****

**自动化测试工具手册**

**深圳市比昂芯科技有限公司**

**V2.00 2023/04/21 吕世燕**

版本修改记录

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 日期 | 作者 | 版本记录 | 审阅人 | 审阅记录 |
| 2022/03/28 | 高平洁 | V1.00 |  |  |
| 2023/04/21 | 吕世燕 | V2.00 |  |  |
|  |  |  |  |  |

目录

[一、 自动化测试流程图 4](#_Toc3545)

[二、 概述 5](#_Toc12436)

[三、 AutoTestCls()类 5](#_Toc17689)

[四、 主函数Main 6](#_Toc8923)

[五、 解析命令参数 6](#_Toc17239)

[六、 初始化 7](#_Toc20876)

[1. 构造函数\_\_init\_\_(opt)： 7](#_Toc16675)

[2. InitCaseForm()： 9](#_Toc10282)

[4. not\_check\_file\_list(self, netfile): 11](#_Toc1119)

[5. change\_suffix(file, suffix)： 11](#_Toc12644)

[七、 执行仿真 12](#_Toc27727)

[1. sim\_folder()： 12](#_Toc23163)

[2. run\_simulation\_g(index)： 12](#_Toc22049)

[八、 仿真状态统计 13](#_Toc26587)

[1. global\_log\_check()： 13](#_Toc24347)

[2. logfile\_check(file)： 15](#_Toc1221)

[3. time\_divcheck(file)： 16](#_Toc32629)

[九、 仿真结果写入Excel 17](#_Toc25223)

[十、 仿真结果对比 17](#_Toc15712)

[1. diffout()： 18](#_Toc26713)

[2. outfile\_parser(file\_name)： 19](#_Toc7632)

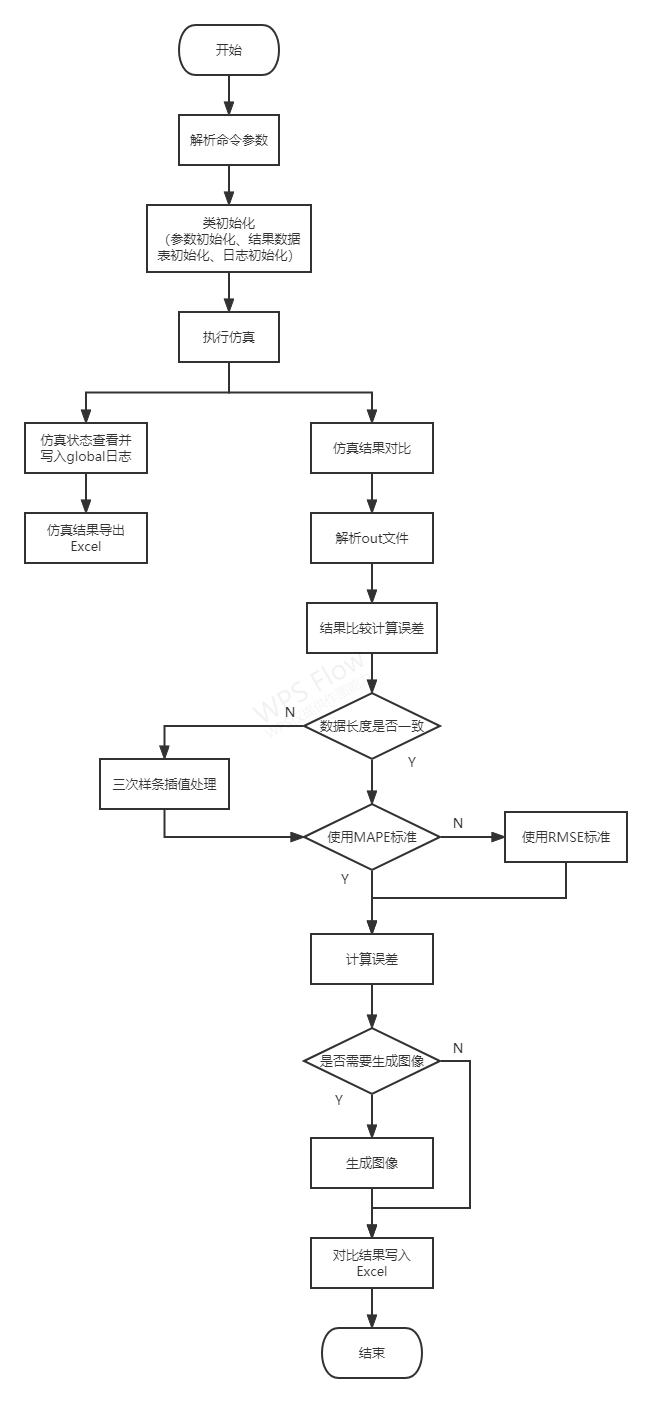
[3. calc\_error(index, outfile\_result, ref\_file\_result)： 24](#_Toc24443)

[4. save\_plot(index, out\_results\_dict, ref\_results\_dict, compare)： 30](#_Toc9396)

[5. 对比结果写入Excel 34](#_Toc25862)

[十一、 持续优化 34](#_Toc10253)

1. 自动化测试流程图



1. 概述

请大家在执行脚本前阅读 README 里面有执行命令的总结和命令的参数含义解释

脚本文件主要由两大部分组成，AutoTestCls()类和main函数。

AutoTestCls()中定义了类的对象，并做了初始化。同时还包含了所有用到的具体方法，例如logfile\_check(), outfile\_parser(), calc\_error()等。

Main函数则记录了整个脚本执行的几步操作，例如执行仿真、仿真结果对比并写入excel等。

1. AutoTestCls()类

“类”的概念，是某种类型集合的描述。比如，“人类”可以被看做一个类，然后用人类这个类可以定义出每个具体的人：你、我、他，作为其对象。在用类定义成对象时，会先调用\_\_init\_\_构造函数，以初始化对象的各类属性，比如人的名字、身高、体重等。人类除了身高体重这些属性，还具有方法，比如吃饭、睡觉等。

在本自动化测试的脚本中，构造函数以及所有类的方法归纳如下：

|  |  |
| --- | --- |
| 函数名称 | 描述 |
| def \_\_init\_\_(self, opt) | 构造函数，定义对象的基本属性 |
| def InitCaseForm(self) | 遍历测试路径下的所有文件，并给对象中的相关的文件路径赋值 |
| def del\_case\_dir | 删除当前路径下已存在的测试目录 |
| def not\_check\_file\_list | 判断case是否是需要忽略的case |
| def run\_simulation\_g(self, netfileID) | 执行仿真 |
| def sim\_folder | 多线程执行仿真 |
| def change\_suffix(self, file, suffix) | 修改文件后缀 |
| def global\_log\_check(self) | 检查生成的global log文件 |
| def logfile\_check(self, file) | 检查每个case具体的log文件 |
| def limit\_logfile\_check | 检查log文件中是否包含“This version is limited to”区分华力版本器件数限制 |
| def outfile\_parser(self, file\_name) | 解析仿真结束生成的out文件，提取并记录其中的关键数据 |
| def getCaseIndex(self, index) | 获取网表文件名称中case index的值 |
| def time\_divcheck(self): | 对比仿真时间wall time 和golden时间，超过15% 即为对比失败 |
| def calc\_error(self, index,  original\_results\_dict, new\_results\_dict) | 根据评估指标[MAPE, RMSE]对比out文件中的数据和bench文件中的out文件中的数据 |
| def save\_plot(self, index,  out\_results\_dict, ref\_results\_dict, compare) | 将out文件的数据和bench文件中out文件的数据进行画图比对 |
| def diffout(self) | 解析两份out文件，并调用calc\_error()进行计算、save\_plot()进行绘图 |
| def result\_statistics | 统计仿真失败，对比失败，仿真时间超长的case总数并打印出来 |
| def bench\_error\_data | 将仿真失败和对比失败的案例放入失败案例集中 |
| def outputTerm(self): | 打印仿真失败、对比失败、仿真时间超时的案例 |

各方法/函数的具体操作会在后续展开介绍。

1. 主函数Main

想要快速了解一个脚本文件做了什么，首先看主函数。

主函数主要由五个部分组成：

1. 解析命令参数
2. 初始化AutoTestCls类
3. 多线程执行仿真
4. 仿真状态统计
5. 仿真时间统计
6. 仿真结果写入excel
7. 仿真结果对比并写入excel

