WASAoutputSimilarityTest_v2

Introduction

We study 16 cases, detailed earlier and described in the table ***test_cases_description***, to test the forecasting code WASA-SED. These cases are reminded by the table 1.

Folder → / **↓ Options**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**
reservoir	.f.	.f.	**.t.**	**.t.**	**.t.**	**.t.**	**.t.**	**.t.**	**.t.**	**.t.**	**.t.**	**.t.**	**.t.**	**.t.**	**.t.**	**.t.**
doacudes	.t.	.t.	.t.	.t.	**.f.**	**.f.**	.t.	.t.	.t.	.t.	.t.	.t.	.t.	.t.	.t.	.t.
dotrans	.f.	.f.	.f.	.f.	.f.	.f.	**.t.**	.f.	.f.	.f.	.f.	.f.	.f.	.f.	**.t.**	.f.
dohour	.f.	**.t.**	.f.	**.t.**	.f.	**.t.**	.f.	.f.	.f.	**.t.**	.f.	.f.	.f.	.f.	.f.	.f.
dt	24	**1**	24	**1**	24	**1**	24	24	24	**1**	24	24	24	24	24	24
dointc	0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0
sediment	.f.	.f.	.f.	.f.	.f.	.f.	.f.	.f.	**.t.**	**.t.**	**.t.**	**.t.**	**.t.**	**.t.**	**.t.**	**.t.**
river routing	1	1	1	1	1	1	1	1	1	1	**2**	**3**	1	1	1	1
option : load state	.t..f.	.t..f.	.t..f.	.t..f.	.t..f.	.t..f.	.t..f.	.t..f.	.t..f.	.t..f.	.t..f.	.t..f.	**.t.**	**.f.**	.t..f.	.t..f.
option : save state	.t..f.	.t..f.	.t..f.	.t..f.	.t..f.	.t..f.	.t..f.	.t..f.	.t..f.	.t..f.	.t..f.	.t..f.	**.t.**	**.f.**	.t..f.	.t..f.
snow	.f.	.f.	.f.	.f.	.f.	.f.	.f.	.f.	.f.	.f.	.f.	.f.	.f.	.f.	**.t.**	.f.
irrigation	.f.	.f.	.f.	.f.	.f.	.f.	.f.	.f.	.f.	.f.	.f.	.f.	.f.	.f.	**.t.**	**.t.**

Table 1: *Description of the 16 cases tested. Changes from the basic case (case 1) are emphasized by the blue cells.*

Now, the objective is to check the consistence of WASA-SED. How do we proceed? For each case we get two output folders with two different code versions: the reference and the « new-version ». Then, we compare the files by pairs with the same name (one from the reference and the other from the new version) and check if there is any difference between them. We impose an error tolerance, under which we consider the files identical. In order to do this, we write a Python code named **WASAoutputSimilarityTest_v2** which is the upgraded code of **WASAoutputSimilarityTest**. Indeed, this new version is more complete and have better functionalities.

I. Files organization

In the code, we consider the organization for the **wasa_tests** folder presented by figure 1. (it has changes from the previous version of the code).

