append(x)
            dx_dt_list.append(dx_dt)
            # 修正并求解方程后得到的x
            x = x*(1+self.lamuda*step/2)/(1-self.lamuda*step/2)
            # 使用修正后的x 计算
            dx_dt = self.function(x)
        self.dx_dt_list = dx_dt_list
        self.x_list = x_list
```

4. 图形化输出

采用 matplotlib 中 pyplot 的，由于 matplotlib 预设不支持显示中文，因此需要提前设定。设置代码如下

```
# 设置绘图参数
plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False
```

为了便于直观展示，将三组数据作为三个子图绘制在同一个窗口中。求解数据及绘图部分如下

```
plt.figure(figsize=(12, 4))
plt.subplot(1, 3, 1)
plt.plot(t, x1, linewidth=3, color='r', label='欧拉法求解')
plt.xlabel('t', fontsize=24)
plt.ylabel('x', fontsize=24)
plt.legend(prop=font1)
plt.grid()

plt.subplot(1, 3, 2)
plt.plot(t, x2, linewidth=3, color='b', label='改进欧拉法求解')
plt.xlabel('t', fontsize=24)
plt.ylabel('x', fontsize=24)
plt.legend(prop=font1)
plt.grid()

plt.subplot(1, 3, 3)
plt.plot(t, x3, linewidth=3, color='g', label='隐式梯形法求解')
plt.xlabel('t', fontsize=24)
plt.ylabel('x', fontsize=24)
plt.legend(prop=font1)
plt.grid()

plt.show()
```

四、结果分析

输出图片结果如下

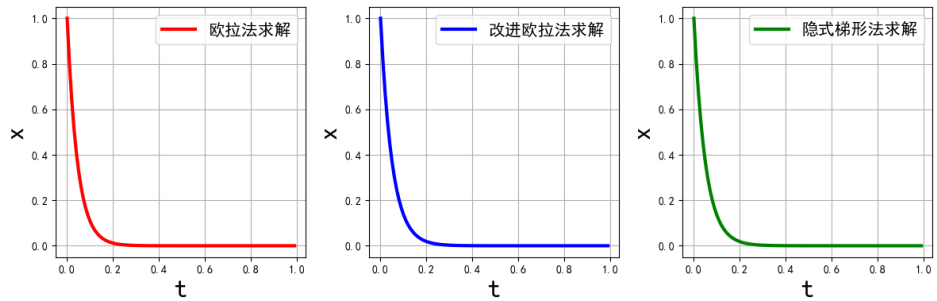


图 8 n=100 时求解结果比较

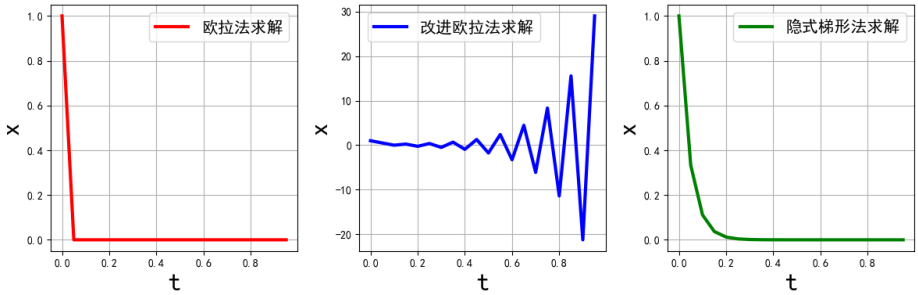


图 9 n=20 时求解结果比较

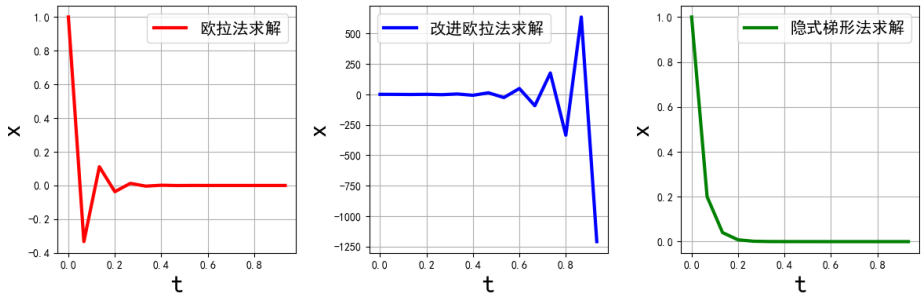


图 10 n=15 时求解结果比较

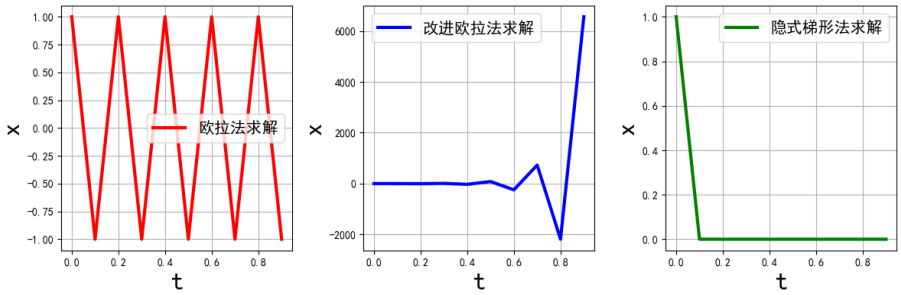


图 11 n=10 时求解结果比较

可以看到，随着步长的不断增加，三种求解方式的结果稳定性、精度都在不断降低。

结果具体分析如下：

对于微分方程$\frac{dx}{dt}=\lambda x$，欧拉法的递推公式为$x_{(k+1)}=x_{(k)}+\frac{dx}{dt}\Big|_k\Delta t=x_{(k)}+\lambda x_{(k)}\Delta t=(1+\lambda\Delta t)x_{(k)}$，则第 k 步的近似值为$\tilde{x}_{(k)}=x_{(k)}+\rho_{(k)}$，第 k+1 步的近似值为$\tilde{x}_{(k+1)}=(1+\lambda\Delta t)\left(x_{(k)}+\rho_{(k)}\right)=x_{(k+1)}+(1+\lambda\Delta t)\rho_{(k)}$，则第 k+1 步的误差为$\rho_{(k+1)}=$

$(1+\lambda\Delta t)\rho_{(k)}$，计算误差不增殖的条件为$|1+\lambda\Delta t|\leq 1$，本题中 λ=-20，因此求解不增殖的条件为$\Delta t<0.1$，即$n>10$,与实验结果基本一致。

同样，对于该微分方程，改进欧拉法的递推公式为$x_{(k+1)}=\left[1+(\lambda\Delta t)+\frac{1}{2}(\lambda\Delta t)^2\right]x_{(k)}$，第 k 步和第 k+1 步的近似值分别为$x_{(k)}+\rho_{(k)}$和$x_{(k+1)}+\left[1+(\lambda\Delta t)+\frac{1}{2}(\lambda\Delta t)^2\right]\rho_{(k)}$，因此，误差不增殖的条件为$\left|1+(\lambda\Delta t)+\frac{1}{2}(\lambda\Delta t)^2\right|\leq 1$，当 λ =-20 时，解得$\Delta t<0.05$，即$n>20$,与实验结果基本一致。因此可以得出结论，改进欧拉法的稳定性弱于欧拉法，但精度更高。

对于该微分方程，按上述分析思路，隐式梯形法计算误差不增殖的条件为$\left|\frac{1+\frac{\lambda\Delta t}{2}}{1-\frac{\lambda\Delta t}{2}}\right|\leq 1$，整理得$\mathrm{Re}\,(\lambda\Delta_t)\leq 0\}$，因此，只要 λ<0，必然稳定。

五、代码附录

这里只附上调用过程，绘制部分前文已存在，不再细述。

