

**《仿真技术在电气工程中领域的应用》**

**课程作业**

**(2019-2020 学年第一学期)**

**作业0202 稀疏线性方程组的求解**

**作业0203 微分方程的求解**

**作业0302 多机系统短路故障后时域仿真**

**学院：电力学院**

**任课老师：武志刚**

**组员信息及分工**

组员：阮艺源 胡嘉铭 周欣缘 汪笑雨

阮艺源 17级电气工程卓越班 201730210466

1. 问题1 部分Python程序编写

2. 问题2 基于欧拉法、改进欧拉法、隐式梯形法的微分方程求解的Python程序编写、绘图、中文文档写作

3. 问题3 通用牛顿拉夫逊法、隐式梯形法接口的Python程序编写、绘图、相应部分文档写作

胡嘉铭 17级电气工程卓越班 201730220489

1. 问题1 部分Python程序编写、中文文档写作
2. 问题2 讨论并共同设计编程思路
3. 问题3 输入阻抗、转移阻抗的通用计算（包括Y.py和impedance.py文件）、通用数据格式接口的设计及程序编写、相应部分中文文档写作

周欣缘 17级电气工程中澳班 201730241286

1. 问题1 MATLAB程序版本

2. 问题2 MATLAB程序版本

3. 问题3 MATLAB程序版本

4. 英文版本的论文写作

汪笑雨 16级电气工程五班 201530212585

1. 问题1 讨论程序编写思路、修改论文
2. 问题2 讨论程序编写思路、修改论文
3. 问题3 节点导纳矩阵的程序编写、修改论文

**作业0202 稀疏线性方程组的求解**

1. **问题重现**

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。



其中：





1. **解题思路**
2. 基本作业要求

由于本文需解决的问题是：编写求解稀疏线性方程组的通用程序，因此可采用基于因子表分解的求解线性方程组的方法。

假设需要求解的方程组为：



又因为每个方阵都可以进行LDU分解，因此矩阵A可分解为下三角矩阵L、对角矩阵D、上三角矩阵的乘积U：



则求解方程组的步骤可分解为以下三步——前代、规格化、回代：







由此可解出未知量x。基于以上方法能将复杂的线性方程组的求解过程，简化为循环代入过程。

由于矩阵A是稀疏矩阵，我们对其进行三角检索存储，可免去对大量零元素进行存储，节约了大量内存。同时，由于我们求解线性方程组时也需要用到L、D、U矩阵，对矩阵完成存储后，也得到了L、D、U矩阵，此举可谓是一举两得。

1. 加分项

以上分析能够解决A矩阵稀疏时的问题。对于实际情况，有时可能我们只需要知道x矩阵中某个分量的值，而不需要将整个矩阵x求解出来。此时，我们可以利用稀疏向量法实现该想法，并且能够提高运算速度。

假设我们需要求解矩阵x中的第x\_index个分量。运用稀疏向量法时，我们可以在原来的回代步骤中进行化简，将与第x\_index个分量无关的分量的计算给舍去。经过分析，第i个元素和x\_index无关的充分条件是U[x\_index-1:i-1][i-1]全为0或i < x\_index。只需将矩阵U中与第x\_index个分量无关的分量对应的行列删去，再用处理后的矩阵U和x进行回代计算，即可实现稀疏向量法。

1. **实现程序**

本文程序是基于Python。

1. 三角检索存储

先对矩阵A进行LDU分解，将L、D、U三个矩阵结合成一个矩阵B，该矩阵B则为矩阵A的因子表。对其因子表进行散居存储，即实现了对原矩阵的三角检索存储。

程序中，函数LDU(A)即为将矩阵A进行LDU分解的函数，输入为矩阵A，输出依次为L、D、U三个矩阵。

函数LDU\_store(A)为将矩阵A进行三角检索存储的函数，函数中首先调用了函数LDU，构建矩阵A的因子表。该函数输出为散居存储格式，为三个元组:

(L\_value,L\_row,L\_column),(D\_value),(U\_value,U\_row,U\_column)

第一个元组有三个list元素：L矩阵的元素、其对应的行号、其对应的列号；

第二个元组有三个list元素：D矩阵的元素；

第三个元组有三个list元素：U矩阵的元素、其对应的行号、其对应的列号。

由于三角检索存储已能够生成L、D、U矩阵，下面的步骤可直接使用LDU\_store函数的输出结果进行计算。

1. 前代计算

由于L为下三角矩阵，经过一系列化简，求解Lz=b的步骤可简化为n次代入，依次求出z1、z2、z3…zn ，如下所示：



程序中，函数front(L,b)则为实现前代过程的函数，其输入为LDU\_store( )函数输出的第一个元组及b矩阵，输出为z矩阵。

1. 规格化计算

由于D为对角矩阵，因此求解Dy=z的步骤可简化为n次除法，依次求出y1、y2、y3…yn ，如下所示：

程序中，函数mid(D,z)则为实现前代过程的函数，其输入为LDU\_store( )函数输出的第二个元组及z矩阵，输出为y矩阵。

1. 回代计算

由于U为下三角矩阵，经过一系列化简，求解Ux=y的步骤可简化为n次代入，依次求出xn、xn-1、xn-2…x1 ，如下所示：



程序中，函数back(U,y)则为实现前代过程的函数，其输入为LDU\_store( )函数输出的第三个元组及y矩阵，输出为x矩阵，即线性方程组的解。

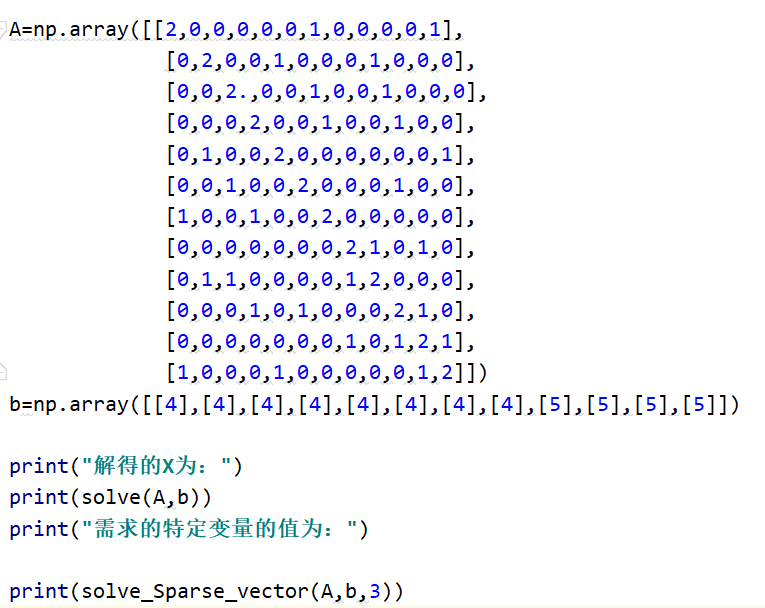
本程序将前代、规格化、回代三个步骤包装在函数solve(A,b)中，其输入为系数矩阵A和常量矩阵b，输出为需求解的未知量矩阵x。