具体每个部分会结合其调用到的AutoTestCls()类中的方法进行描述。

1. 解析命令参数

argparse是python用于解析命令行参数和选项的标准模块，作用是用于解析命令行参数。我们建立了一个名为parser的解析对象，并向其中添加了需要用到的几种属性：

|  |  |
| --- | --- |
| 参数 | 描述 |
| --tp | 指定需要测试的网表路径，默认是当前所在路径 |
| --sh | 指定simulator路径，默认是btdsim |
| --savesimcsv | 指定是否需要保存仿真结果到excel文件，默认是True |
| --savediffcsv | 指定是否需要保存仿真对比结果到excel文件，默认是True |
| --savefig | 指定是否需要保存仿真结果图像，默认是True |
| --metric | 指定比对数据时使用的评判标准，默认使用MAPE标准，[RMSE, MAPE] |
| --isdelold | 是否删除tp目录下的测试集 |
| --rp | 测试集的路径，默认全部”all” |
| --cn | 单独运行指定case 例如 --cn case11 |
| --si | 当运行all测试集合需要指定测试集合为hisi还是huali,默认全部 |
| --bv | 指定btdsim的version：base、plus、rf，默认rf |

parser = argparse.ArgumentParser()

parser.add\_argument("--tp", type=str, default="./", help="the save path to test case")

parser.add\_argument("--sh", type=str, default='btdsim', help="choose simulator path")

parser.add\_argument("--savesimcsv", type=bool, default=True, help="save final simulator csv file")

parser.add\_argument("--savediffcsv", type=bool, default=True, help="save final diff csv file")

parser.add\_argument("--savefig", type=bool, default=False, help="save final plot")

parser.add\_argument("--metric", type=str, default="MAPE", help="select metrics for diff, i.e. RMSE or MAPE")

parser.add\_argument("--isdelold", type=int, default=1, help="Whether to delete the old case dir")

parser.add\_argument("--rp", type=str, default="all", help="""path to test case""")

parser.add\_argument("--cn", type=str, default="", help="case name")

parser.add\_argument("--si", type=str, default="all", help="execute case selector, all、hisi、huali")

parser.add\_argument("--bv", type=str, default="rf", help="btdsim version: base, plus, rf")

parser.add\_argument("--ccost", type=str, default=1, help="Simulatorcost Compare")

opt = parser.parse\_args()

print(opt)

属性给予opt实例，打印opt即可看见本次输入的命令包含了其中哪些参数。使用方法：

python3 testbenchcompare.py --sh btdsimT --tp ./testtmp/ --cn case1 --metric RMSE

这个例子表示，使用的simulator是btdsimT，测试路径在case1文件夹中的网表文件，并用rmse（均方根误差）标准对比数据结果差异。

python testbenchcompare.py --rp huali --bv plus --sh btdsimP

使用 btdsimP 运行huali测试集的所有case 指定版本为plus

1. 初始化

执行仿真前，我们先创建了一个AutoTestCls()实例化的对象，命名为atc。同时，将之前解析命令得到的opt作为参数传入。因此atc便拥有了AutoTestCls()中所有的方法，并且在初始化时，获得了opt传入的值。

1. 构造函数\_\_init\_\_(opt)：

类的初始化主要包括了参数初始化、日志初始化、结果数据表初始化。构造函数\_\_init\_\_()中初始化的具体参数可参见下表：

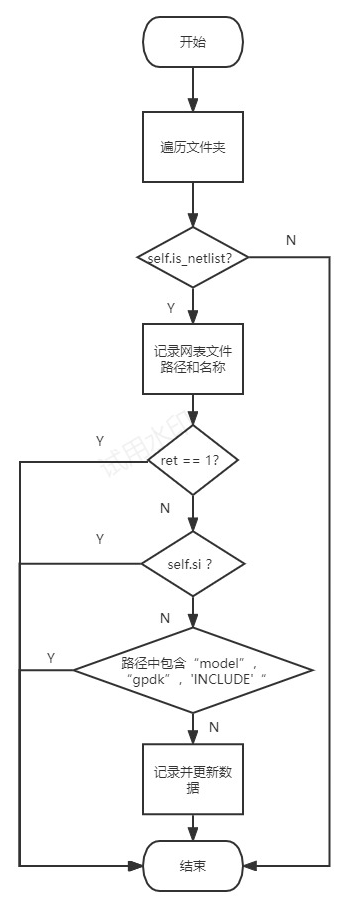
|  |  |
| --- | --- |
| 参数 | 描述 |
| self.spfile\_Num | 0 |
| self.line\_list | [] |
| self.time\_list | [] |
| self.str\_list | [] |
| self.test\_dir | './'  (注：default为 './'，可通过命令--tp传入指定路径) |
| self.ref\_filename | 'bench' |
| self.cases\_nodes\_path | 'cases\_nodes.xlsx' |
| self.sh | btdsim  (注：default: btdsim，可通过命令--sh传入指定simulator) |
| self.si | All() (注：default：all, 可通过命令 --si 传入指定运行测试集合，适用于选择all测试集中的huali 或 hisi 的case) |
| self.version | 指定btdsim版本（base、plus、rf）；默认为rf base：报错信息为：器件数超过5000的例子会判定为正确  plus：报错信息为：器件数超过100000的例子会判定为正确 rf：不做任何判定 |
| self.dir\_dict | 测试集合字典，包含所有的测试集合 |
| self.test\_dir | 测试集合的路径 |
| self.cases\_nodes\_path | cases\_nodes.xlsx 文件路径 |
| self.output\_folder | output路径，存放data\_df\_simulator & data\_df\_diff |
| self.case\_dir | Case的路径 单独跑1个case时使用 （--cn case\*） |
| self.not\_check\_case | 需要跳过不进行测试的case |
| self.autoRunlogfile | open("./output/autoRun.log", "a")  (注：如果autoRun.log文件已经存在，新的log会在已有内容之后追加；如果文件不存在，创建新文件并写入) |
| self.data\_df\_simulator | 二维矩阵，共6列。行索引为 'index'，列索引为['index', 'netFile', 'logFile', 'outFile', 'SimulatorStat', 'Simulatorcost'] |
| self.data\_df\_diff | 二维矩阵，共13列。行索引为 'index'，列索引为['index', 'spFile', 'logFile', 'outFile', 'RefoutFile', 'AnalysisType', 'SimulatorStat', 'Simulatorcost', "time\_div", "cost\_div", "outdiff", "outdiffCost", "outdiffdetail"] |
| self.is\_netlist | True/False  (注：遍历文件时具体判断并赋值) |
| self.check\_nodes\_dict | 二维数组，读取了外部excel数据。其中记录了每个case需要对比的节点 |
| self.check\_time\_dict | 二维数组，读取了外部excel数据。其中记录了每个case的golden时间 |
| self.case\_limit | 同上记录case的版本对应的限制（bace/plus） |

同时，构造函数\_\_init\_\_()还调用了InitCaseForm()，初始化文件。

1. InitCaseForm()：

该函数具体内容可参见流程图。

首先，遍历了test\_dir下所有的文件，并调用self.is\_netlist(filename) 判断文件是否为网表文件。



self.is\_netlist = lambda x: any(x.endswith(extension)

    for extension in ['.sp', '.cir', 'scs'])

如果self.is\_netlist(filename) 返回值为False，不做任何处理并继续遍历下一个文件；如果返回值为True，获取网表文件路径+名称。

netfile = os.path.join(filepath, filename)

如果ret==1 ,跳过; 否则则记录并更新相关数据（跳过忽略的case）

def not\_check\_file\_list(self, netfile):

    caseindex = netfile.split('case')[1].split('/')[0]

    #测试打通阶段略略这些case

    if f"case{caseindex}" in self.not\_check\_case:

        return 1

    else:

        return 0

如果si==huali ,判断caseindex 是否大于1000，是则跳过，否则记录并更新相关数据（只跑华力的case）

如果si==hisi ,判断caseindex是否小于1000，是则跳过， 否则记录并更新相关数据（只跑海思的case）

if self.si == "huali" and caseindex >=1000:

    continue

elif self.si == "hisi" and caseindex <=1000:

    continue

如果返回的网表文件绝对路径(netfile)中包含 ['model', 'gpdk', 'INCLUDE']，这类网表文件不进行仿真，跳过；如果不包含，则记录并更新相关数据，见下表。

self.spfile\_Num += 1

# log文件

logFile = self.change\_suffix(netfile, '.log')

# 仿真结果

SimulatorStat = 0 # 1为仿真成功，0为仿真失败，default = 0

# 仿真时间

Simulatorcost = None

# 仿真类型

AnalysisType = None

# 仿真sp得到的out文件

outFile = self.change\_suffix(netfile, '.out')