```
if __name__ == '__main__':
    x0 =1
    t_range=[0,1]
    n=10
    function_str = '-20*x'
    t = np.arange(t_range[0],t_range[1],1/n)

    x1 = Euler_Slove(x0,t_range,n,function_str).x_list
    x2 = Modified_Euler_Slove(x0,t_range,n,function_str).x_list
    x3 = Implicit_Trapezoid_Method(x0,t_range,n,-20).x_list
```

题目三 三机电力系统受到扰动后的时域仿真。

一、题目内容

三机电力系统如下图所示，已知各元件参数如下（电抗均为标幺值）：

发电机 G-1：$x'_d = 0.1$，$x_2 = 0.1$，$T_J = 10s$；

发电机 G-2：$x'_d = 0.15$，$x_2 = 0.15$，$T_J = 7s$；

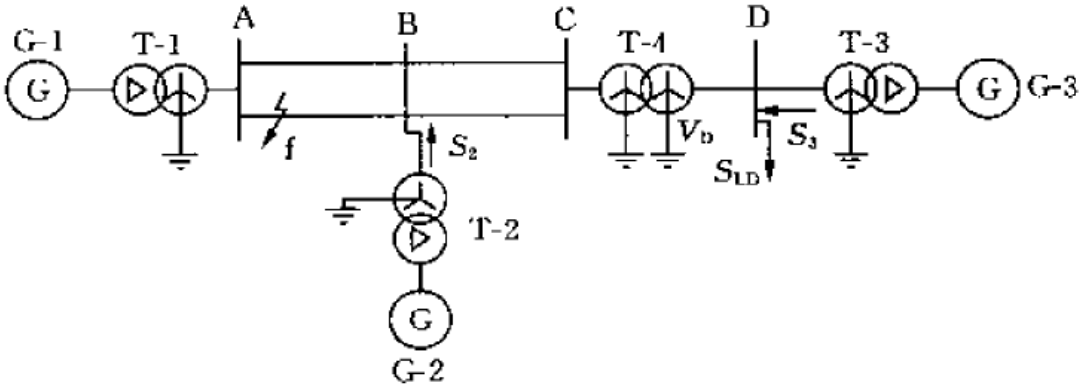
发电机 G-3：$x'_d = 0.06$，$x_2 = 0.06$，$T_J = 15s$；

变压器电抗：$x_{T1} = 0.08$，$x_{T2} = 0.1$，$x_{T3} = 0.04$，$x_{T4} = 0.05$；

线路电抗：AB 段双回$x_{L1} = 0.2$，$x_{L0} = 3.5x_{L1}$；BC 段双回$x_{L1} = 0.1$，$x_{L0} = 3.5x_{L1}$。

系统的初始状态：$V_{D0} = 1.0$，$S_{LD0} = 5.5 + j1.25$，$S_{20} = 1.0 + j0.5$，$S_{30} = 3.0 + j0.8$。

扰动事件描述：在线路 AB 段首端f点发生两相短路接地，经$0.1s$切除故障线路。



要求：

1. 使用 MATLAB 或 Python 语言编写；
2. 选择适合自己的数值计算方法编写计算程序；
3. 步长建议取$\Delta t = 0.02s$；
4. 自行分析应该建立的常微分方程组或微分代数方程组的具体表现形式；
5. 自行学习图形化输出方法，将仿真计算结果输出成图像，包括所有状态量以及所有相对角度；

6. 程序完成后撰写编写思路的文档，课程结束后和代码一起提交作为课程分数依据。

设计通用的数据格式，把案例原始数据用通用格式保存，程序通过读取数据文件获得原始数据。也就是说程序是通用的，案例的特殊性通过具体的数据文件体现，换一个满足格式的案例，不需要修改程序代码就能得到计算结果，同时输入输出要尽可能简单。

二、方案论证

1.暂态稳定的基本概念

主要目的是检查系统在大扰动下各发电机组能否保保持同步运行。当电力系统受到大扰动时，发电机的输入机械功率和输出电磁功率失去平衡，引起转子速度的及角度的变化，各机组间发生相对摇摆，其结果可能有两种不同的情况，第一种情况时摇摆逐渐平息下来，系统中各发电机仍然能保持同步运行，过渡到一个新的运行状态，则认为系统在此扰动下是暂态稳定的。另一种情况是这种摇摆最终使一些发电机的相对角度不断增大，这时我们称发电机失去了同步。因此，电力系统暂态稳定分析的基本任务中最大量的分析是功角稳定问题。

2. 潮流方程求解

对于 N 个节点的电力系统，每个节点有四个运行变量(例如，对于节点 i 有 Pi，Qi，Vi,和 θ i)，故全系统共有 4N 个变量。一般来说，每个节点的四个变量中给定两个，另外两个代求，哪两个作为定量由该节点的类型决定。

对于负荷节点，给节点的 P,Q 是由负荷需求决定的，一般是不可控的。该类节点的特点是 P,Q 给定，则该节点 V，θ 待求，这类节点称为 PQ 节点。无注入的联络节点也可以看作 P,Q 给定节点，其 P,Q 值都为 0

对于发电机节点，由于发电机励磁调节的作用使该节点的电压幅值维持不变，有功功率由发电机输出功率决定，所以该节点的 P,V 给定，V、θ 待求，这类节点称为 PV 节点。

全系统还应该满足功率平衡条件，因此只要有一个节点的 P、Q 不能提前给出，其值要待潮流计算结束，Ploss 和 Qloss 确定之后才能确定，该节点称为平衡节点。

为了满足程序的通用性，本文中并未采用像 17-7 习题解答中由负荷点电压为参考电压，默认电机 PQ 功率恒定，通过计算功率损耗从而计算电压降落的方法，而是模仿

ieee 标准数据的格式并对该格式进行简化和修改，将该系统变为一个四节点三发电机系统，发电机节点中选取未接负荷的电机 1 作为平衡节点，另外两个电机则作为 PV 节点，使用直角坐标系下的牛拉法进行潮流计算求解电压和功率。