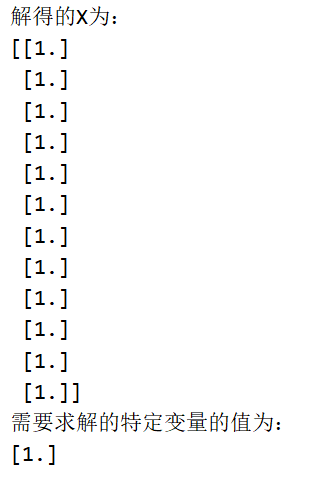
1. 稀疏向量法

在第二节已阐述了稀疏向量法的目的、意义及实现方法，下面具体介绍实际函数。本程序通过函数solve\_Sparse\_vector(A,b,x\_index)实现稀疏向量法，其输入为系数矩阵A、常量矩阵b和未知量矩阵x中需求解分量的序号x\_index，输出为需求解分量的值。

1. **结果展示**

本程序将原始数据通过参数传入函数solve( )中，解得未知量矩阵x，并且通过solve\_Sparse\_vector( )函数利用稀疏向量法求解出矩阵x中的第三个分量x3。





**作业0203 微分方程的求解**

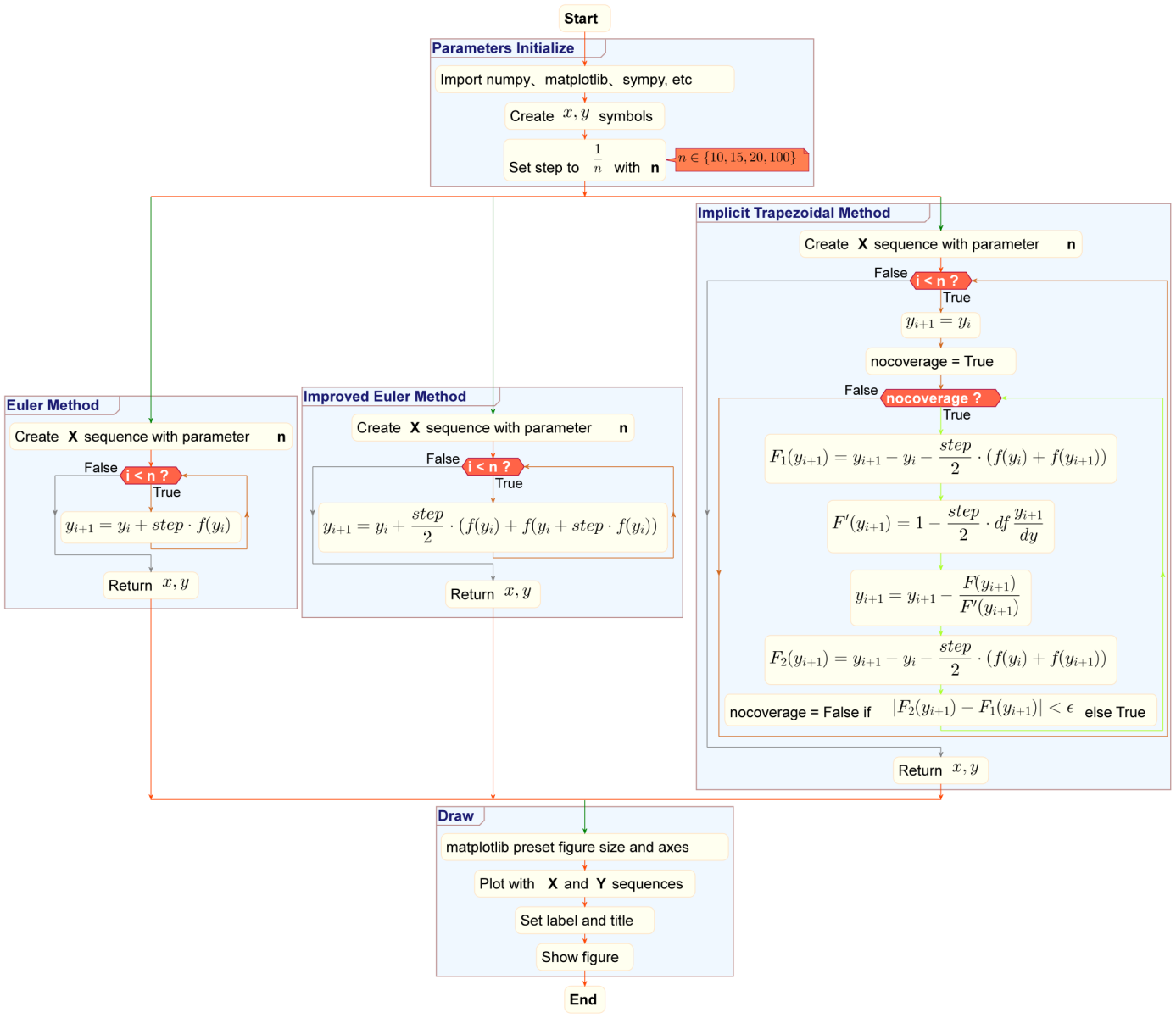
1. 问题重述

求解具有如下初值的微分方程



方程中f(y)不显含x，求解的结果为x与y在定义域[a,b]上的数值序列

二、程序涉及的微分方程求解的流程图

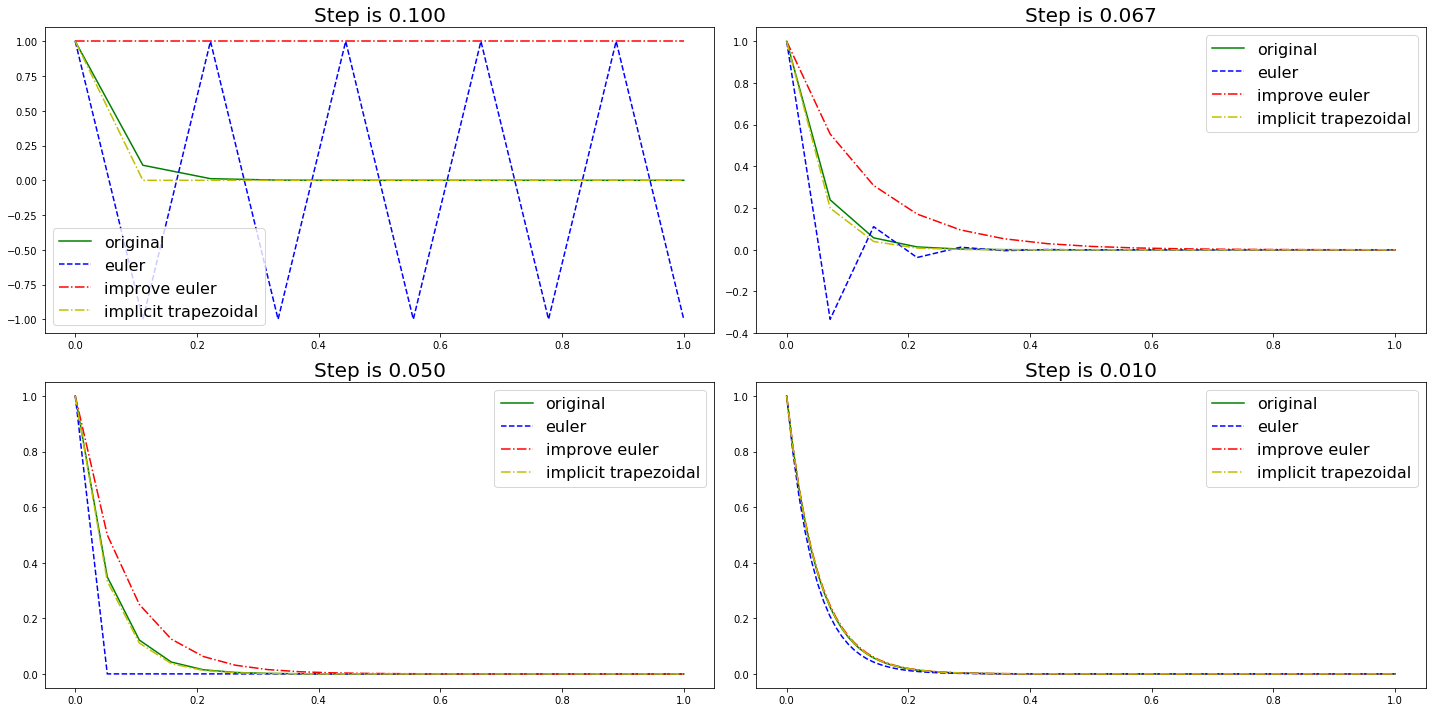


三、程序运行后的结果对比

求解的具体微分方程为：



当步长中n分别取10，15，20，100对应的欧拉法、改进欧拉法、隐式梯形法的图像分别如下：



1.当步长为**0.1**时(n=10)

欧拉法求得的图像上下波动，原因是f(y)=-20y与y负相关，过大的步长产生较大的误差，甚至使与其叠加后的y1为负值，于是f(y1)变为正值，叠加后导致y2又变为正值，如此往复，甚至形成一个无法收敛的波动。

改进欧拉法为一条与x轴平行的直线，其原因在于y0=1，f(y0)+f(y0+0.1**·**f(y0))=0,则叠加后y1=y0=1,于是根据数学归纳法，yn=yn-1=...=y0=1，即改进欧拉法的图像为一条直线。

隐式梯形法由于每一轮循环都要迭代至该轮收敛，因而在step=0.1时就能很快收敛至原函数。

1. 当步长为**0.067**时(n=15)

欧拉法的图像为衰减地收敛到与真实曲线重合，而改进欧拉法较真实解偏大，隐式梯形法的曲线依然与真实曲线高度吻合。

3当步长为**0.05**时(n=20)

欧拉法图像在y1后就保持为0值，原因是y1=y0+0.05\*f(y0)=0,则y2=y1+0.05\*f(y1)=0,由数学归纳法yn=yn-1=...=y1=0

改进欧拉法依然较真实解偏大。

4当步长为**0.01**时(n=50)

三种方法的解皆与真实解贴近，且三者的精确度为 ε排序为 ε隐>ε改>ε欧。

四、微分方程求解方法的数值评价

由于方程为非线性方程，不能选取拟合优度**R2**作为评价指标，这里直接选取误差平方和**ESS**作为评价指标，**ESS**指标如下：



|  |  |  |  |
| --- | --- | --- | --- |