# 需要对比的out文件

ref\_path = os.path.join(os.path.dirname(outFile), self.ref\_filename)

ref\_file = os.path.join(ref\_path, os.path.basename(outFile))

# diff result

Simulatordiff = None

# diff result detail

outdiffdetail = {}

# diff time\_div

time\_div = None #时间对比是否通过

cost\_div = None #仿真时间-golden时间/golden时间（仿真时间超时百分比）

outdiffCost = None #数据对比时间

其中logFile和outFile在赋值时调用了change\_suffix(file, suffix) 函数，请参见下一小节内容。

对于需要对比的out文件，我们先假设它存在，且存放于同级目录下的bench文件夹里，名称和生成的out文件相同。

上表中的数据更新/初始化完后，需要将它们同步更新到二维矩阵data\_df\_simulator和data\_df\_diff之中，以备后续excel文件的导出。

self.data\_df\_simulator.loc[self.spfile\_Num] = [netfile, logFile, outFile, SimulatorStat,Simulatorcost]

self.data\_df\_diff.loc[self.spfile\_Num] = [netfile, logFile, outFile, ref\_file, AnalysisType,SimulatorStat, Simulatorcost, time\_div, cost\_div, Simulatordiff,outdiffCost,outdiffdetail]

1. def del\_case\_dir(self):

判断当前路径是否存在要测试的测试集文件，如果有则删除

def del\_case\_dir(self):

    case\_dir = f"{opt.tp}/{self.dir\_dict[opt.rp]}"

    delCmd = f"rm -rf {case\_dir}"

    if os.path.exists(case\_dir):

        print("INFO DEL CMD: "+delCmd)

        os.system(delCmd)

1. not\_check\_file\_list(self, netfile):

测试阶段忽略掉self.not\_check\_case集合里的case。

def not\_check\_file\_list(self, netfile):

    # netfile: /home/IC/Case\_wayne/TestBench\_1BaseCases1/>>>>>case1<<<<<</VCO/lab1\_pss\_pnoise\_btd.scs

    caseindex = netfile.split('case')[1].split('/')[0]

    #测试打通阶段略略这些case

    if f"case{caseindex}" in self.not\_check\_case:

        return 1

    else:

        return 0

1. change\_suffix(file, suffix)：

调用这个函数的目的是为了在遍历网表文件时，同时保存其log文件和out文件的路径+后缀名，以便保存到二维矩阵data\_df\_simulator和data\_df\_diff中，供后续查阅文件时能找到对应的文件路径。

# 修改文件后缀

def change\_suffix(self, file, suffix):

    if file.endswith('.sp'):

        file = file.replace('.sp', suffix)

    elif file.endswith('.cir'):

        file = file.replace('.cir', suffix)

    elif file.endswith('.scs'):

        file = file.replace('.scs', suffix)

    else:

        pass

    return file

例如，我们的网表文件名称+后缀为：'./case18/LNA\_Lab2\_Pnoise/input.scs'，我们想要保存log file的路径+名称，便可以通过如下方式调用：

netfile = './case18/LNA\_Lab2\_Pnoise/input.scs'

logFile = self.change\_suffix(netfile, '.log')

得到的返回结果：

logFile = './case18/LNA\_Lab2\_Pnoise/input.log'

1. 执行仿真

我们先在目录下创建了一个output文件夹，用于保存测试生成的global log文件、仿真状态excel文件、结果对比excel文件和生成的图像文件。如果output文件夹已存在，生成的文件会替换或继续添加到该文件夹中。

os.makedirs("output", exist\_ok=True)

在上一章初始化完成后，我们调用atc.sim\_folder()开始仿真。

1. sim\_folder()：

通过for循环，对每份网表文件进行一次仿真调用self.run\_simulation(index)。同时，会对先前在6.2节中预存的bench文件路径进行核查，如果该路径实际不存在，则重新赋值为空。

def sim\_folder(self,thread\_list):

    for index in range(1, self.spfile\_Num + 1):

        # 找case对应的bench文件结果

        ref\_file = self.data\_df\_diff.loc[index].RefoutFile

        if not os.path.exists(ref\_file):

            self.data\_df\_diff.loc[index, "RefoutFile"] = None

            self.data\_df\_diff.loc[index, "RefoutFile"] = None

        # 执行仿真

        t1 = threading.Thread(target=self.run\_simulation\_g,args=(index,))

        thread\_list.append(t1)

1. run\_simulation\_g(index)：

如下代码块为执行仿真并记录仿真时间。其中，netfile和logfile是根据6.2节中遍历完文件所记录的网表文件和日志文件路径。

RunCmd表示用self.sh的simulator对netfile进行仿真，并存入logfile日志文件。

def run\_simulation\_g(self, netfileID):

    netfile = self.data\_df\_simulator.loc[netfileID].netFile

    if netfile:

        logfile = self.data\_df\_simulator.loc[netfileID].logFile

        # 执行仿真并记录仿真时间

        if "case13/transient\_analysis/hao\_testx0\_post2.sp" in netfile :

            RunCmd = self.sh + " {} -f nutascii > {}".format(netfile, logfile)

        else:

            RunCmd = self.sh + " {} -f nutascii -m4 > {}".format(netfile, logfile)

        start = time.time()

        os.system(RunCmd)

        end = time.time()

        cost = int((end - start) \* 1000) / 1000

将上述仿真时间cost更新至二维矩阵data\_df\_simulator和data\_df\_diff中：

        self.data\_df\_simulator.loc[netfileID, "Simulatorcost"] = cost

        self.data\_df\_diff.loc[netfileID, "Simulatorcost"] = cost

1. 仿真状态统计
2. global\_log\_check()：

使用for循环，调用logfile\_check(logfile) 检查每个case的log文件，并将检查结果写入global的日志文件。

判断 tag = self.case\_limit[caseindex] (已写入”cases\_nodes.xlsx”文件夹中)

当btdsim的版本是 "base" 版本，tag =bace（器件数超5000） 和 plus（器件数超100000）

调用方法：limit\_logfile\_check(self, file);

当btdsim的版本是 "plus" 版本tag = plus（器件数超100000）

调用方法：limit\_logfile\_check(self, file);

否则调用方法：logfile\_check(self, file)

def global\_log\_check(self):

    for id in range(1, self.spfile\_Num + 1):

        spfile = self.data\_df\_simulator.loc[id].netFile

        logfile = self.data\_df\_simulator.loc[id].logFile

        if logfile:

            caseindex = self.getCaseIndex(id)

            tag = self.case\_limit[caseindex]

            if self.version == "base":

                if tag == "base" or tag == "plus":

                    stat = self.limit\_logfile\_check(logfile)

                else:

                    stat = self.logfile\_check(logfile)

            elif self.version == "plus":

                if tag == "plus":

                    stat = self.limit\_logfile\_check(logfile)

                else:

                    stat = self.logfile\_check(logfile)

            else:

                stat = self.logfile\_check(logfile)

limit\_logfile\_check(self, file)方法检查log文件是否包含 'This version is limited to' 关于器件数的报错，如有返回1(超5000器件数报错符合预期) 否则返回0

def limit\_logfile\_check(self, file):

    if os.path.exists(file):

        try:

            with open(file) as f:

                for line in f.readlines():

                    if 'This version is limited to' in line:

                        return 1

        except:

            try:

                with open(file, encoding='latin-1') as f:

                    for line in f.readlines():

                        if 'This version is limited to' in line:

                            return 1

            except:

                print("error decode:" + file)

    return 0

logfile\_check(self, file)方法检查log文件中是否包含 'SIMULATION is completed successfully' 仿真成功，如有返回1，否则返回0

def logfile\_check(self, file):

    if os.path.exists(file):

        try:

            with open(file) as f:

                for line in f.readlines():

                    if 'SIMULATION is completed successfully' in line:

                        return 1

        except:

            try:

                with open(file, encoding='latin-1') as f:

                    for line in f.readlines():

                        if 'SIMULATION is completed successfully' in line:

                            return 1

            except:

                print("error decode:" + file)

    return 0

同时，将仿真结果更新到二维矩阵data\_df\_simulator和data\_df\_diff中。

if stat:

    SimulatorStat = 1

    # self.get\_time(id)

    self.autoRunlogfile.write(' '.join([spfile, 'YES']))

else:

    SimulatorStat = 0

    self.data\_df\_simulator.loc[id, "Simulatorcost"] = None

    self.data\_df\_diff.loc[id, "Simulatorcost"] = None

    self.autoRunlogfile.write(' '.join([spfile, 'NO']))

self.data\_df\_simulator.loc[id, "SimulatorStat"] = SimulatorStat

self.data\_df\_diff.loc[id, "SimulatorStat"] = SimulatorStat

1. logfile\_check(file)：

首先检查log文件是否真是存在于路径中，确认log文件存在后，使用try except方式打开日志文件，并处理无法译解的乱码问题。

在try代码块中，打开文件默认的编码方式是encoding='utf-8'，如果捕获到异常，在except代码块中，会继续try encoding='latin-1'的编码方式，如果仍然捕获到异常，就打印报错，告知无法译解文件。

def logfile\_check(self, file):

    if os.path.exists(file):

        try:

            with open(file) as f:

                for line in f.readlines():

                    if 'SIMULATION is completed successfully' in line:

                        return 1

        except:

            try:

                with open(file, encoding='latin-1') as f:

                    for line in f.readlines():

                        if 'SIMULATION is completed successfully' in line:

                            return 1

            except:

                print("error decode:" + file)

    return 0

1. time\_divcheck(file)：

对仿真成功的case逐一进行时间比较，当仿真时间超过golden时间的15%，则对比失败，对比结果和百分比记录到 df\_diff.loc 中，并将对比失败的case打印。

def time\_divcheck(self):

    for id in range(1,self.spfile\_Num+1):

        if self.data\_df\_diff.loc[id].SimulatorStat:

            simulator\_cost = self.data\_df\_diff.loc[id, "Simulatorcost"]

            caseindex = self.getCaseIndex(id)

            stand\_cost = self.check\_time\_dict[caseindex]

            diff\_cost = (simulator\_cost - stand\_cost)/ stand\_cost \*100

            tag = self.case\_limit[caseindex]

            self.data\_df\_diff.loc[id,"cost\_div"] = '%.3f'%diff\_cost+"%"

            if diff\_cost<= 15 :

                self.data\_df\_diff.loc[id,"time\_div"] = 1

            elif diff\_cost>15:

                if self.version == "base":

                    if tag == "base" or tag == "plus":

                        continue

                    else:

                        self.data\_df\_diff.loc[id, "time\_div"] = 0

                elif self.version == "plus":

                    if tag == "plus":

                        continue

                    else:

                        self.data\_df\_diff.loc[id, "time\_div"] = 0

                else:

                    self.data\_df\_diff.loc[id, "time\_div"] = 0

1. 仿真结果写入Excel

将二维矩阵data\_df\_simulator记录的数据导出写入Excel，方便直接查看仿真结果。

date\_str读取了当前系统时间，并将它拼接在了文件名称中以便在有多份仿真结果时能区分每份文件的生成时间。

opt.savesimcsv为解析得到的命令参数，默认为True，表示默认生成仿真结果文件，如果不需要可以在输入命令时改为 --savesimcsv False。

date\_str = time.strftime("%m%d%H%M%S", time.localtime())

if opt.savesimcsv:

    # 将仿真结果写入excel

    writer1 = pd.ExcelWriter(f'{atc.output\_folder}/data\_df\_simulator\_{date\_str}.xlsx')

    atc.data\_df\_simulator.to\_excel(writer1)

    writer1.save()

1. 仿真结果对比

类似的，命令参数 --savediffcsv 默认值为True，会自动跟bench文件中的数据进行对比并生成报告。我们通过if条件判断opt.savediffcsv的值，如果为真，则调用diffout()函数进行数据比对。

1. diffout()：

使用for循环，对比每一份网表的仿真结果与bench文件预存的仿真结果。

对于每一个case，根据二维矩阵data\_df\_diff存入的数据，并记录开始和结束时间

首先判断case是否是base版本或plus版本中预期因器件数仿真失败的case，若是则直接将data\_df\_diff 中的”outdiff”值复制为”PASS”并跳过数据对比

def diffout(self):

    for netId in range(1, self.spfile\_Num + 1):

        start = time.time()

        caseindex = self.getCaseIndex(netId)

        tag = self.case\_limit[caseindex]

        if self.version == "base" and self.data\_df\_diff.loc[netId].SimulatorStat:

            if tag == "base" or tag == "plus":

                self.data\_df\_diff.loc[netId, "outdiff"] = "PASS"

                continue

        elif self.version == "plus" and self.data\_df\_diff.loc[netId].SimulatorStat:

            if tag == "plus":

                self.data\_df\_diff.loc[netId, "outdiff"] = "PASS"

                continue

        else:

            pass

然后判断一下仿真结果是否为真并且out文件是否实际存在于路径中。

if self.data\_df\_diff.loc[netId].SimulatorStat & os.path.exists(self.data\_df\_diff.loc[netId].outFile):

上述条件满足后，我们先取值并赋值给变量，以便后续对比操作：

outfile = self.data\_df\_diff.loc[netId].outFile #仿真生成的out文件

# bench文件

ref\_file = self.data\_df\_diff.loc[netId].RefoutFile #bench中的golden out 文件

初始化变量，用于存储对比结果：

compare = None

com\_result = str({})

判断需要对比的两份out文件是否实际存在于路径中：

if os.path.exists(outfile) and os.path.exists(ref\_file):

调用outfile\_parser(file\_name) 函数（10.2节详细描述该函数）解析out文件，并获得分析类型对应节点的字典和分析类型名称的集合：

results\_1\_dict, plotname\_arr1 = self.outfile\_parser(outfile)

results\_2\_dict, plotname\_arr2 = self.outfile\_parser(ref\_file)

调用calc\_error(index, outfile\_result, ref\_file\_result) 函数（10.3节详细描述该函数）计算误差并将结果赋值给compare和com\_result：

compare, com\_result = self.calc\_error(netId, results\_1\_dict, results\_2\_dict)

如果 --savefig 参数指定了保存图片（default = True），则调用save\_plot() 函数（10.4节详细描述该函数）进行画图：

if opt.savefig:

    self.save\_plot(netId, results\_1\_dict, results\_2\_dict, compare)

将上面记录下来的plotname\_arr1放入set，删除重复数据，并存成list数据结构赋值给AnalysisTypes：

AnalysisTypes = list(set(plotname\_arr1))

将上述结果更新至二维矩阵data\_df\_diff以便导出成excel：

self.data\_df\_diff.loc[netId, "AnalysisType"] = ";".join(AnalysisTypes)

self.data\_df\_diff.loc[netId, "outdiff"] = compare

self.data\_df\_diff.loc[netId, "outdiffdetail"] = com\_result

记录对比时间到data\_df\_diff中，并打印出来

cost = end-start

self.data\_df\_diff.loc[netId, "outdiffCost"] = cost

print(f"""INFO: case{caseindex}: \n    startTime: {start}\n    endTime: {end}\n    diffCost: {cost}""")

1. outfile\_parser(file\_name)：

用 'latin-1' 编码方式打开传入的文件，并通过python中readlines() 方法读取整个文件并返回列表，一行为一个元素。

fo = open(file\_name, "r", encoding='latin-1')

output\_file = fo.readlines()

定义一些变量，用于记录遍历文件获得的数据：

|  |  |
| --- | --- |
| 变量 | 描述 |
| nodelist | []，  存储从文件中读取的同一个分析类型下的所有行，在判断读到一个新的分析类型时被重置为空列表 |
| nodelists | {}，  实质上是一个二维数组，其中的每个元素是一个分析类型，也是一个list，即为上述nodelist |
| notenum | 从1开始的计数器，每读到一个新的分析类型时，计数器+1 |
| plotname\_arr | 分析类型名称的集合array |

nodelist和nodelists的关系可以抽象如下：

nodelists = { #分析类型的集合

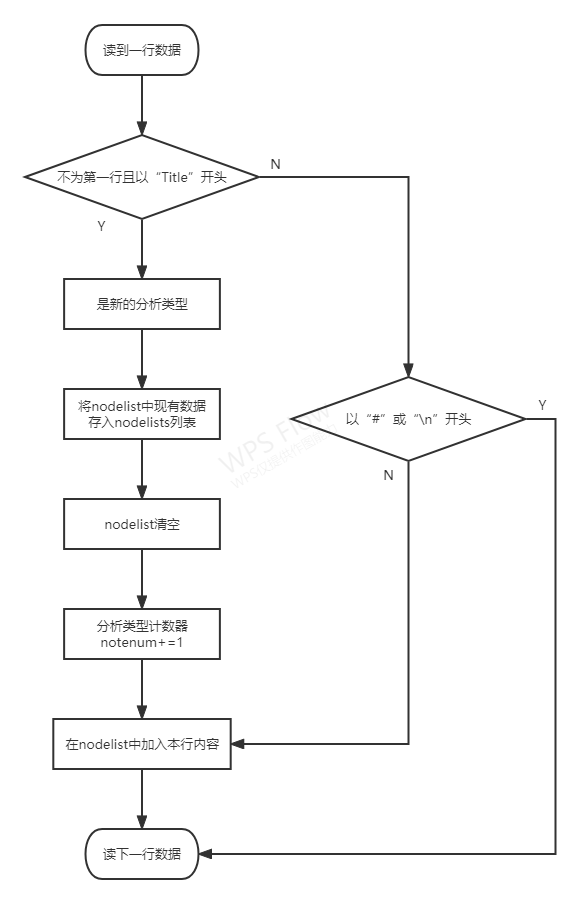
nodelist = [value, value, value], #分析类型A中值的集合