以下为直角坐标系下牛拉法的潮流计算流程：

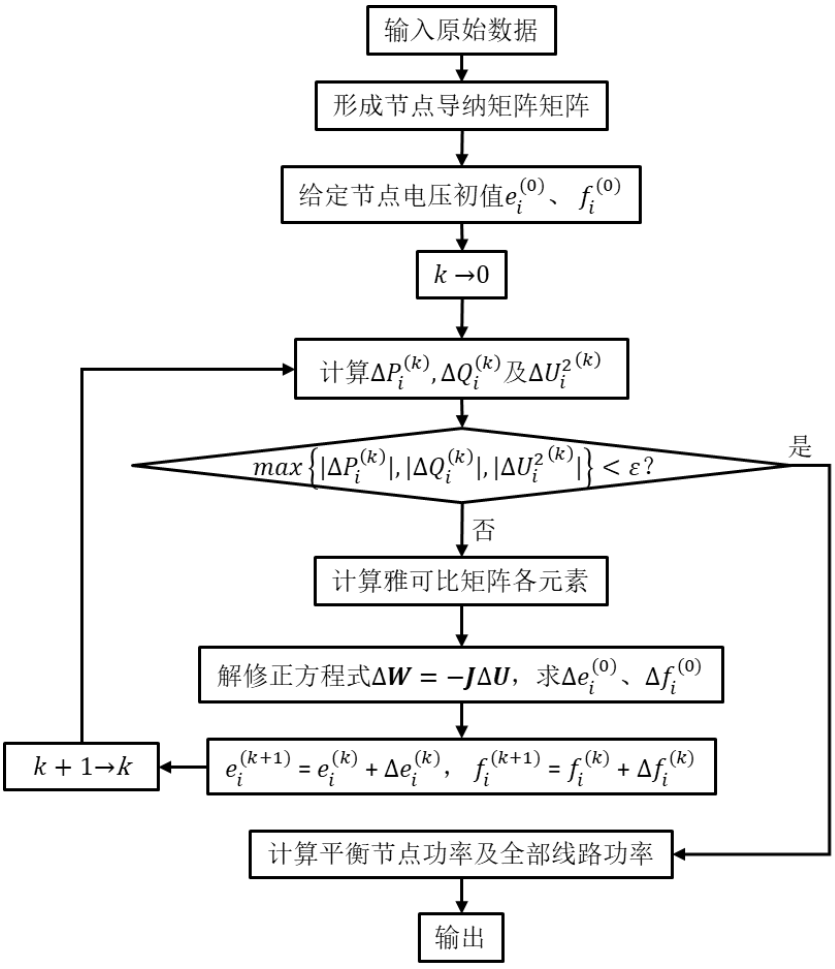


图 12 直角坐标系下牛拉法潮流计算流程

3. 时域仿真法

将电力系统各元件模型根据元件间拓扑关系形成全系统模型，这是一组联立的微分方程组和代数方程组，然后以稳态工况或潮流解为初值，求扰动下的数值解，即逐步求得系统的状态量和代数量随时间的变化曲线，并根据发电机转子的摇摆曲线来判别系统在大扰动下能否保持同步运行，即暂态稳定性。

本文中采用潮流计算的解为初值，对发电机采用经典二阶模型，忽略凸极效应，并假设暂态电抗 Xd 与暂态电动势 E'的幅值恒定，从而忽略励磁系统的动态，以简化分析。

E'恒定一定程度上计及了励磁系统的作用，即认为励磁系统足够强，从而保持 Xd'后的 E'恒定。另外，本文中忽略调速器和原动机动态作用，即认为机械功率 Pm 为定常值。其中需要注意的要点如下：

(1)在潮流计算的基础上进行导纳矩阵修改

负荷模型采用最简单的线性负荷模型，从而对于三相对称负荷有：$\dot{U}_{\mathrm{L}} = Z_{\mathrm{L}}\dot{I}_{\mathrm{L}}$将因此可以将负荷阻抗并入导纳矩阵，从而将负荷接点转化为网络节点，相应节点的注入电流化为 0。

发电机忽略定子绕组内阻的定子电压标幺值方程为：$\dot{U}_{\mathrm{G}} = \dot{E}' - \mathrm{j}X'_d\dot{I}_{\mathrm{G}}$，将发电机方程改写为导纳方程形式，因此可把 Yg 并入网络导纳矩阵，则联立求解发电机和网络方程的问题就转化为在发电机节点注入电流时，网络方程的求解。

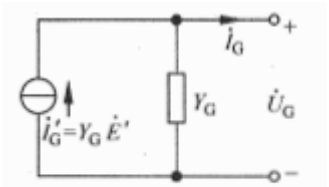


图 8-2 经典模型发电机等值电路图

图 13 经典模型发电机等值电路图

(2)初值确定

(3)转子运动方程求解

本文中采用改进欧拉法对三个电机的角度及角速度六个状态量进行微分方程的求解。

已知转子运动方程为

$$\begin{cases}\omega_{n+1} \approx \omega_n + \dot{\omega}_n h = \omega_n + h\left(P_{m,n} - P_{e,n}\right)/T_J \\ \delta_{n+1} \approx \delta_n + \left[\dfrac{\omega_n + \omega_{n+1}}{2} - 1\right]h\end{cases}$$

计算 t 时刻的$\left|\frac{\mathrm{d}\omega}{\mathrm{d}t}\right|_{t_n}$ 和 $\left|\frac{\mathrm{d}\delta}{\mathrm{d}t}\right|_{t_n}$，

$$\begin{cases}\left|\dfrac{\mathrm{d}\omega}{\mathrm{d}t}\right|_4 = \left[P_m - P_{e,n} - D(\omega_n - 1)\right]/T_1 \\ \left|\dfrac{\mathrm{d}\delta}{\mathrm{d}t}\right|_{\tau_n} = (\omega_n - 1)\omega_{\mathrm{a}}\end{cases}$$

其中发电机功率部分在下一小节细述

预报$\omega_{n+1}^{(0)}, \delta_{n+1}^{(0)}$为，$\begin{cases}\omega_{x+1}^{(0)}=\omega_n+\left.\frac{d\omega}{\mathrm{d}t}\right|_{t_n} h \\ \delta_{n+1}^{(t)}=\delta_n+\left.\frac{\mathrm{d}\delta}{\mathrm{d}t}\right|_{t_n} h\end{cases}$，式中$h=t_{n+1}-t_n$。

预估完毕后，对发电机状态量进行校正

$$\begin{cases}\omega_{n+1}=\omega_{n+1}^{(0)}+\frac{h}{2}\left[\left(P_m-P_{e,n+1}^{(0)}\right)/T_J-\left.\frac{\mathrm{d}\omega}{\mathrm{d}t}\right|_{\tau_\alpha}\right] \\ \delta_{m+1}=\delta_{n+1}^{(0)}+\frac{h}{2}\left[\left(\omega_{n+1}^{(0)}-1\right)\omega_{\mathrm{n}}-\left.\frac{\mathrm{d}\delta}{\mathrm{d}t}\right|_{t_n}\right]\end{cases}$$