| **Method  N** | **Euler** | **Improve Euler** | **Implicit Trapezoidal** |
| **10** | 9.20742838 | 8.76880526 | 0.01188318 |
| **15** | 0.333880795 | 0.200184481 | 0.001912445 |
| **20** | 0.138710433 | 0.049243171 | 0.000398136 |
| **100** | 0.011638651 | 0.000396699 | 5.57732E-05 |

注：蓝色为低精度，粉红色为高精度

由色阶图，欧拉法和改进欧拉法在步长很大时的精度远低于隐式梯形法，随着步长减小，欧拉法和隐式梯形法的精度均有明显提升，而在步长变化过程中隐式梯形法始终能保持很高的精度。

**作业0302 多机系统短路故障后时域仿真**

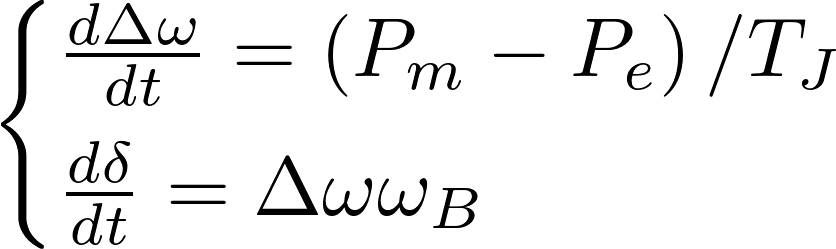
**一、问题重述**

给定初始状态稳定和元件参数已知的多机电力系统，对其短路故障和故障切除后的状态进行时域仿真。

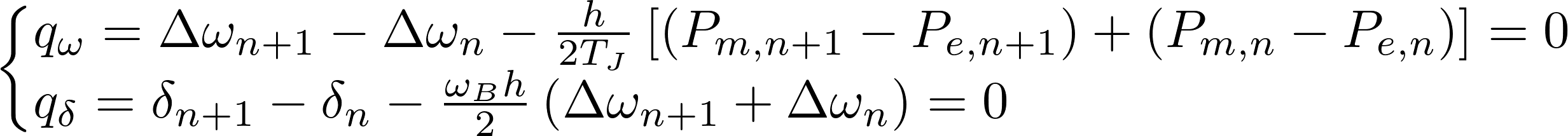
1. **问题建模**
2. 主要模型

研究电力系统运行的稳定性，主要是确定各同步电机是否处于同步运行状态，即各转子间有无相对摇摆。功角δ随时间变化描述了各发电机转子间的相对运动，恰好反映了各发电机之间是否同步。

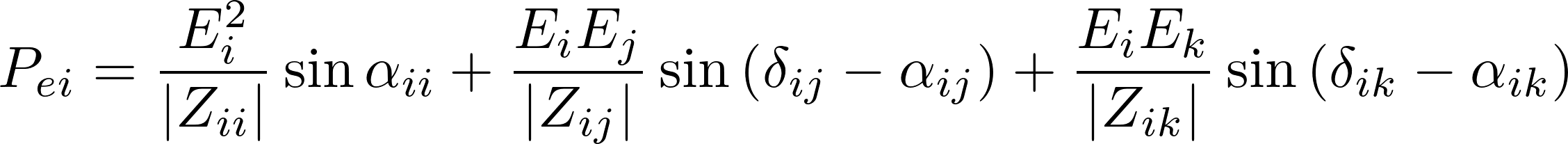
根据算例所给的条件特点，考虑使用同步电机转子二阶模型，即



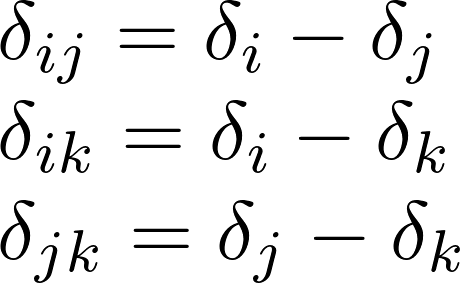
由隐式梯形法得，tn至tn-1时步的差分方程为



对于三机系统，任意一台发电机的电磁功率如下



其中



1. 方程组中未知参数的求解

本小节主要针对电磁功率Pe表达式中的未知参数进行计算。其中，未知参数主要为各台发电机的输入阻抗和之间的转移阻抗，Pe表达式中需要用到上述阻抗的模值及阻抗角。

计算输入阻抗和转移阻抗的关键是生成节点阻抗矩阵，根据计算公式即能算得所需阻抗值。为了正确生成故障发生时和故障切除后的节点阻抗矩阵，需要按以下步骤进行操作：

1. 定义通用的数据格式
2. 生成正负零序的节点导纳矩阵
3. 计算附加阻抗
4. 生成节点阻抗矩阵，计算发电机节点的输入阻抗和之间的转移阻抗

具体程序及实现方法将在第三节进行详细说明。

1. **程序流程**

本程序的实现流程及具体实现方法将在本节详细介绍。

1. 主程序

主程序，即利用牛顿-拉夫逊法及隐式梯形法求解模型的流程图，如下图所示：

****

该部分由power\_system\_stability.py文件内程序实现。

1. 接口函数

* newton\_laphson(Functions, init\_values, e=0.00001)