nodelist = [value, value, value], #分析类型B中值的集合

nodelist = [value, value, value] #分析类型C中值得集合

}

使用for循环对文件进行逐行遍历，逻辑流程如下图：



for i, line in enumerate(output\_file):

    if i != 0 and line.startswith('Title'):

        nodelists.append(nodelist)

        notenum += 1

        nodelist = []

        nodelist.append(line)

    elif line.startswith('#') or line.startswith(' \n'):

        pass

    else:

        nodelist.append(line)

nodelists.append(nodelist)

assert len(nodelists) == notenum

需要注意的是，for循环遍历完文件最后一行并未将nodelist中内容添加到nodelists中，所以在退出for循环后需要单独进行添加。同时，notenum为分析类型计数器，nodelists分析类型的集合，所以notenum的value应与nodelists长度相等，如果不等则assert触发异常。

创建nodelistsresult用于存储每个节点和其对应的数据用于返回。这里使用了python中collections模块自带的orderedDict来替代普通的dict，因为普通的字典是按照hash来存储的，orderedDict则是根据放入元素的先后顺序进行排序，以确保存入字典的键值对有序，以便取出plotname的第一个键值对，后续画图需要用来作为x轴。

nodelistsresult = collections.OrderedDict()

使用for循环对nodelists（分析类型的集合）中的每一个分析类型进行处理。 在处理每个分析类型时，先创建如下变量：

|  |  |  |
| --- | --- | --- |
| 变量 | 赋值 | 描述 |
| number\_of\_variables | int(titles[4].split(':', 1)[1]) | titles实际为之前nodelists中的每一个nodelist，是从out文件解析出来的一行一行的数据。titles[4]对应的out文件内容类似此格式：“No. Variables: 107”，通过左边代码赋值便可得到number\_of\_variables = 107 |
| plotname | titles[2].split(':')[-1].split('\n')[0] | titles[2]对应的out文件内容类似此格式：“Plotname: Periodic Steady State Analysis”，通过左边代码赋值便可得到plotname = Periodic Steady State Analysis  并将每次得到的plotname添加到plotname\_arr当中。 |
| nodelistsresult[plotname] | collections.OrderedDict() | 创建存放plotname和其对应数据的有序字典 |

取nodename节点名和其对应的单位：

for variable in titles[8:8 + number\_of\_variables]: #用于存储该plotname下所有nodename和其对应的单位

# 取节点名

    var\_name = variable.split('\t', -1)[2] #e.g. var\_name 得到的值为'time'

# 取节点单位

    var\_unit = variable.split('\t', -1)[3].split('\n')[0] #e.g. var\_unit得到的值为's'

# 节点名和单位之间用----拼接，赋给variable\_name\_unit

    var\_name\_unit = var\_name + "----" + var\_unit #e.g. var\_name\_unit得到的值为'time---s'

    variable\_name\_unit.append(var\_name\_unit)

获得每个nodename下的所有值，一些细节将以注释的形式在代码中批注：

results = collections.OrderedDict() #用于存储每个节点对应的数据，key为nodename

variable\_index = 0

for variable in variable\_name\_unit:

results[variable] = [] #每个nodename初始化一个空list

for value in titles[8 + number\_of\_variables + 1:]:

    if variable\_index != number\_of\_variables - 1:#除最后一个节点以外的所有节点

        st = value.split('\t', -1)[-1]

        if st.startswith("FAIL"):

            st = "0"

        try:

            vlu = eval(st)

        except:

            st = value.split(' ', -1)[-1]

        vlu = eval(st)

#eval函数能识别字符串中的表达式并将其转换为相应的类型

#如果是虚数，out文件中会有两个值，我们只取实部

    if type(vlu) == tuple:

        vlu\_abs = abs(complex(vlu[0], vlu[1]))

        if vlu\_abs < self.min\_threshold:

            results[variable\_name\_unit[variable\_index]].append(0)

        else:

            results[variable\_name\_unit[variable\_index]].append(vlu\_abs)

    else:

        if vlu < self.min\_threshold:

            results[variable\_name\_unit[variable\_index]].append(0)

        else:

            results[variable\_name\_unit[variable\_index]].append(vlu)

    variable\_index += 1

else:

    st = value.split('\t', -1)[-1].split('\n')[0]

    # 如果out文件中出现Fail字符串，将其转换成0

    if st.startswith("FAIL"):

        st = "0"

    vlu = eval(st)

    if type(vlu) == tuple:

        if vlu[0] < self.min\_threshold:

            results[variable\_name\_unit[variable\_index]].append(0)

        else:

            results[variable\_name\_unit[variable\_index]].append(vlu[0])

    else:

        if vlu < self.min\_threshold:

            results[variable\_name\_unit[variable\_index]].append(0)

        else:

            results[variable\_name\_unit[variable\_index]].append(vlu)

variable\_index = 0

至此，results这个有序字典存储了每个分析类型中所有nodes和其对应的values，最后将每个分析类型的results添加到nodelistsresults[plotname]中，即存储了所有分析类型下所有节点的数据：

nodelistsresult[plotname].append(results)

最终将nodelistsresult和plotname\_arr返回，用于后续对比计算：

return nodelistsresult, plotname\_arr

1. calc\_error(index, outfile\_result, ref\_file\_result)：

参数outfile\_result和ref\_file\_result为经过outfile\_parser处理后得到的有序dictionary。在进行outfile数据和ref\_file数据对比前，我们先获取plotnames和nodenames。

plotname\_nodes = []

plotnames = set()

# 获取所有的plotnames

for item in original\_results\_dict:

    plotnames.add(item)

print("Total plotnames: {}".format(plotnames))

for plotname in plotnames:

    node\_dict = original\_results\_dict[plotname]

    # 获取所有的nodename

    if type(node\_dict) == list:

        nodes = node\_dict[0].keys()

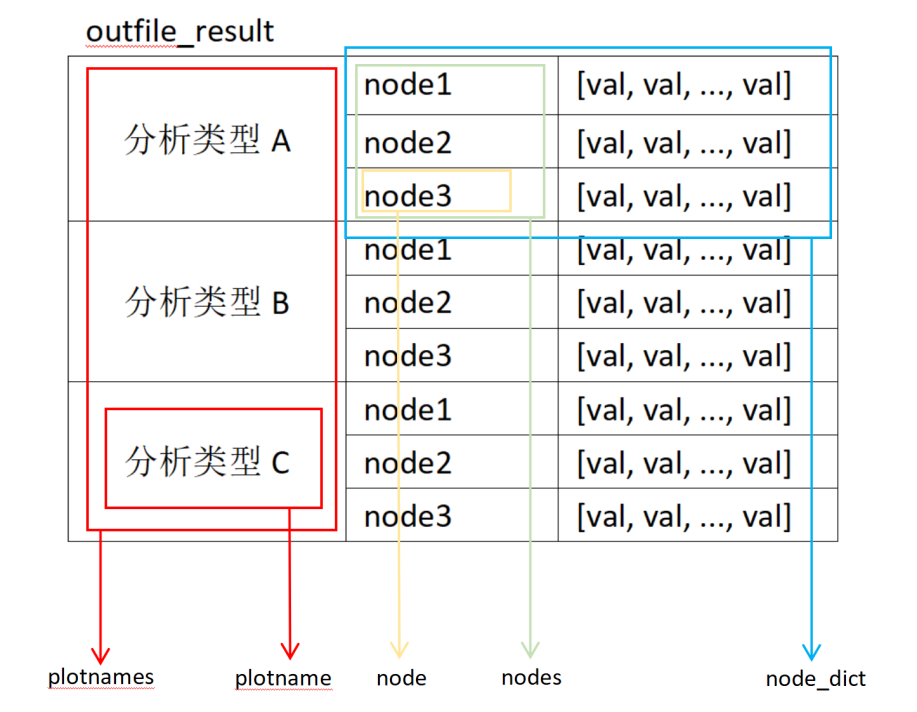
    else:

        print("node\_dict type error.")