使用校正后的值再次求解t_{n+1}时刻的电磁功率，此时$t_n \sim t_{n+1}$时步的计算工作完成。

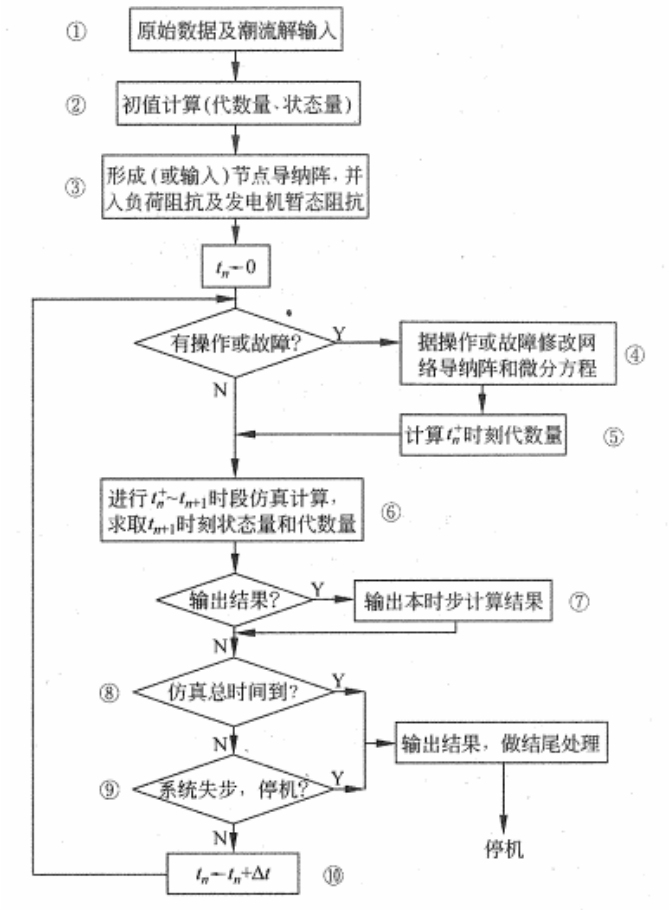


图 14 本文的模型暂态稳定分析流程图

4. 多机功率特性

采用下图中等值电路，仅保留发电机电势源节点和参考节点，对潮流计算用节点

导纳矩阵进行修改，得到一个多电势源的线性网络，此线性网络的导纳节点方程为$\boldsymbol{I}_{\mathrm{G}}=\boldsymbol{Y}_{C;}\boldsymbol{E}_{\mathrm{G}}$，$\boldsymbol{I}_{\mathrm{G}}=[\dot{I}_{\mathrm{G1}}\quad \dot{I}_{\mathrm{G2}}\cdots\quad \dot{I}_{\mathrm{Gm}}]^{\mathrm{T}}$是发电机输出电流的列向量，$\boldsymbol{E}_{\mathrm{G}}=[\dot{E}_{\mathrm{Gl}}\quad \dot{E}_{\mathrm{G2}}\cdots\dot{E}_{\mathrm{G}n}]^{\mathrm{T}}$是各发电机电势的列向量。

因此求得发电机电流$\dot{I}_{\mathrm{G}i}=\sum_{j=1}^{n}Y_{ij}\dot{E}_{\mathrm{G}j}(i=1,2,\cdots,n)$，带入到发电机功率算式，有$S_{Gi}=P_{Gi}+jQ_{Gi}=\dot{E}_{Gi}I_{Gi}^{*}$，整理后得：

$$\left.\begin{aligned}P_{\mathrm{Gi}}&=E_{\mathrm{Gi}}^{2}|Y_{ii}|\sin\ \alpha_{ii}+\sum_{j=1,j\neq i}^{n}E_{\mathrm{Gi}}E_{\mathrm{G}j}|Y_{ij}|\sin\left(\delta_{ij}-\alpha_{ij}\right)\\Q_{\mathrm{Gi}}&=E_{\mathrm{Gi}}^{2}|Y_{ii}|\cos\ \alpha_{ii}-\sum_{j=1,j\neq i}^{n}E_{\mathrm{Gi}}E_{\mathrm{G}j}|Y_{ij}|\cos\left(\delta_{ij}-\alpha_{ij}\right)\end{aligned}\right)$$

$$\left.\begin{aligned}\alpha_{ij}&=90^{\circ}-\operatorname{arctg}\frac{-B_{ij}}{G_{ii}}\\\alpha_{ij}&=90^{\circ}-\operatorname{arctg}\frac{B_{ij}}{-G_{ij}}\end{aligned}\right\}$$

也可表示为

$$\left.\begin{aligned}P_{\mathrm{Gi}}&=\frac{E_{\mathrm{Gi}}^{2}}{|Z_{\mathrm{u}}|}\sin\ \alpha_{ii}+\sum_{j=1.j\neq i}\frac{E_{\mathrm{Gi}}E_{\mathrm{G}j}}{|Z_{ij}|}\sin\left(\delta_{ij}-\alpha_{ij}\right)\\Q_{\mathrm{Gi}}&=\frac{E_{\mathrm{Gi}}^{2}}{|Z_{ii}|}\cos\ \alpha_{ii}-\sum_{j=1,j\neq i}^{n}\frac{E_{\mathrm{Gi}}E_{\mathrm{Gj}}}{|Z_{ij}|}\cos\left(\delta_{ij}-\alpha_{ij}\right)\end{aligned}\right\}$$

其中Z_{ii}和Z_{ij}并不是对导纳矩阵求逆后对应位置的元素，或者说互阻抗，而是自导纳Y_{ii}和互导纳Y_{ij}的倒数，也就是所谓的输入阻抗和转移阻抗。

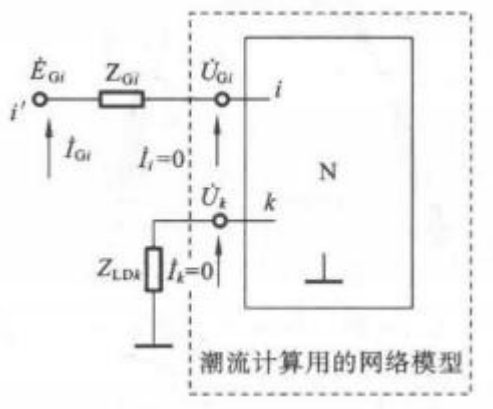


图 15 仅保留发电机电势节点的网络模型

5. 网络操作及故障处理

本文中设网络元件的三相参数对称，采用 0、1、2 对称分量坐标，在三相对称运行时，零序网、负序网不起作用，仅正序网起作用，它同发电机、负荷接口，全系统只存在正序电量。因此，对于简单的不对称故障，则在形成网络正序、负序、零序导纳的基础上，将故障点 a,b,c 三相电压电流关系转化为该节点处序网电量关系。由于暂态稳定中一般不要求计算负序、零序电量以及各相电量的分布，故负序、零序网的作用可以看作正序网中一个等值的阻抗，从而各时步的计算实际上在正序网中进行，下图为简单不对称故障中序网连接下负序、零序接入正序网的等值阻抗。