牛顿拉夫逊法的接口函数，其参数如下：

·Functions 需要通过牛拉法迭代的形如f(x,y,...,z)=0的一组代数方程

·init\_values 初始值的字典

·e 前后两次迭代的差值，所有方程组一次迭代前后差值最大者 < e

该函数将传入的方程组迭代收敛至给定误差后，返回迭代后值的字典。

* hiding\_trapezium(derivative, n=50, limit=(0,1), e=0.0001)

隐式梯形法的接口函数，其参数如下：

·derivative含所有方程（包括微分方程和代数方程）及相应初值的字典

·n 绘制点个数

·limit 绘图定义域

·e 传入牛拉法中的迭代精度

该函数将微分方程先变形为代数方程，并与代数方程一同传入牛拉法函数迭代。

* Pe(E=None, Delta=None, Z\_value=None, Alpha=None)

·E 算得得三台同步电机电势向量

·Delta 三台电机功角sympy符号

·Z\_value 每一次系统状态改变后，相应的发电机组输入、转移阻抗模值矩阵。

·Alpha 每一次系统状态改变后，相应的弧度制阻抗角矩阵

返回值为三台电机的电磁功率向量。

1. 程序编写思路
2. 隐式梯形法
3. 生成定义域内对应步长的numpy序列；
4. 将所有方程中属于微分方程的定义转换为隐式梯形法对应的代数方程；
5. 利用给定初始值生成每一时间因变量的numpy序列；
6. 将每一次牛拉法迭代后的初值和代数方程一同传入牛拉法迭代，产生下一时间点数据，修改因变量序列对应时间索引的值。
7. 牛顿拉夫逊法

为了发挥numpy、sympy数值计算的最大优势，使用sympy自动求导功能，并且成功尝试使用sy.diff函数，让代数n个方程组成的向量对由n个自变量组成的向量求导，从而能一步产生n\*n的求导后矩阵，大大简化了程序编写，具体过程如下：

1. 初始化雅可比矩阵及各参数；
2. 进入迭代过程；
3. 将每一次迭代初值代入代数方程；
4. 将每一次迭代初值代入雅可比矩阵，求解代数方程组计算因变量变化量Δ；
5. 利用Δ修改每一次迭代值，重复步骤3；
6. 判断两次的代数方程差是否小于给定误差，若小于则返回最终迭代值，否则回到过程3。
7. 计算Pe表达式参数流程：

****

该部分由Y.py和impedance.py文件中的程序实现。具体程序实现如下所示。

1. 定义通用的数据格式

本程序定义了通用的数据格式，使得对于任何满足格式的案例，不需要修改程序代码就能得到计算结果。

本程序定义了三个数据文件：描述节点参数的Node\_data.npy、描述支路参数的branch\_data.npy、描述变压器参数的trans\_data.npy。它们定义的皆为numpy的array格式，例题的数据保存在附件中的data文件夹中。具体定义格式分别如下所示：

Node\_data.npy的数据格式：

|  |  |
| --- | --- |
| 列数 | 含义 |
| 1 | 节点序号 |
| 2 | 节点类型（短路节点记为0,负荷节点或联络节点记为1,电源节点记为2） |
| 3 | 电源节点电动势E' |
| 4 | 电源节点的Xd'（正序） |
| 5 | 电源节点X2（负序） |
| 6 | 电源节点X0（零序） |

附：

1. 列3-6：不是电源节点的节点标为0
2. 本题中，节点0：地；节点1：A；节点2：B；节点3：C；

节点4：D；节点5：E1；节点6：E2；节点7：E3。

1. 电源节点定义是在Xd'外。

branch\_data.npy的数据格式：

|  |  |
| --- | --- |
| 列数 | 含义 |
| 1 | 起始节点编号 |
| 2 | 终止节点编号 |
| 3 | 切除故障前支路上等价的电阻 |
| 4 | 切除故障前支路上等价的电抗 |
| 5 | 切除故障后支路上等价的电阻 |
| 6 | 切除故障后支路上等价的电抗 |
| 7 | 零序电路中的电抗 |

附：

（1）若终止节点为地（负荷），则将其置零。

（2）将短路节点定为节点1。  
（3）原始数据中不需要加入Xd'。

trans\_data.npy的数据格式：

|  |  |
| --- | --- |
| 列数 | 含义 |
| 1 | 变压器两端节点中靠近短路点的节点序号 |
| 2 | 变压器两端节点中远离短路点的节点序号 |
| 3 | 变压器在列1对应一端的绕组接法 |
| 4 | 变压器在列2对应一端的绕组接法  （YN：0, Y:1, D:2） |

1. 生成节点导纳矩阵

利用支路追加法，根据支路参数的数据可生成节点导纳矩阵。

该部分程序在Y.py文件中。文件中定义了两个函数，Y\_generation( )和Y\_generation\_ZS( )，前者可以生成正负序的节点导纳矩阵，后者可生成零序的节点导纳矩阵。由于正负序电路中除Xd’外的参数是相同的，而零序电路中线路电抗与正负序电路不同，故设计两个函数分别计算。

1. 计算附加阻抗

根据正序等效定则，不对称故障分析时，可将负序、零序电路的效果等效为附加阻抗叠加在正序电路上，使得分析不对称故障时只需分析扩展正序电路。

1. 负序、零序短路点处的输入阻抗

向branch\_data中增加发电机节点的Xd2’及改变负荷等效阻抗值为负序电抗值，即能生成新的temp\_branch。将temp\_branch传入节点导纳矩阵生成函数Y\_generation( )中，能够生成负序电路的节点导纳矩阵Y2。将Y2求逆即可得到节点阻抗矩阵Z2，从中便能读出负序电路中短路点处的输入阻抗Z2ff 。

零序电路的求解方法类似，不同之处为：零序电路需要考虑零序电流的流通情况，此时则需要变压器参数的数据trans\_data。对于两侧不都是星型接地的变压器，零序电流不能流经其下游电路。并且，零序电路中不考虑负荷。

程序中，通过NSI( )和ZSI( )分别计算得到负序、零序短路点处的输入阻抗。

1. 附加阻抗的生成

根据正序等效定则，对于不同的不对称故障类型，附加阻抗的计算方式不同。本程序由函数X\_star( )生成附加阻抗X△，其第四个输入参数为短路类型，输入不同的数值则能计算得到不同的。其定义如下所示：

|  |  |  |
| --- | --- | --- |
| 短路类型 | 对应的第四个输入参数 | 附加阻抗的计算 |
| 三相短路 | 1 | 0 |
| 两相接地短路 | 2 | X2ff // X0ff |
| 两相短路 | 3 | X2ff |
| 单相短路 | 4 | X2ff + X0ff |