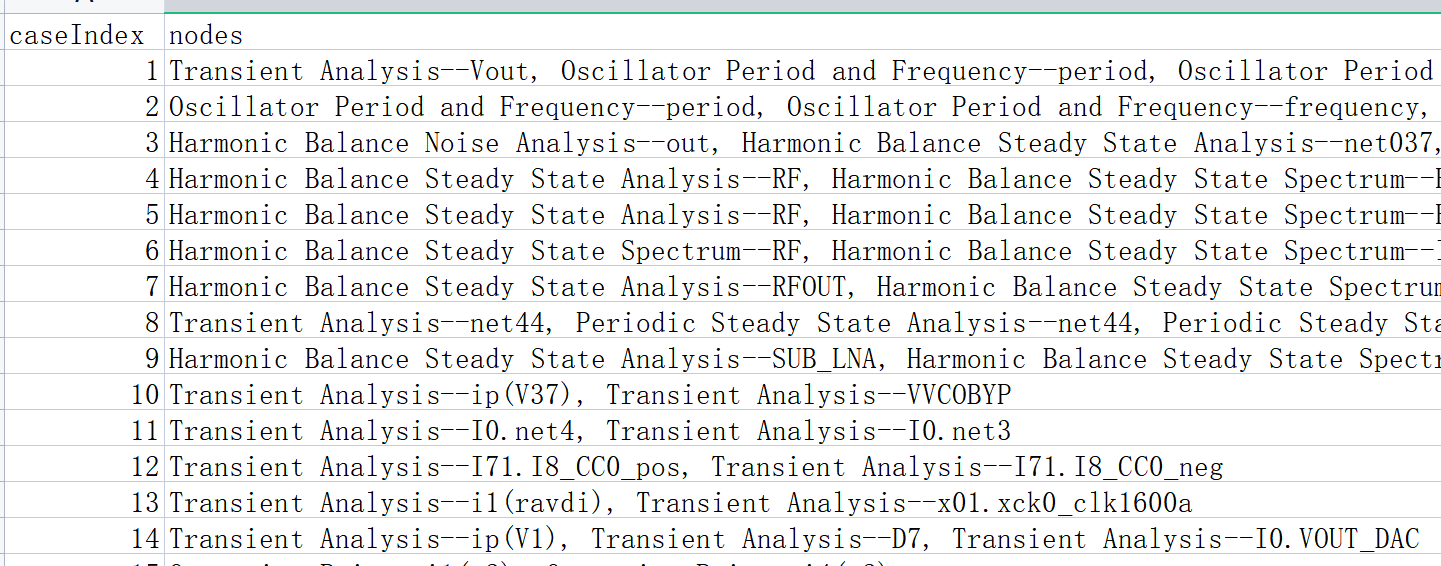
    for node in nodes:

        plotname\_nodes.append('--'.join((plotname, node)))

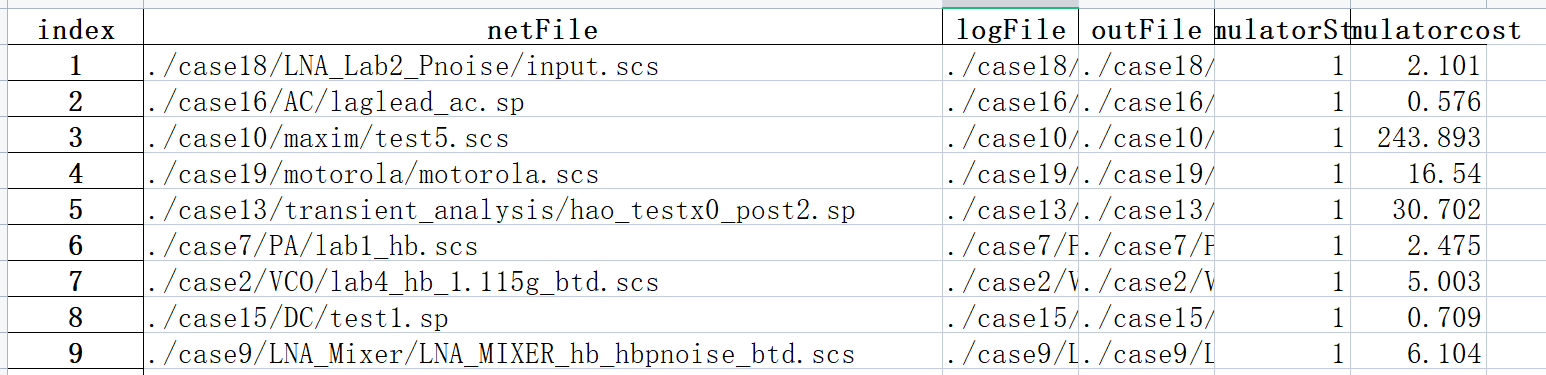
其中涉及到的参数名类似且较多，为了便于直观理解，参数名、和数据结构见下图：



定义plotname\_nodes变量获得plotname和node的mapping，是为了从外部文件'cases\_nodes.xlsx' 中寻找对应的case需要比对的plotnames和nodes。每个case需要对比的数据不同，均记录在该excel文件中。cases\_nodes.xlsx内容与结构可见下图：



然而，遍历文件时赋予的index与caseIndex并不一定一致，下图是二维矩阵data\_df\_simulator导出的excel截图，从图中可知，index和caseIndex确实不一致，所以需要调用getCaseIndex(index) 方法传入index，找到对应的caseindex并返回。



获得caseIndex后，便可去cases\_nodes.xlsx中获取该case需要对比的plotnames和nodes，并将他们赋给变量check\_nodes。

caseindex = self.getCaseIndex(index)

check\_nodes = self.check\_nodes\_dict[caseindex]

getCaseIndex(index) 函数具体如下：

def getCaseIndex(self, index):

    netfile = self.data\_df\_simulator.loc[index].netFile

    print(netfile)

    caseindex = netfile.split('case')[1].split('/')[0]

    print(caseindex)

    return int(caseindex)

对于check\_nodes中的每一个plotname（分析类型）和node（节点） 进行数据对比。

for plotname\_node in check\_nodes:

    plotn = plotname\_node.split('--')[0]  #获取当前对比的plotname

    noden = plotname\_node.split('--')[1]  #获取当前对比的nodename

    out\_plot = original\_results\_dict[plotn] #outfile中该plotname对应的nodes\_dict

    ref\_plot = new\_results\_dict[plotn] #ref\_file中该plotname对应的nodes\_dict

    if len(out\_plot) != len(ref\_plot):

        return False, 'length of out\_plot and ref\_plot not match,please check the outfile'

    for i in range(len(out\_plot)):

        # 同名plotname情况下，第i个plotname中的所有nodes

        copy\_out\_plot = out\_plot[i]

        # first one is x axis

        first\_out = next(iter(copy\_out\_plot.values()))

        #获取outfile中plotname第一个节点的对应值，实质为后续画图中的x轴

        outx = first\_out

        copy\_ref\_plot = ref\_plot[i]

        first\_ref = next(iter(copy\_ref\_plot.values()))

        #获取ref\_file中plotname第一个节点的对应值，同时为后续画图中的x轴

        refx = first\_ref

        tag = 1

        for nodename in out\_plot[i].keys(): #遍历该plotname中所有的node

            if nodename.startswith(noden): #如果当前节点即为需要对比的nodename

                outdata = out\_plot[i][nodename] #获取outfile中该node对应的值，同时为后续画图中的y轴

                refdata = ref\_plot[i][nodename] #获取ref\_file中该node对应的值

通常来说，outfile和ref\_file同一个节点的数据长度是相同的，如果长度不相同，进行数据对齐：

# 对齐数据

if len(refdata) != len(outdata):

    x\_value, outdata, refdata = AlignDataLen(outx, refx, outdata, refdata)

    if x\_value is None:

        return False, "AlignData Failed"

assert len(refdata) == len(outdata)

我们采用的是三次样条插值方法进行的数据对齐，以ref\_file的x数据为基准，模拟出outfile以此为x能够得到的一系列y值的数据。AlignDataLen(outx, refx, outdata, refdata)函数具体如下：

def AlignDataLen(outx, refx, outdata, refdata):

    """