表 8-1　简单不对称故障时的序网连接(a 相为特殊相)

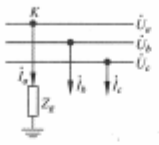
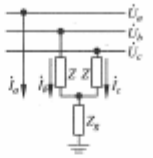
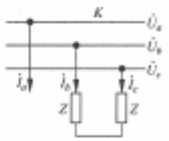
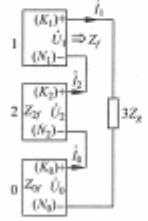
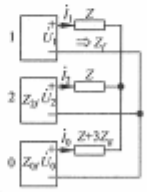
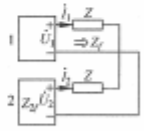
故障形式 / 序网连接及参数	单相接地	两相接地	两相短路
短路点参数			
边界条件（相分量）	$\dot{I}_b-\dot{I}_c=0$ $\dot{U}_a=Z_g\dot{I}_a$	$\dot{I}_a=0$ $\dot{U}_b-Z\dot{I}_b-\dot{U}_c-Z\dot{I}_c$ $=(\dot{I}_b+\dot{I}_c)Z_g$	$\dot{I}_a=0,\dot{I}_b=-\dot{I}_c$ $\dot{U}_b-Z\dot{I}_b-\dot{U}_c-Z\dot{I}_c$
边界条件（序分量）	$\dot{I}_0=\dot{I}_1=\dot{I}_2$ $\dot{U}_0+\dot{U}_1+\dot{U}_2-3Z_g\dot{I}_0=0$	$\dot{I}_0+\dot{I}_1+\dot{I}_2=0$ $\dot{U}_0-(Z+3Z_g)\dot{I}_0$ $-\dot{U}_1-Z\dot{I}_1=\dot{U}_2-Z\dot{I}_2$	$\dot{I}_1+\dot{I}_2=0,\dot{I}_0=0,$ $\dot{U}_1-Z\dot{I}_1=\dot{U}_2-Z\dot{I}_2,$ $\dot{U}_0=0$
序网连接			
负序、零序网接入正序网的等值阻抗	$Z_f=Z_{2f}+Z_{0f}+3Z_g$	$Z_f=(Z_{2f}+Z)/\!/$ $(Z_{0f}+Z+3Z_g)+Z$	$Z_f=(Z_{2f}+Z)+Z$
金属短路时（$Z=Z_g=0$）等值阻抗	$Z_f=Z_{2f}+Z_{0f}$	$Z_f=Z_{2f}/\!/Z_{0f}$	$Z_f=Z_{2f}$

图 16 简单不对称故障序网连接分析

本文中采用 Python 为编程语言，使用 PyCharm 为 IDE。

三、过程论述

1. 理论思路

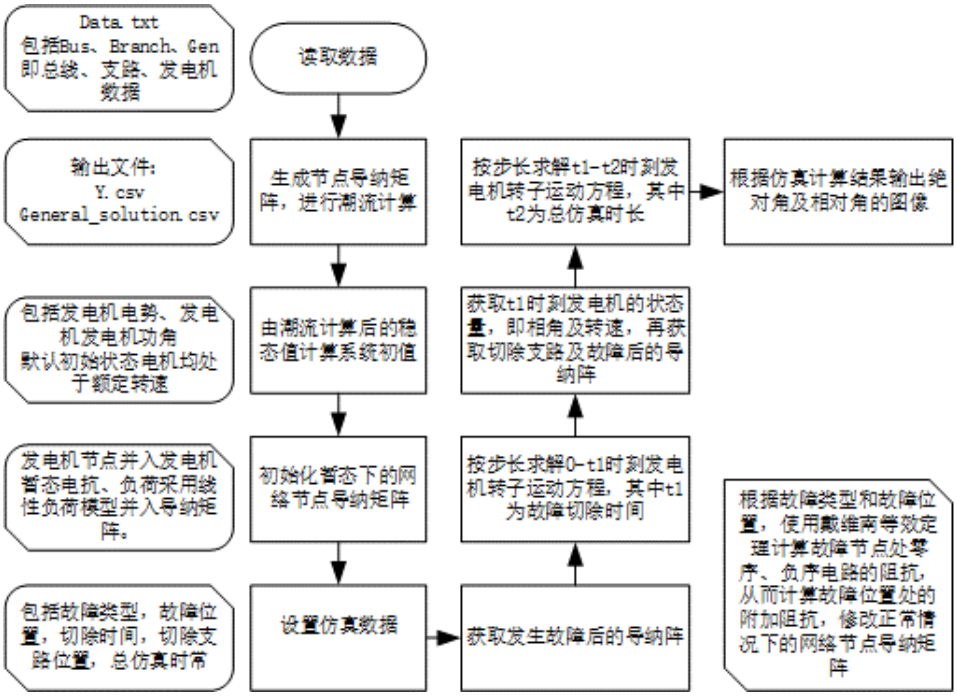


图 6 代码总体思路

2. 代码流程图

根据理论思路设计大致框图如下：

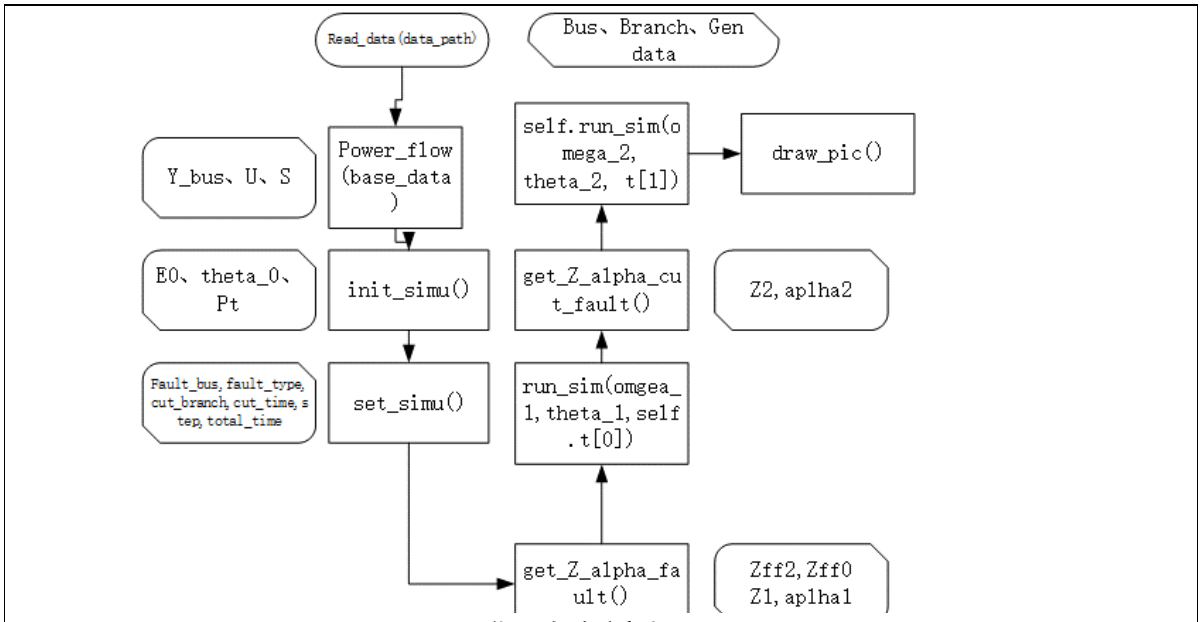


图 7 代码大致流程框图

3. 输入文本说明(data.txt)

双斜杠//表示该行为注释说明，读入数据时会被自动忽略。

输入设计分为三部分，分别为母线数据(BUS)、支路数据(BRANCH)、发电机(GEN)数据，以数据类型的大写名称作为段落的划分。

数据设计表如下，程序中提取注释行之外的文字，按列读取数据。

本文中按题目要求设计的数据格式如下：

表格 1 BUS **数据格式说明**

BUS				
列名	BusNumber	Type	FinV	FinAngle
对应列数	1-11	12-16	17-22	23-31
说明	母线编号	节点类型	节点电压	节点相角
列名	LoadP	LoadQ	GenP	GenQ
对应列数	32-38	39-45	46-50	51-60
说明	有功负荷	无功负荷	发电机有功	发电机无功

表格 2 BRANCH **数据说明**

BRANCH					
列名	BranchName	Tnumber	Znumber	R1	X1
对应列数	1-8	9-16	17-21	22-25	26-31
说明	支路编号	变压器非基准变比侧节点	变压器阻抗侧	正序电阻	正序电抗
列名	R2	X2	R0	X0	
对应列数	32-35	35-43	43-48	48-55	
说明	负序电阻	负序电抗	零序电阻	零序电抗	

表格 3 GEN **数据说明**

GEN			
列名	Name	Bus	Xd1
对应列数	1-8	9-13	14-19

说明	变压器编号	所连接母线节点	暂态电抗
列名	X2	Tj	Xt
对应列数	19-25	26-30	31-40
说明	负序电抗	惯性时间常数	升压变压器电抗