1. 计算发电机节点的输入阻抗和之间的转移阻抗

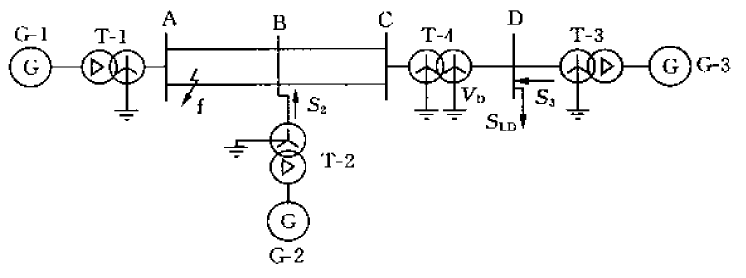
此步需要计算出节点阻抗矩阵，并由此得到输入阻抗和转移阻抗。生成节点阻抗矩阵前，首先必须得到对应的正确的支路数据。

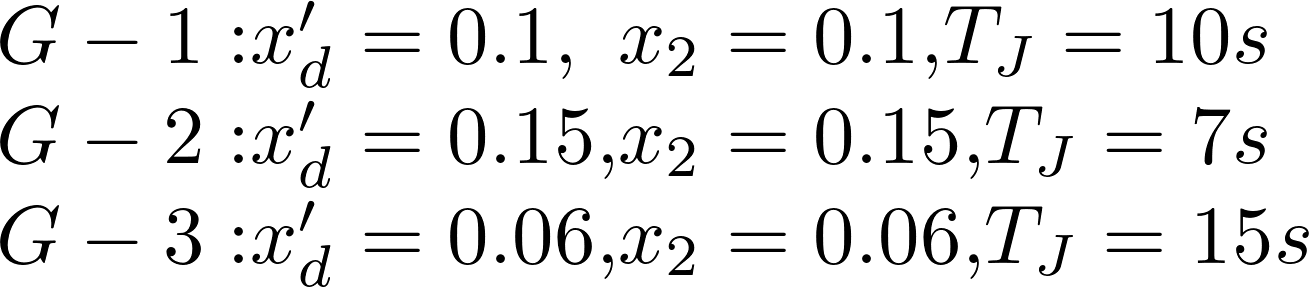
本程序通过函数branch\_data\_for\_imp\_SC( )，向branch\_data中增加发电机节点的Xd2’及短路节点处的X△，生成故障发生时的新支路数据。利用函数Impedance( )调用新支路数据，生成发电机节点的输入阻抗和之间的转移阻抗。输出结果为m×m矩阵Z，对角元为输入阻抗，非对角元为转移阻抗（m为发电机节点的个数）。

通过函数branch\_data\_for\_imp\_afterCut( )，向branch\_data中增加发电机节点的Xd2’并更新切除故障处附近的支路数据，生成故障切除后的新支路数据。类似上段做法，生成故障切除后的发电机节点输入阻抗和转移阻抗组成的m×m矩阵。

通过函数Z\_value( )和alpha( )，可以生成上述矩阵Z各元素的模值和阻抗角，将其代入模型的Pe表达式中。

1. **程序求解的算例**

****

****

变压器电抗：，，，；

线路电抗：AB段双回，；BC段双回，。

系统的初始状态：，，，。

扰动事件描述：在线路AB段首端点发生两相短路接地，经切除故障线路。

1. **程序运行结果及分析**
2. 故障各阶段仿真时间的分配

仿真过程中0-0.05s为稳定状态，0.05sf点发生两相接地短路，故障持续至0.15s，0.15s后故障切除。

1. 仿真0.5s后各状态量，及同步电机相对功角变化曲线图

由曲线图1，可得出以下结论：

1. 由系统中各同步电机相对功角变化，可知该系统在电机第一摆能保持稳定，而未发散。
2. 三台机组中，仅有电机一在故障切除，即0.15s左右速度达到第一摆动最大，电机二、电机三分别在0.44s及0.5s后才至第一摆动最大值。

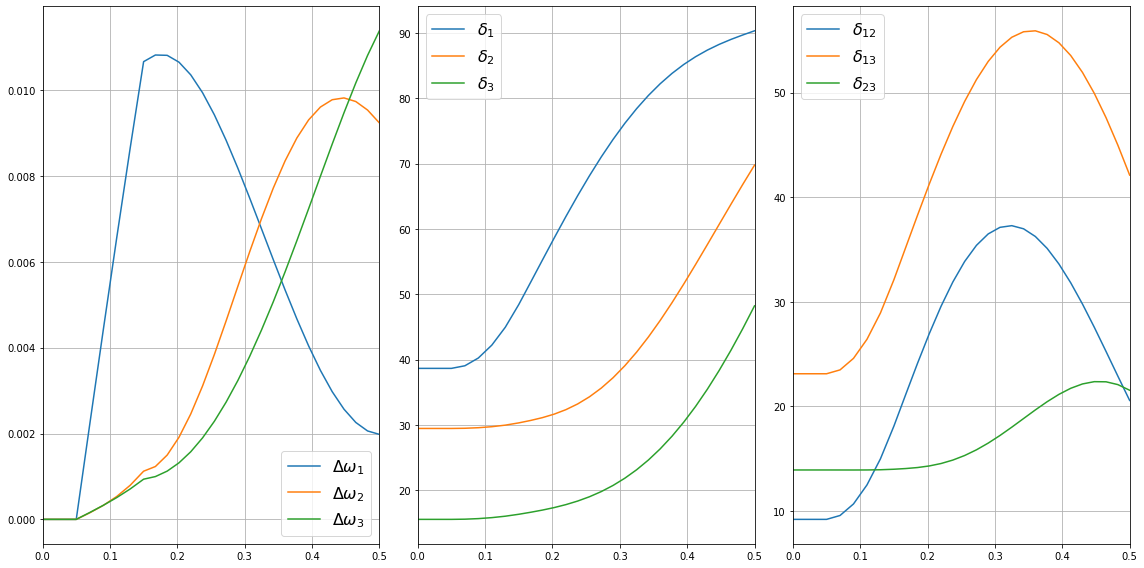


图1 仿真0.5s同步电机各状态量及相对功角变化

1. 仿真3s后各状态量，及同步电机相对功角变化曲线图

为了探究各台机组在故障切除后最终能否达到相对稳定，讲仿真时间延长至3s，从图2有如下新的发现：

1. 三台同步电机在故障切除后都无法达到相对稳定状态，而是在上下反复波动。原因是模型中没有考虑阻尼项导致摆动无法衰减。
2. 三台机组的转速是波动上升的，说明转速变化同时具有波动分量和升高分量。