    三次样条插值

    """

    outx, outdata = check\_duplicate(outx, outdata)

    refx, refdata = check\_duplicate(refx, refdata)

    kind = "cubic"  # 插值方式

    from scipy import interpolate

    if len(outdata) > len(refdata):

        f = interpolate.interp1d(outx, outdata, kind=kind)

        outdata = f(refx)

        return refx, outdata, refdata

    else:

        f = interpolate.interp1d(refx, refdata, kind=kind)

        refdata = f(outx)

        return outx, outdata, refdata

关于三次样条插值的概念：

插值（interpolation）是在已知部分数据节点的情况下，求解经过这些已知点的曲线，然后根据得到的曲线进行未知位置点函数值预测的方法（未知点在上述已知点自变量范围内）。

样条插值是一种分段多项式插值法。数学上，曲线光滑需要在曲线上处处一阶导连续，因此，在节点处需要满足一阶导数相等。另外，为了使得曲线的曲率最小，要求曲线二阶导连续，在节点处需要二阶导相等。

三次及以上多项式可以满足节点处光滑和曲率最小要求，但是次数高的曲线容易震荡，因此，就选用三次多项式即可。

数据对齐后，outdata和refdata的x数据必定相同，我们对插值后的outdata\_cubic和refdata进行数据对比即可。默认使用MAPE指标评估数据差异，如果命令参数指定了使用RMSE指标进行数据差异对比，则使用RMSE。

            if opt.metric == "RMSE":

                metrix\_value = get\_rmse(outdata, refdata)

            else:

                metrix\_value = get\_mape(outdata, refdata)

            comp\_value = 3e-2

            unit = nodename.split("----")[-1]

            if unit=="A":

                metrix\_value2 = max\_diff(outdata, refdata)

                comp\_value2 = 1e-9

                compareflag1 = True if metrix\_value <= comp\_value else False

                compareflag2 = True if metrix\_value2 <= comp\_value2 else False

                compareflag = compareflag1 | compareflag2

            elif unit=="V":

                metrix\_value2 = max\_diff(outdata, refdata)

                comp\_value2 = 1e-4

                compareflag1 = True if metrix\_value <= comp\_value else False

                compareflag2 = True if metrix\_value2 <= comp\_value2 else False

                compareflag = compareflag1 | compareflag2

            else:

                compareflag = True if metrix\_value <= comp\_value else False

            if plotname\_node in comp\_result.keys():

                if compareflag == False:

                    comp\_result[plotname\_node+"\_"+str(tag)] = str((metrix\_value, comp\_value, compareflag))

            else:

                comp\_result[plotname\_node] = str((metrix\_value, comp\_value, compareflag))

其中，get\_rmse(outdata, refdata) 具体如下：

def get\_mse(records\_real, records\_predict):

    """

    均方误差 估计值与真值 偏差

    """

    if len(records\_real) == len(records\_predict):

        return sum([(x - y) \*\* 2 for x, y in zip(records\_real, records\_predict)]) / len(records\_real)

    else:

        return None

def get\_rmse(records\_real, records\_predict):

    """

    均方根误差：是均方误差的算术平方根

    """

    mse = get\_mse(records\_real, records\_predict)

    if mse or mse == 0:

        return math.sqrt(mse)

    else:

        return None

else:

return None

如果outdata的平均值为0，我们将comp\_value赋值为1e-3，否则我们将outdata平均值的绝对值赋值给comp\_value。如果返回的rmse <= comp\_value或者rmse < 1e-5，我们判定数据差异在合理范围内，将compareflag标记为True，否则记为False。我们将rmse, comp\_value, compareflag拼接为字符串，并赋值给comp\_result[plotname\_node]，保存该plotname（分析类型）和该节点的数据比对结果。

除了RMSE指标，调用MAPE指标的函数如下：

def get\_mape(records\_real, records\_predict):

    """

    平均绝对百分比误差：mean(abs((YReal - YPred)./YReal))

    """

    # print(records\_real)

    # print(records\_predict)

    error = (np.array(records\_real) - np.array(records\_predict)).tolist()

    pe = []

    if np.array(records\_predict).any():

        for i, predict in enumerate(records\_predict):

            if predict != 0:

                pe.append(error[i] / predict)

        # pe = [error / real for real in np.array(records\_real) if real != 0]

        m = math.ceil(0.9\*len(pe))

        pe =sorted(np.abs(pe), key=lambda x:x)[:m]

        return np.mean(np.abs(pe))

    else:

        return np.mean(np.array(records\_real))

如果outdata的平均值 > 1e-6，我们将comp\_value赋值为1e-3，否则我们给comp\_value赋值为1。如果返回的mape <= comp\_value，我们判断数据差异在合理范围内，将compareflag标记为True，否则记为False。我们将mape, comp\_value, compareflag拼接为字符串，并赋值给comp\_result[plotname\_node]，保存该plotname（分析类型）和该节点的数据比对结果。

无论是使用了RMSE指标还是MAPE指标，我们都将对比结果compareflag添加到对比结果列表comparelist中。只要comparelist中存在False标记，我们就将返回值compare标为False；若comparelist不含False，则将compare标为True返回。

comparelist.append(compareflag)

compare = False if False in comparelist else True

函数calc\_error() 的返回值为compare (True/False) 和comp\_result字符串。

return compare, str(comp\_result)

1. save\_plot(index, out\_results\_dict, ref\_results\_dict, compare)：

数据的对比结果是抽象的，将仿真对比结果画图导出，可以方便肉眼直观的观察对比结果。保存图像命令 --savefig 默认为True。

我们需要先从传入的out文件解析结果out\_results\_dict中提取出所有plotname和node一一对应的mapping，并赋值给plotname\_nodes这个list。

def save\_plot(self, index, out\_results\_dict, ref\_results\_dict, compare):

    # 根据对比结果定义标题颜色

    titlecolor = 'green' if compare else 'red'

    plotname\_nodes = []

    plotnames = out\_results\_dict.keys()

    for plotname in plotnames:

        node\_dict = out\_results\_dict[plotname]

        code\_node\_dict = node\_dict

        nodes = node\_dict[0].keys()

        for node in nodes:

            plotname\_nodes.append('--'.join((plotname, node)))

并且和之前相同，需要根据index找到对应的caseIndex，因为我们并不是所有的plotname-node都画图，我们只根据对应case需要观察的节点进行画图。

# 读取case number，作为index找到对应case需要查看的节点

caseindex = self.getCaseIndex(index)

check\_nodes = self.check\_nodes\_dict[caseindex]

使用for循环，对于每一个check\_nodes中存在的plotname-node配对进行数据处理和画图。

首先，我们需要获得当前遍历的plotname和nodename分别是什么，并且从out\_results\_dict和ref\_results\_dict获得对应plotname（分析类型）的所有节点。获得所有节点的目的主要是为了获得第一个节点，第一个节点为画图的横坐标x。

for plotname\_node in check\_nodes:

    plotn = plotname\_node.split('--')[0]  #当前遍历的plotname

    noden = plotname\_node.split('--')[1] #当前遍历的nodename

    out\_plot = out\_results\_dict[plotn] #out文件中该plotname下的nodes\_dict

    ref\_plot = ref\_results\_dict[plotn]  #bench文件中该plotname下的nodes\_dict

    if len(out\_plot) != len(ref\_plot):

        continue

#得到第一个nodename、单位、值，作为后续节点画图的横坐标

for i in range(len(out\_plot)):

    # print("current i: {}".format(i))

    x\_info\_key = list(out\_plot[i].keys())[0]

    out\_xname = x\_info\_key.split('----')[0]

    out\_xunit = x\_info\_key.split('----')[1]

    outx = next(iter(out\_plot[i].values()))

    refx = next(iter(ref\_plot[i].values()))

获得x轴数据后，我们取需要观察的节点数据，即画图的y轴数据。

for nodename in out\_plot[i].keys():

    if nodename.startswith(noden): #找到我们需要的节点node

        # 得到y轴的单位

        yunit = nodename.split("----")[1]

        # 得到y轴的数据

        outdata = out\_plot[i][nodename]

        refdata = ref\_plot[i][nodename]

同样，如果数据长度不一致，我们则调用AlignData() 函数进行数据对齐处理：

# 长度不一致需做数据对齐

if len(refdata) != len(outdata):

    xnew, outdata, refdata = AlignDataLen(outx, refx, outdata, refdata)

    if xnew is None:

        continue

else:

    xnew = outx #长度一致时，直接取out文件该plotname的第一个节点数据

assert len(refdata) == len(outdata)

如果y轴数据长度为1，代表只有一个点，无需进行画图：

if len(outdata) == 1:

    continue

为了便于醒目，对于数据对比差异较大我们认为有异常的图，标题设置为红色，差异在误差范围内的，标题设置为绿色。

titlecolor = 'green' if compare else 'red'