```
BUS
BusNumber Type FinV FinAngle LoadP LoadQ   GenP GenQ
1          3    1     0        0      0      0    0
2          2    1     0        0      0      1.0  0.5
3          1    1     0        0      0      0    0
4          2    1     0        5.5    1.25   3    0.8

//默认是标幺值
BRANCH
BranchName TNumber ZNumber R1  X1    R2   X2     R0   X0
1       1       2    0   0.2   0   0.2     0    0.7
2       2       3    0   0.1   0   0.1     0    0.35
3       3       4    0   0.05  0   0.05    0    0.05
//自带变压器的发电机，默认连接电机侧的变压器都是三角接法，因此零序电路不考虑发电机阻抗
GEN_T
Name   BUS   Xd1   X2   Tj  Xt
1       1    0.1   0.1   10   0.08
2       2    0.15  0.15  7    0.1
3       4    0.06  0.06  15   0.04
```

图 18 按习题 17-7 设计的数据

4. 代码模块说明

(1)read_data.py

读取 data.txt 并解析数据，保存至 MyData 类的 bus_data_dict_list、branch_data_dict_list、gen_data_dict_list 三个列表中，方便使用。

(2)power_flow.py

使用 MyData 类的数据作为 base_data 即数据基类，进行节点导纳矩阵的形成，使用直角坐标系下的牛拉法进行潮流方程的求解。

输出文件'Y_bus.csv','general_solution.csv'为暂态仿真提供稳态数据。其中'general_solution.csv'包括节点的电压幅值、向量及功率，'Y_bus'为节点导纳矩阵。

(3)transient_sim.py

暂态仿真的整个过程，以下是部分函数说明：

a.read_data()

用 MyData 类的数据作为数据基类，读取潮流解算模块下的文件'Y_bus.csv','general_solution.csv'

b.set_simu()

设置仿真过程的数据，包括切除时间、故障类型、故障节点、仿真总时间、仿真步长等。

c.init_sim_data()

根据输入的潮流结果及同步电机稳态向量关系计算系统状态量以及代数量在 t=0 刻的初值,包括电机功率、转速、相角、电压值等。并按第二部分中所分析的修改系统稳态工况下的导纳矩阵。

d.run_sim_1()

根据故障节点及故障类型计算故障点的附加阻抗，从而修改导纳矩阵。计算转移阻抗从而得到发电机功率与相角、转速的关系。按步长求解 0-t1 时刻发电机转子运动方程，其中 t1 为故障切除时间。

e.run_sim_f()

根据切除的部分修改导纳矩阵。计算转移阻抗从而得到发电机功率与相角、转速的关系。按步长求解 t1-t2 时刻发电机转子运动方程，其中 t2 为总仿真时长

f.plot_data()

通过时域仿真下计算出的各发电机绝对角计算各发电机之间的相对相角，根据步长使用 pyplot 绘制 δ -t 曲线。

由于篇幅限制，文后只附加部分关键计算的代码，总体代码见附件

5. 交互界面设计

为了方便多次修改仿真的故障参数以及查看变量值，使用 pyqt5 设计如下界面，

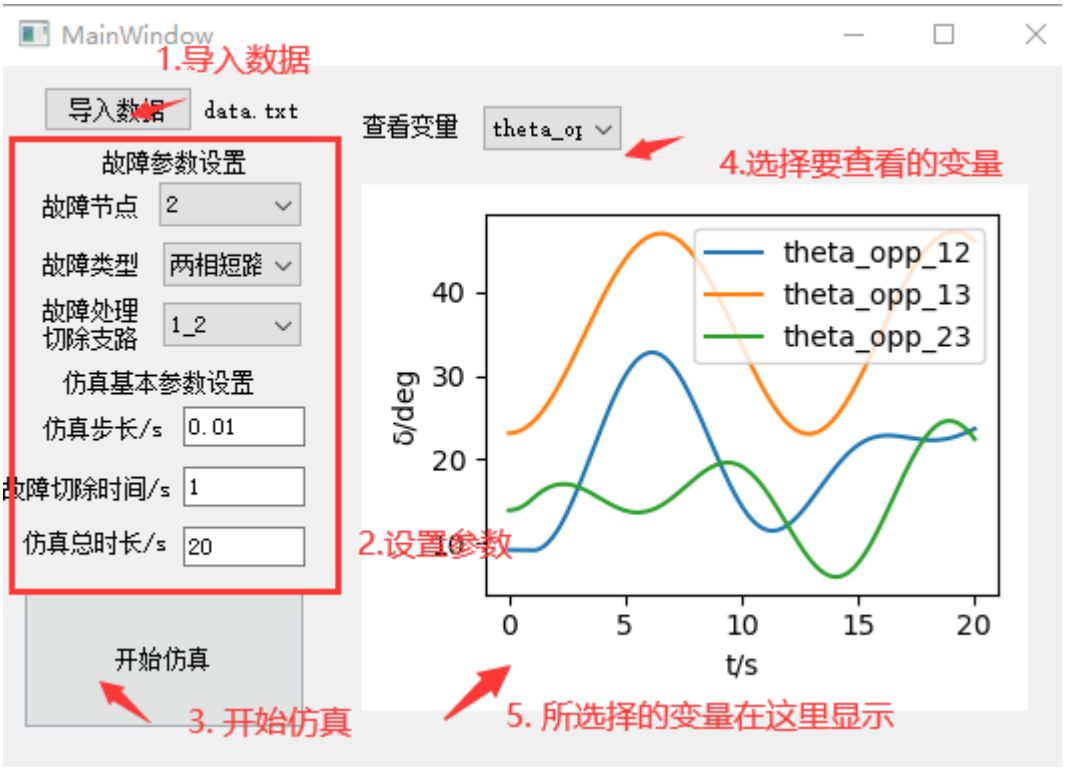


图 8 交互界面说明

具体作品展示请看附件中视频。

由于使用 pyinstaller 打包成 exe 过程中遇见尚未明白原因的编译问题（似乎和使用了 numpy 库，调用了 mkl 有关），因此没有按预想生成打包好的 exe，因此，需要直接运行 py 文件， 程序入口为 mainWindow.py。

四、结果分析

根据最终结果按步长绘制绝对角和相对角。

得到 17-7 所给条件下各发电机绝对角和相对角变化如下图

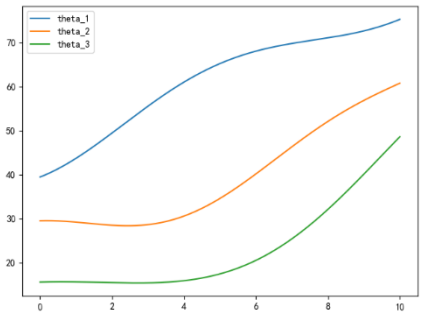


图 19 各发电机绝对角变化

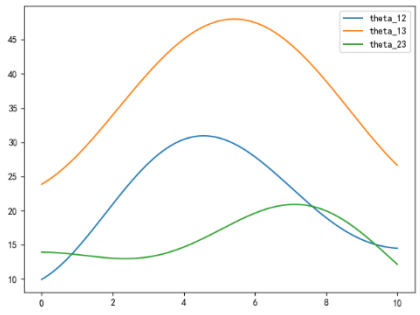
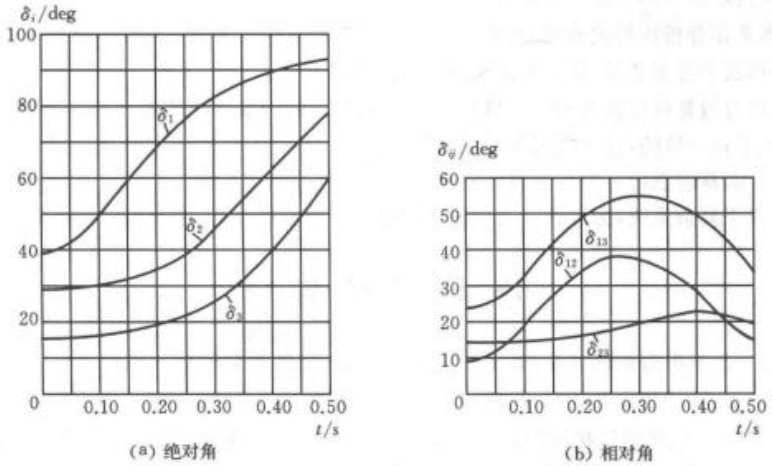


图 20 各发电机相对角变化

与课本上求解曲线进行对照，基本一致。



题 17-7 图(6) 转子摇摆曲线

图 21 习题 17-7 解题结果

五、代码附录

由于代码内容较多，这里只取 transient_simu.py 即暂态仿真模块的关键部分

1. 获取故障后的只含电机节点的转移阻抗矩阵以及角度，用于初态和后续暂态仿真的计算