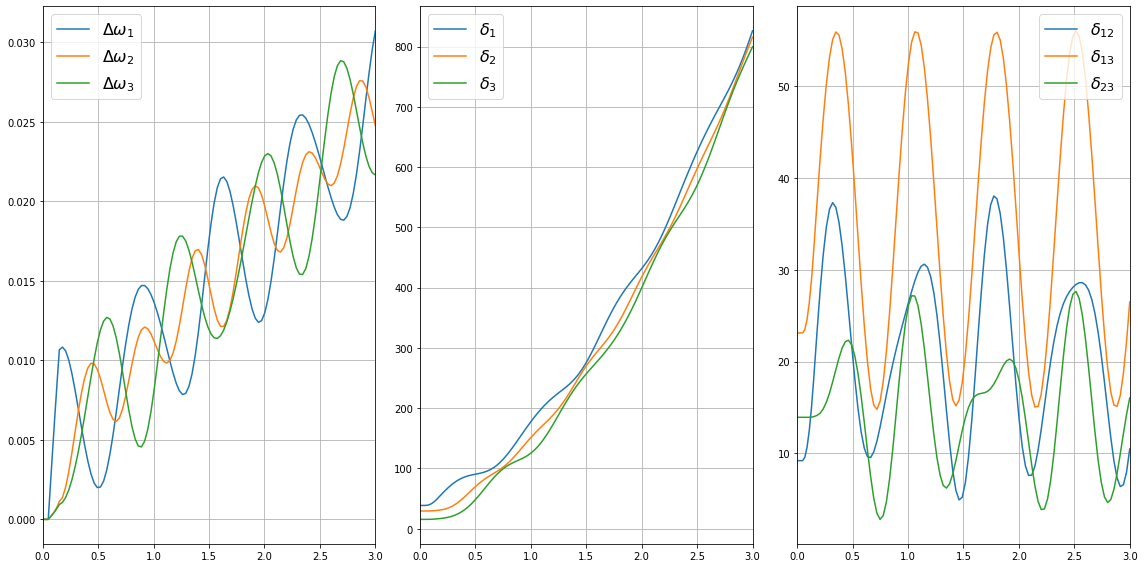


图2 仿真3s同步电机各状态量及相对功角变化

1. **程序创新点**
2. 本程序定义了通用的数据格式，使得对于任何满足格式的案例，不需要修改程序代码就能得到计算结果。
3. 本程序利用了sympy符号库的自动求导功能与优化的latex输出，免去了求解雅可比矩阵时繁重的求导任务。
4. 本程序编写了许多通用的函数，如：牛顿-拉夫逊法、生成节点导纳矩阵、求解输入阻抗及转移阻抗等，日后针对其他问题均能调用这些函数进行求解。

**附录**

一、作业202python代码

import numpy as np

def LDU(A): #LDU分解函数

size=A.shape[0]

B=A.copy() #直接的=会将地址赋值，要重新建立一个矩阵，要用copy()

L= np.zeros((size, size))

D= np.zeros((size, size))

U= np.zeros((size, size))

D[0][0]=B[0][0]

for i in range(size):

L[i][i] = 1

U[i][i] = 1

for i1 in range(size):

for i2 in range(i1,size):

for j in range(i1+1,size):

if(i2==i1):

U[i2][j]=B[i2][j]/D[i2][i2]

if(i2>i1):

B[i2][j]=B[i2][j]-B[i2][i1]\*U[i1][j]

if(i2==i1+1):

D[i2][i2] = B[i2][i2]

if (i2>i1):

L[i2][i1]=B[i2][i1]/D[i1][i1]

return L,D,U

def LDU\_store(A): #三角存储检索，不是实际的坐标

L,D,U=LDU(A)

size=A.shape[0]

L\_value=[]

L\_row=[]

L\_column=[]

D\_value=[]

U\_value=[]

U\_row=[]

U\_column=[]

for i in range(1,size):

for j in range(i):

if(L[i][j]!=0):

L\_value.append(L[i][j])

L\_row.append(i)

L\_column.append(j)

for i in range(size-1):

for j in range(i+1,size):

if(U[i][j]!=0):

U\_value.append(U[i][j])

U\_row.append(i)

U\_column.append(j)

for i in range(size):

D\_value.append(D[i][i])

return (L\_value,L\_row,L\_column),(D\_value),(U\_value,U\_row,U\_column)

def front(L,b): #前代,用LDU系数存储

size=b.shape[0]

z=np.zeros((size,1))

for i in range(len(b)):

sum = 0

if L[1].count(i)!=0:

for j in range(L[1].index(i),L[1].index(i)+L[1].count(i)):

sum+=L[0][j]\*z[L[2][j],0]

z[i,0]=b[i]-sum

return z

def front1(L,b): #前代，没用LDU系数存储

size = b.shape[0]

z=np.zeros((size,1))

sum\_list=[]

for i in range(len(b)):

sum=0

for j in range(i):

if(i>0):

sum+=L[i][j]\*z[j,0]

sum\_list.append(sum)

z[i,0] = b[i,0]-sum\_list[i]

return z

def mid(D,z): #规格化，用LDU系数存储

size = z.shape[0]

y = np.zeros((size,1))

for i in range(len(D)):

y[i,0]=z[i,0]/D[i]

return y

def mid1(D,z): #规格化，没用LDU系数存储

size = z.shape[0]

y = np.zeros((size,1))

for i in range(D.shape[0]):

y[i,0]=z[i,0]/D[i][i]

return y

def back(U,y): #回代，用LDU系数存储

size=y.shape[0]

x=np.zeros((size,1))

for i in range(size):

sum = 0

if U[1].count(size-i-1)!=0:

for j in range(U[1].index(size-i-1),U[1].index(size-i-1)+U[1].count(size-i-1)):

sum+=U[0][j]\*x[U[2][j],0]

x[size-i-1,0]=y[size-i-1,0]-sum #现在x是从x[size-1]到x[0]

x\_reverse=np.zeros((size,1))

for i in range(size):

x\_reverse[i,0]=x[size-1-i,0]

return x\_reverse

def back1(U,y): #回代，没用LDU系数存储

size=y.shape[0]

x = np.zeros((size, 1))

sum\_list=[]

for i in range(U.shape[0]):

sum=0

for j in range(i):

if(i>0):

sum+=U[size-i-1][size-j-1]\*x[size-j-1,0]

sum\_list.append(sum)

x[size-i-1,0] = y[size-i-1,0]-sum\_list[i]

x\_reverse = np.zeros((size, 1))

for i in range(size):

x\_reverse[i,0] = x[size - 1 - i,0]

return x\_reverse

def solve(A,b): #求解AX=b

L,D,U=LDU\_store(A)

X=back(U,mid(D,front(L,b)))

return X

def solve\_Sparse\_vector(A,b,x\_index): #稀疏向量法，x\_index为所需求解的特定变量

L,D,U=LDU(A)

y=mid1(D, front1(L, b))

list\_delete\_x=[]

for i in range(x\_index,len(b)):

if not any(U[x\_index-1:i,i]): #若第i个元素和x\_index无关，则U[x\_index-1:i][i]应全为0;

list\_delete\_x.append(i)

for i in list\_delete\_x:

U = np.delete(U, i, axis=0)

U = np.delete(U, i, axis=1)

y = np.delete(y, i, axis=0)

for i in range(x\_index-1):

U = np.delete(U, 0, axis=0) #每次循环都删第一行第一列，循环后则把原矩阵的前k行k列都删去了

U = np.delete(U, 0, axis=1)

y = np.delete(y, 0, axis=0)

x=back1(U,y)

return x[0]

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]])

b=np.array([[4],[4],[4],[4],[4],[4],[4],[4],[5],[5],[5],[5]])

print("解得的X为：")

print(solve(A,b))

print("需要求解的特定变量的值为：")

print(solve\_Sparse\_vector(A,b,3))