通常来说，如果该数据是频谱，我们画柱状图，否则选取折线图进行绘制。一些画图的细节处理在代码中以注释的形式进行描述。

if "Spectrum" in plotn:

    # 设置柱状图y轴的起始值

    bottomvalue = np.min(np.array(outdata)) - 10

    # GHz transform

    # 对于过大的x轴数据，进行单位转换

    if np.max(xnew) > 1e9:

        xnew = (np.array(xnew) \* 1e-9).tolist()

        out\_xunit = "G" + out\_xunit

    total\_width, n = 0.6, 2

    width = total\_width / n

    # fig = plt.figure()

    outdata = (np.array(outdata) + np.abs(bottomvalue)).tolist()

    refdata = (np.array(refdata) + np.abs(bottomvalue)).tolist()

# 用蓝色绘制此次仿真结果outdata

plt.bar(xnew, outdata, width=width, color='b', label='outdata', bottom=bottomvalue)

# for a, b in zip(xnew, outdata):  # 柱子上的数字显示

# 柱状图中第二个条形需要进行宽度的平移

for j in range(len(xnew)):

    xnew[j] = xnew[j] + width

# 用红色绘制bench文件中的refdata

plt.bar(xnew, refdata, width=width, color='r', label='refdata', bottom=bottomvalue)

plt.xticks(rotation=90)  # 横坐标刻度上的数字逆时针旋转90度

plt.xlabel(out\_xname + "(" + out\_xunit + ")")  # 横坐标标签

plt.ylabel(noden + "(" + yunit + ")")  # 纵坐标标签

if i == 0:

    mark = "first"

else:

    mark = "last"

plt.title("{} {}".format(plotname\_node, mark), backgroundcolor=titlecolor)  # bar图标题

# plt.grid(axis='x')

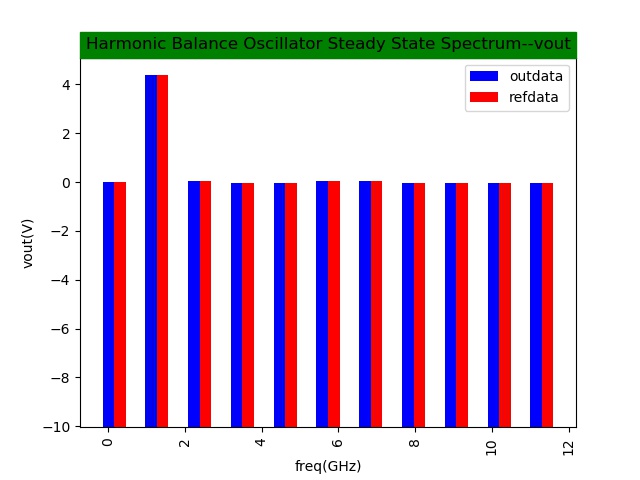
plt.legend(loc='best') # 在画布上选取最佳位置放置图例标签

savepath = f"{self.output\_folder}/case{str(caseindex)}\_{plotname\_node} {mark}.jpg"

plt.savefig(savepath) # 将绘制好的图片保存到指定路径

plt.close()

绘制完成的柱状图样例见下图：



绘制折线图的代码块如下，同样一些画图的细节处理在代码中以注释的形式进行描述：

if "noise" in plotname\_node or "Noise" in plotname\_node:

    # log transform  对于noise节点的数据，做log转换

    # if "noise" in noden or "Hz" in out\_xunit:

    xnew = np.log10(xnew)

    out\_xname = out\_xname + "\_log"

else:

    # 对于过大的x轴数据，进行GHz转换

    if np.max(xnew) > 1e6:

        xnew = (np.array(xnew) \* 1e-9).tolist()

        out\_xunit = "G" + out\_xunit

    # nano transform 对于nano 数据进行单位转换

    elif np.max(xnew) < 1e-6:

        xnew = (np.array(xnew) \* 1e9).tolist()

        out\_xunit = "n" + out\_xunit

    else:

        pass

plt.plot(xnew, outdata, 'b', label='outdata')  # (横坐标，纵坐标) # 用蓝色线绘制此次仿真结果outdata

plt.plot(xnew, refdata, 'r', label='refdata')  # (横坐标，纵坐标) # 用红色线绘制bench文件中的refdata

plt.xlabel(out\_xname + "(" + out\_xunit + ")")  # 横坐标标签

plt.ylabel(noden + "(" + yunit + ")")  # 纵坐标标签

plt.grid() # 绘制背景网格线

if i == 0:

    mark = "first"

else:

    mark = "last"

plt.title("{} {}".format(plotname\_node, mark), backgroundcolor=titlecolor)  # 折线图标题

plt.legend(loc='best') # 在画布上选取最佳位置放置图例标签

savepath = f"{self.output\_folder}/case" + str(caseindex) + "\_" + plotname\_node + " " + mark + '.jpg'

plt.savefig(savepath) # 将绘制好的图片保存到指定路径

# plt.show()

plt.close()

if i == 0:

    mark = "first"

else:

    mark = "last"

plt.title("{} {}".format(plotname\_node, mark), backgroundcolor=titlecolor)  # 折线图标题

plt.legend(loc='best') # 在画布上选取最佳位置放置图例标签

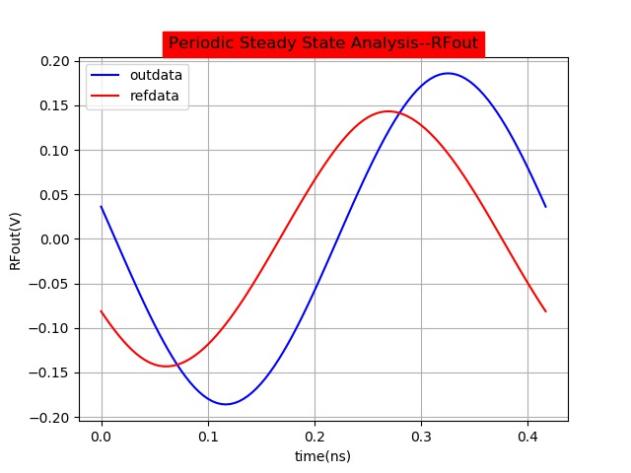
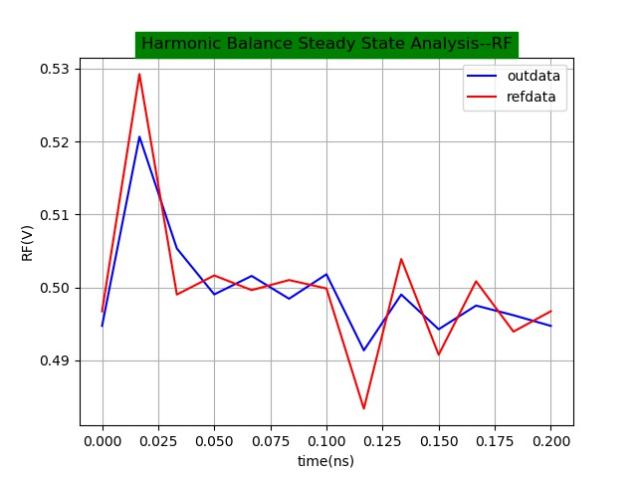
savepath = f"{self.output\_folder}/case" + str(caseindex) + "\_" + plotname\_node + " " + mark + '.jpg'

plt.savefig(savepath) # 将绘制好的图片保存到指定路径

# plt.show()

plt.close()

绘制完成的折线图样例见下图：



1. 对比结果写入Excel

将二维矩阵data\_df\_diff记录的对比结果数据导出写入Excel，方便直接查看数据比对结果。date\_str为先前读取的当前系统时间，并将它拼接在了文件名称中以便在有多份结果对比文件时能根据生成的时间加以区分。

writer2 = pd.ExcelWriter(f'{atc.output\_folder}/data\_df\_diff\_{date\_str}.xlsx')

atc.data\_df\_diff.to\_excel(writer2)

writer2.save()

1. 持续优化
2. 多线程或多进程进行结果比较
3. 多线程或多进程进行仿真
4. -h 信息尽可能详细，各种参数的含义，以及一条完整的使用脚本的语句
5. V5的功能：
   1. GUI界面
   2. 测试集可选（一个或多个）（测试集包括spectre语法、Hspice语法、器件数超过5000、器件数超过10000、bulk模型、cmg模型、hbt模型、eehemt模型等等）
   3. 可选btdsim版本