```
def get_Z_alpha_fault(self):
    bus_num = len(self.base_data.bus_data_dict_list)
    gen_num = len(self.base_data.gen_data_dict_list)
```

```
# 初始化为潮流计算用的节点导纳矩阵

Y1 = self.Y_powerflow

zero_row = np.zeros([bus_num], dtype=complex)
# 增加矩阵的维度,维度数为电机数
for i in range(0, gen_num):
    Y1 = np.row_stack((Y1, zero_row))
zero_cloumn = np.zeros([gen_num + bus_num], dtype=complex)
for i in range(0, gen_num):
    Y1 = np.column_stack((Y1, zero_cloumn))

# 生成端口-节点关联矩阵
Ml = []
for index, gen_data in enumerate(self.base_data.gen_data_dict_list):
    # bus_index = gen_data['BUS']-1
    Ml_temp = np.zeros(bus_num + gen_num)
    # 发电机
    Ml_temp[index + bus_num] = 1
    Ml.append(Ml_temp)
Ml = np.array(Ml)

# 修改导纳矩阵
# 修改矩阵,将暂态电抗并入导纳方程，发电机节点电压即为暂态电压，并且幅值不变，相角改变
for index, gen_data in enumerate(self.base_data.gen_data_dict_list):
    bus_index = gen_data['BUS'] - 1

    Xd1 = gen_data['Xd1']
    Xt = gen_data['Xt']
    X1 = complex(0, Xd1 + Xt)

    # 修改自阻抗
    Y1[bus_num + index][bus_num + index] += 1 / X1
    # 修改连接母线节点的自阻抗
    Y1[bus_index][bus_index] += 1 / X1
    # 修改所连接的母线节点的自阻抗
    Y1[bus_index][bus_num + index] -= 1 / X1
    Y1[bus_num + index][bus_index] -= 1 / X1

# 修改由负荷影响的矩阵
for index, bus_data in enumerate(self.base_data.bus_data_dict_list):
```

```
        bus_index = bus_data['BusNumber'] - 1
        loadP = bus_data['LoadP']
        loadQ = bus_data['LoadQ']
        loadS = complex(loadP, loadQ)
        U = self.u_powerflow[bus_index]

        # 由于负荷接地，修改自导纳即可
        if loadP != 0 or loadQ != 0:
            Z_ld = loadS * (abs(U) ** 2) / (abs(loadS) ** 2)
            Y1[bus_index][bus_index] += 1 / Z_ld

# 修改由附加电抗影响的矩阵部分
fault_type_list = ['单相接地', '两相短路', '两相接地', '三相短路']
# 计算附加阻抗
self.zff2 = self.get_Zff2()
self.zff0 = self.get_Zff0()

fault_type = '两相接地'
self.z_delta = complex(0, 0)
if fault_type == '单相接地':
    self.z_delta = self.zff0 + self.zff2
elif fault_type == '两相短路':
    self.z_delta = self.zff2
elif fault_type == '两相接地':
    self.z_delta = (self.zff0 * self.zff2) / (self.zff0 + self.zff2)
elif fault_type == '三相短路':
    self.z_delta = 0

# 只需修改故障节点的自导纳
if self.z_delta != 0:
    y_delta = 1 / self.z_delta
    Y1[self.fault_bus - 1][self.fault_bus - 1] += y_delta

Ml = Ml.T
Y1 = Y1.astype(np.complex)

Z1 = np.linalg.inv(Y1)


Zeq = np.matmul(Ml.T, np.matmul(Z1, Ml))
# 以电机连接节点为端口的戴维南等效电阻
Zeq = Zeq.astype(np.complex)
```

```
    # 以电机连接节点为端口的诺顿等效电路
    Yeq = np.linalg.inv(Zeq)

    # 转移阻抗矩阵
    Z = np.zeros([gen_num, gen_num], dtype=float)
    alpha = np.zeros([gen_num, gen_num], dtype=float)
    for index1, item1 in enumerate(Yeq):
        for index2, item2 in enumerate(item1):
            # 转移阻抗是节点导纳矩阵的倒数，而不是求逆
            z = 1 / item2

            Z[index1][index2] = abs(z)
            # 角度注意是弧度角！！！
            alpha_temp = math.atan(z.imag / z.real) * 180 / 3.14
            # 转化到1，2 象限
            if alpha_temp < 0:
                alpha_temp = alpha_temp + 180
            alpha[index1][index2] = 90-alpha_temp

    return Z, alpha
```

2. 故障节点为 1，需要获得负序看进去的戴维南，从而获得附加电抗

```
def get_Zff2(self):
    # 母线数量
    bus_num = len(self.base_data.bus_data_dict_list)

    # 关联向量,长度为节点数
    Ml = np.zeros(bus_num)
    # 故障节点为1
    Ml[self.fault_bus-1] = 1

    # 形成负序节点导纳矩阵,节点数目仍为母线数目，因为发电机接地了
    Y_2 =np.zeros([bus_num,bus_num],dtype=complex)
    # 可以作为一个单独的程序,参数为电路的序，待完善
    for bus_data_dict in self.base_data.bus_data_dict_list:
        # 获取当前节点
        node_num =bus_data_dict['BusNumber']
```

```
        # 遍历该节点连接的母线
        for branch_data_dict in self.base_data.branch_data_dict_list:
            if    branch_data_dict['TNumber']==(node_num):
                to_node_num = branch_data_dict['ZNumber']

                # 考虑线路的阻抗对矩阵的影响
                node_data_z = complex(branch_data_dict['R2'],
                                      branch_data_dict['X2'])
                node_data_y = 1 / node_data_z

                # 修改节点自导纳
                Y_2[node_num-1, node_num-1] +=node_data_y
                # 修改连接节点自导纳
                Y_2[to_node_num-1, to_node_num-1] += node_data_y
                # 修改互导纳
                Y_2[node_num-1, to_node_num-1] -= node_data_y
                Y_2[to_node_num-1, node_num-1] -= node_data_y




# 修改导纳矩阵
# 修改发电机阻抗对节点的影响
for index,gen_data in enumerate(self.base_data.gen_data_dict_list):
        bus_index = gen_data['BUS']-1
        # Xd1 = gen_data['Xd1']
        X2 = gen_data['X2']
        Xt = gen_data['Xt']
        Xg2 = complex(0,X2+Xt)
        # 修改连接母线节点的自阻抗
        Y_2[bus_index][bus_index] += 1 / Xg2
```