#solve\_Sparse\_vector函数的第三个参数则为需求解的某个特定变量的下标；如需求x3，则设置为3

二、作业203python代码

import numpy as np

import matplotlib.pyplot as plt

import sympy as sy

import pandas as pd

# 定义欧拉法、隐式梯形法、改进欧拉法的函数

def calculate\_f\_xy(derivative, x=None, y=None):

'''根据f(x,y)参数个数采用不同赋值'''

if x and y:

return None

elif 'x' in derivative.\_\_code\_\_.co\_varnames:

dy\_dx = derivative(x=x)

elif 'y' in derivative.\_\_code\_\_.co\_varnames:

dy\_dx = derivative(y=y)

else:

dy\_dx = derivative(x=x, y=y)

def base\_euler(derivative, n=50, limit=(0, 1), begin=(0, 1)):

'''

欧拉法求微分方程

Parameters

----------

@derivative 导数dy/dx

@n 步长

@limit 函数区间（start, end）

@begin 初值（x0, y0）

'''

step = (limit[1] - limit[0]) / n

x = np.linspace(limit[0], limit[1], n)

y = np.full(n, begin[1], dtype=np.float32)

for i in range(1, n):

y[i] = y[i - 1] + step \* derivative(y[i - 1])

return x, y

def improve\_euler(derivative, n=50, limit=(0, 1), begin=(0, 1)):

'''

改进欧拉法求微分方程

Parameters

----------

@derivative 导数dy/dx

@n 步长

@limit 函数区间（start, end）

@begin 初值（x0, y0）

'''

step = (limit[1] - limit[0]) / n

x = np.linspace(limit[0], limit[1], n)

y = np.full(n, begin[1], dtype=np.float32)

for i in range(1, n):

y[i] = y[i - 1] + step / 2 \* (derivative(y[i - 1]) +

derivative(y[i - 1] + step \* derivative(y[i - 1])))

return x, y

def implicit\_trapezoidal(derivative, derivative\_again, n=50, limit=(0, 1), begin=(0, 1), e=0.00001):

'''

隐式梯形法求微分方程

Parameters

----------

@derivative 导数dy/dx

@derivative\_again d(dy/dx)/dy

@n 步长

@limit 函数区间（start, end）

@begin 初值（x0, y0）

@e 收敛判据|y'-y| < e

'''

step = (limit[1] - limit[0]) / n

x = np.linspace(limit[0], limit[1], n)

y = np.full(n, begin[1], dtype=np.float32)

for i in range(1, n):

nocoverage = True

while(nocoverage):

F\_y = y[i] - y[i - 1] - step / 2 \* \

(derivative(y[i - 1]) + derivative(y[i]))

dF\_dy = 1 - step / 2 \* derivative\_again(y[i])

y[i] -= F\_y / dF\_dy

F\_y\_iter = y[i] - y[i - 1] - step / 2 \* \

(derivative(y[i - 1]) + derivative(y[i]))

nocoverage = False if abs(F\_y\_iter - F\_y) < e else True

return x, y

# 定义输入的导数、步长、函数定义域、原函数

# sympy定义x、y数学符号

x, y = sy.symbols('x y')

# 用x、y定义f(x,y)

deriv = -20 \* y

deriv\_2 = sy.lambdify(y, sy.diff(deriv, y), "numpy")

deriv = sy.lambdify(y, deriv, "numpy")

ESS = pd.DataFrame(columns=['Euler', 'Improve Euler', 'Implicit Trapezoidal'])

ESS.index.name = 'N'

# 图形绘制

fig, axes = plt.subplots(2, 2, figsize=(20, 10))

N = [10, 15, 20, 100]

# titles = [$'Steps is 100'$, $'Steps is 50'$, $'Steps is 20'$, $'Steps is 10'$]

for ax, n in zip(axes.flatten(), N):

n, limit = n, (0, 1)

x\_r = np.linspace(limit[0], limit[1], n)

y\_r = sy.lambdify(x, np.e\*\*(-20 \* x), "numpy")(x\_r)

x\_b, y\_b = base\_euler(deriv, limit=limit, n=n)

x\_i, y\_i = improve\_euler(deriv, limit=limit, n=n)

x\_it, y\_it = implicit\_trapezoidal(deriv, deriv\_2, limit=limit, n=n)

ax.plot(x\_r, y\_r, '-g', label='original')

ax.plot(x\_b, y\_b, '--b', label='euler')

ax.plot(x\_i, y\_i, '-.r', label='improve euler')

ax.plot(x\_it, y\_it, '-.y', label='implicit trapezoidal')

ax.set\_title(f"Step is {1/n:.3f}", fontsize=20)

ax.legend(loc='best', prop={'size': 16})

# 三种方法的误差平方和

ESS.loc[str(n), 'Euler'] = sum((y\_r - y\_b)\*\*2)

ESS.loc[str(n), 'Improve Euler'] = sum((y\_r - y\_i)\*\*2)

ESS.loc[str(n), 'Implicit Trapezoidal'] = sum((y\_r - y\_it)\*\*2)

plt.tight\_layout()

plt.show()

# plt.savefig('differential\_solve\_graph.png', bbox\_inches='tight')

三、作业302python主函数代码

import numpy as np

import pandas as pd

import sympy as sy

import time

from scipy.linalg import solve

import matplotlib.pyplot as plt

from impedance import Z\_value, alpha,branch\_data\_for\_imp\_SC,branch\_data\_for\_imp\_afterCut

import Y

def subs(func, value):

'''将表达式用value字典替换'''

if type(func) in (int,float):

return func

elif callable(func):

return func(\*value.values())

else:

return func.subs(value)

def Pe(E=None, Delta=None, Z\_value=None, Alpha=None):

'''三机系统Pe方程'''

Pe\_list = []

for i in range(3):

j,k = {0,1,2} - {i}

pe = E[i]\*\*2/Z\_value[i,i]\*sy.sin(Alpha[i,i])+E[i]\*E[j]/Z\_value[i,j]\*sy.sin((Delta[i]-Delta[j])\*sy.pi/180-Alpha[i,j])+E[i]\*E[k]/Z\_value[i,k]\*sy.sin((Delta[i]-Delta[k])\*sy.pi/180-Alpha[i,k])

Pe\_list.extend(pe)

return Pe\_list

def newton\_laphson(Functions, init\_values, e=0.00001):

'''

牛顿拉夫逊法

parameters

==========

functions 多个形如f(x,y,z,...)=0的待迭代函数的迭代类型（向量）

init\_values 函数待求解变量的初始值,其中元素个数与所有函数的总变量个数相同

h 步长

e 收敛满足max(|F(n+1)-Fn|) < e

'''

#初始化雅可比矩阵

iter\_keys, iter\_values = np.array(list(init\_values.keys())),np.array(list(init\_values.values()))

Jacobian = sy.diff(Functions, iter\_keys).transpose()

Functions = sy.lambdify(all\_var, Functions,'numpy')

nocoverage = True

while(nocoverage):

#前一次所有faunction的值构成的array

Functions\_values = np.array(Functions(\*iter\_values))

Jacobian\_values = Jacobian.subs(dict(zip(iter\_keys,iter\_values)))

#将雅可比矩阵转成np.float32类型，否则无法求解

Jacobian\_values = np.array(Jacobian\_values.tomatrix(),dtype=np.float32)

iter\_values -= solve(Jacobian\_values,Functions\_values)