```
    # 修改由负荷影响的矩阵
    for index,bus_data in enumerate(self.base_data.bus_data_dict_list):
        bus_index = bus_data['BusNumber']-1
        loadP = bus_data['LoadP']
        loadQ = bus_data['LoadQ']
        loadS = complex(loadP,loadQ)
        U = self.u_powerflow[bus_index]

        # 由于负荷接地，修改自导纳即可
        if loadP!=0 or loadQ!=0:
            Z_ld = loadS*(abs(U)**2)/(abs(loadS)**2)
            z = complex(0,0.35*abs(Z_ld))
            Y_2[bus_index][bus_index] += 1/z

    Y_2 = Y_2.astype(np.complex)
    Z_2 = np.linalg.inv(Y_2)
    Zeq = np.matmul(Ml.T, np.matmul(Z_2, Ml))
    return Zeq
```

3. 时域仿真的计算部分

```
# 开始仿真,仿真只和初值和仿真时段有关
def run_sim(self,omega_init,theta_init,t):

    gen_num = len(self.base_data.gen_data_dict_list)
    omega_B = 2*180*self.f_B

    # 应该是一个 n*t 的矩阵，n 是发电机数目，t 是仿真的全部步长
    P = [np.zeros(gen_num) for i in t]
    omega = [np.zeros(gen_num) for i in t]
    theta =[np.zeros(gen_num) for i in t]

    # 状态量微分值
    domega_dt = [np.zeros(gen_num) for i in t]
    dtheta_dt = [np.zeros(gen_num) for i in t]
```

```
        # 微分方程初值
        # 初始角度
        theta[0] = theta_init
        # 初始转速都为额定转速
        omega[0] = omega_init

        # 开始计算,从 t=1 开始
        for t_index,t_step in enumerate(t[:-1]) :
            # 用第 i 步状态量的初值计算代数量
            P[t_index] = self.get_P(theta[t_index])

            # dw/dt_n = Pm - Pen - D(omega -1)
            # D 为常阻尼系数，需要计及机械阻尼时才有
            for gen_index in range(gen_num):
                # 前半步,计算初始时间段变化速度
                domega_dt[t_index][gen_index] = ((self.PT[gen_index] -
P[t_index][gen_index])/self.Tj[gen_index])*180/3.14
                dtheta_dt[t_index][gen_index]=omega[t_index][gen_index]-omega_B

                # 求得时间末段近似值
                omega[t_index+1][gen_index] = omega[t_index][gen_index]+
domega_dt[t_index][gen_index]*self.step
                theta[t_index+1][gen_index] = theta[t_index][gen_index]+
dtheta_dt[t_index][gen_index]*self.step

            # 根据 i+1 的预测状态量预估 i+1 时刻的代数量
            P[t_index+1]= self.get_P(theta[t_index+1])

            # 计算时间末端变化速度
            for gen_index in range(gen_num):

                domega_dt[t_index+1][gen_index] = ((self.PT[gen_index] -
P[t_index+1][gen_index])/self.Tj[gen_index])*180/3.14
                dtheta_dt[t_index+1][gen_index]= omega[t_index+1][gen_index]-omega_B

            # 校正数据
            for gen_index in range(gen_num):
                omega[t_index+1][gen_index] =
omega[t_index][gen_index]+(domega_dt[t_index+1][gen_index]+domega_dt[t_index][gen_index])*self.ste
p/2
```

```
                theta[t_index+1][gen_index] =
theta[t_index][gen_index]+(dtheta_dt[t_index+1][gen_index]+dtheta_dt[t_index][gen_index])*self.step/2

        self.omega.append(omega)
        self.theta.append(theta)
        self.P.append(P)
        return omega[-1],theta[-1]
```

4. 点击开始仿真后的整个流程

```
# 启动仿真
def start_sim(self):
        gen_num = len(self.base_data.gen_data_dict_list)
        omega_B = 2*180*self.f_B

        self.init_sim_data()

        omgea_1 =np.array([omega_B for i in range(gen_num)])
        theta_1 = np.array(self.E0_theta)
        omega_2,theta_2 = self.run_sim(omgea_1,theta_1,self.t[0])

        # 获取故障切除后的转移阻抗
        self.Z, self.alpha = self.get_Z_alpha_cut_fault()
        omega3, theta3 = self.run_sim(omega_2, theta_2, self.t[1])
```

5. 数据的处理与绘制

```
def process_result(self,para_name):
        gen_num = len(self.base_data.gen_data_dict_list)
        t =[]
        for para_list in self.t:
                t.extend(para_list)
        t = np.array(t)
        t = t.reshape((-1,1))
        result_list = []
        index_list =[]
        if para_name == 'omega':
                omega = []
                for para_list in self.omega:
                        omega.extend(para_list)
                theta = np.array(omega)
                for i in range(gen_num):
                        temp_result = theta[:,i]
```

```
            temp_result = temp_result.reshape((-1,1))
            result_list.append(temp_result)
            index_list.append(i+1)

    if para_name == 'theta_abs':
        theta = []
        for para_list in self.theta:
            theta.extend(para_list)
        theta = np.array(theta)
        for i in range(gen_num):
            temp_result = theta[:,i]
            temp_result = temp_result.reshape((-1,1))
            result_list.append(temp_result)
            index_list.append(i + 1)

    if para_name == 'theta_opp':
        theta = []
        for para_list in self.theta:
            theta.extend(para_list)
        theta = np.array(theta)

        # 用组合的方式算出有哪些相对角
        opp_list =    list(combinations(range(gen_num), 2))
        for i_j in opp_list:
            i,j =i_j
            temp_result_i = theta[:, i]
            temp_result_j = theta[:, j]
            temp_result = temp_result_i-temp_result_j
            result_list.append(temp_result)
            index_list.append('{}{}'.format(str(i+1),str(j+1)))

    if para_name == 'P':
        P = []
        for para_list in self.P:
            P.extend(para_list)
        P = np.array(P)
        for i in range(gen_num):
            temp_result = P[:,i]
            temp_result = temp_result.reshape((-1,1))
            result_list.append(temp_result)
            index_list.append(i + 1)
```

```
        return t,result_list,index_list

    def draw_pic(self):
        self.fig.clf()
        self.axe = self.fig.add_subplot(111)
        para_name = self.com_var.currentText()
        t,result_list,index_list= self.transient_sim.process_result(para_name)

        for index,result in enumerate (result_list):
            self.axe.plot(t, result, label='{}_{}'.format(para_name,index_list[index]))
        self.axe.set_xlabel('t/s')
        self.axe.set_ylabel("δ /deg")
        self.axe.legend()
        # 必须重新调整子图，不然不会显示坐标
        self.fig.tight_layout()
        self.canvas.draw()
```

参考文献

［1］倪以信，陈寿孙，张宝霖 动态电力系统的理论和分析［M］. 清华大学出版社，2002. 125-179

［2］张伯明，陈寿孙，严正 高等电力网络分析［M］. 清华大学出版社，2007. 25-299

［3］张伯明，郭庆来 高等电力网络分析习题解答［M］. 清华大学出版社，2009. 38-44

［4］何仰赞，温增银 电力系统分析(下)［M］. 华中科技大学出版社，2016. 151-205

［5］何仰赞，温增银 电力系统题解［M］. 华中科技大学出版社，2008. 152-163