Functions\_values\_again = np.array(Functions(\*iter\_values))

nocoverage = False if max(abs(Functions\_values\_again-Functions\_values)) < e else True

return dict(zip(iter\_keys, iter\_values))

def hiding\_trapezium(derivative, n=50, limit=(0,1), e=0.0001):

step = (limit[1] - limit[0]) / n

init\_values = derivative.pop('init\_values')

#协同运算但不使用牛顿拉夫逊法的量

co\_vars,co\_init\_values = derivative.pop('co\_vars'),derivative.pop('co\_init\_values')

#生成定义域内的时间序列

t = np.linspace(limit[0], limit[1], n+1)

all\_init\_values = init\_values+co\_init\_values

#将所有初始值连接成一整个字典

iter\_values = dict(zip((\*derivative,\*co\_vars), all\_init\_values))

all\_iter\_values = np.full((n+1,len(all\_init\_values)), all\_init\_values, dtype=np.float32).T

all\_iter\_values = dict(zip(iter\_values.keys(),all\_iter\_values))

for i in range(1, n+1):

print(f'\r第[{i}]步',end='')

new\_derivative = {key:key-subs(key,iter\_values)-step/2\*(value+subs(value,iter\_values)) \

for key,value in derivative.items()}

#将derivative剩下的元素与co\_vars对应的值转换为array

Functions = np.array(list(new\_derivative.values()) + list(co\_vars.values()))

iter\_values = newton\_laphson(Functions, iter\_values, e=e)

for key,value in iter\_values.items():

all\_iter\_values[key][i] = value

return t,all\_iter\_values

#读取节点分支数据

file\_name = 'data/'

Node\_data = np.load(f"{file\_name}Node\_data.npy")

branch\_data = np.load(f"{file\_name}branch\_data.npy")

trans\_data = np.load(f"{file\_name}trans\_data.npy")

E = Node\_data[np.argwhere(Node\_data[:,1]==2),2]

#初始化参数与仿真

PM1, PM2, PM3 = 1.5, 1, 3

omegaB = 18000

x, y, t = sy.symbols('x,y,t')

dw1,dw2,dw3 = sy.symbols('\Delta\omega\_1,\Delta\omega\_2,\Delta\omega\_3')

pe1,pe2,pe3 = sy.symbols('P\_e1,P\_e2,P\_e3')

dt1,dt2,dt3= sy.symbols('\delta\_1, \delta\_2, \delta\_3')

all\_var = (dw1,dw2,dw3,dt1,dt2,dt3,pe1,pe2,pe3)

Delta = (dt1,dt2,dt3)

#对节点支路数据处理，获取短路后与故障切除后输入阻抗、转移阻抗

SC\_branch = branch\_data\_for\_imp\_SC(branch\_data,Node\_data,trans\_data)

after\_branch = branch\_data\_for\_imp\_afterCut(branch\_data,Node\_data,trans\_data)

z\_value\_SC = Z\_value(SC\_branch, Node\_data)

z\_value\_after\_cut = Z\_value(after\_branch, Node\_data)

alpha\_SC = alpha(SC\_branch, Node\_data)

alpha\_after\_cut = alpha(after\_branch, Node\_data)

#稳态开始

deriv\_dict\_start= {dw1:0,dw2:0,dw3:0,dt1:omegaB\*dw1,dt2:omegaB\*dw2,dt3:omegaB\*dw3,

'co\_vars':{pe1:pe1-1.5,pe2:pe2-1,pe3:pe3-3},

'co\_init\_values':(1.5,1,3),

'init\_values':(0,0,0,38.64149,29.44408,15.52411)}

x\_stable, y\_stable = hiding\_trapezium(deriv\_dict\_start, limit=(0,0.05), n=1)

print(y\_stable)

#短路开始

pe1\_expr, pe2\_expr, pe3\_expr = Pe(E,Delta, z\_value\_SC, alpha\_SC)

short\_init\_values = [i[-1] for i in y\_stable.values()]

short\_dict = dict(zip(all\_var, short\_init\_values))

Pe\_short = [float(subs(pe1\_expr,short\_dict)),

float(subs(pe2\_expr,short\_dict)),

float(subs(pe3\_expr,short\_dict))]

deriv\_dict\_short = {dw1:(PM1-pe1)/10,dw2:(PM2-pe2)/7,dw3:(PM3-pe3)/15,

dt1:omegaB\*dw1,dt2:omegaB\*dw2,dt3:omegaB\*dw3,

"co\_vars":{pe1:pe1-pe1\_expr,pe2:pe2-pe2\_expr,pe3:pe3-pe3\_expr},

'co\_init\_values':Pe\_short,

'init\_values':short\_init\_values[:-3]}

x\_short,y\_short = hiding\_trapezium(deriv\_dict\_short, limit=(0.05,0.15), n=5)

print(y\_short)

#故障切除

pe1\_expr\_cut, pe2\_expr\_cut, pe3\_expr\_cut = Pe(E,Delta, z\_value\_after\_cut, alpha\_after\_cut)

cut\_init\_values = [i[-1] for i in y\_short.values()]

cut\_dict = dict(zip(all\_var, cut\_init\_values))

Pe\_cut = [float(subs(pe1\_expr\_cut,cut\_dict)),

float(subs(pe2\_expr\_cut,cut\_dict)),

float(subs(pe3\_expr\_cut,cut\_dict))]

deriv\_dict\_short = {dw1:(PM1-pe1)/10,dw2:(PM2-pe2)/7,dw3:(PM3-pe3)/15,

dt1:omegaB\*dw1,dt2:omegaB\*dw2,dt3:omegaB\*dw3,

"co\_vars":{pe1:pe1-pe1\_expr\_cut,pe2:pe2-pe2\_expr\_cut,pe3:pe3-pe3\_expr\_cut},

'co\_init\_values':Pe\_cut,

'init\_values':cut\_init\_values[:-3]}

x\_cut,y\_cut = hiding\_trapezium(deriv\_dict\_short, limit=(0.15,3), n=100)

x = np.concatenate((x\_stable,x\_short,x\_cut))

y = {i:np.concatenate((y\_stable[i],y\_short[i],y\_cut[i])) for i in (dt1,dt2,dt3,dw1,dw2,dw3)}

fig, axes = plt.subplots(1,3,figsize=(16,8))

Delta, graph = ((dt1,dw1),(dt2,dw2),(dt3,dw3)), (x, y)

for delta, omega in Delta:

axes[1].plot(graph[0], graph[1][delta], label='$'+str(delta)+'$')

axes[0].plot(graph[0], graph[1][omega], label='$'+str(omega)+'$')

axes[1].legend(loc='best', prop={'size':16})

axes[0].legend(loc='best', prop={'size':16})

axes[2].plot(graph[0], graph[1][dt1]-graph[1][dt2],label='$\delta\_{12}$')

axes[2].plot(graph[0], graph[1][dt1]-graph[1][dt3],label='$\delta\_{13}$')

axes[2].plot(graph[0], graph[1][dt2]-graph[1][dt3],label='$\delta\_{23}$')

axes[2].legend(loc='best', prop={'size':16})

axes[0].grid();axes[1].grid();axes[2].grid()

axes[0].set\_xlim(0, 3)

axes[1].set\_xlim(0, 3)

axes[2].set\_xlim(0, 3)

plt.tight\_layout()

plt.show()

# plt.savefig('stablity\_3s.png',bbox\_inches = 'tight')