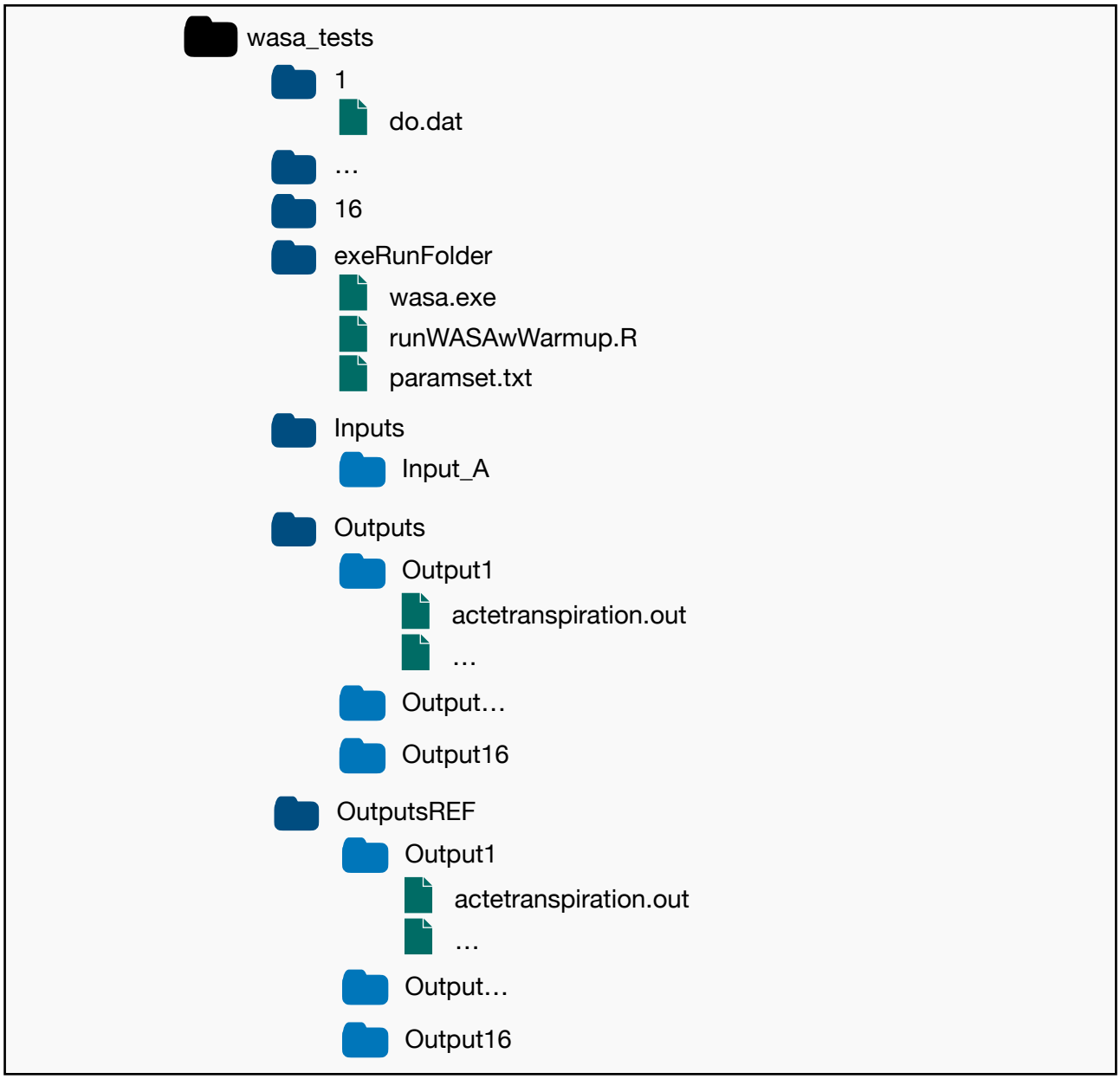

Figure 1: *Files and folders organization to use the code*

The folders organization is important into the code. If this one is not followed the actual code will not work. But, it is easy to change this part into the code. So, if in the future we decide to change the path of the files it won't be difficult to modify the code.

In this situation the two folders we compare are **Outputs** (new version) and **OutputsREF** (the reference). They are contained into the repository **wasa_tests**. In these two folders we have 16 output folders corresponding to the 16 cases we are studying and in which there are the files we want to compare.

Note: Currently in the repository the folders **Outputs** and **OutputsREF** do not contain the final data. Indeed, we needed to create some ‘fake’ folders to try the code because we did not have define the reference and what will be the new version yet. So we just put on them, two random versions of the code. Also, the code is in the folder **WASA_pyProject** in the repository.

II. Code description

1. General

The aim of this code is to return if, for one case, there is errors or not between the two code versions. Indeed, after comparing the output files of the two code versions for each case, the code returns a text file containing the results of the comparison. An example of what this results file can be is presented by the figure 2.

This file presents the results of the similarity checking of the output files between the reference (benchmark) and the new version of WASA-SED by using the Python code WASAoutputSimilarityTest_v2.
NOTE: The file names which appear in the section -General issues- are not compared. We only compare the files which can be compared.

* Case 1 :
  - General issues :

The file(s) interflow_storage.stat_start, is (are) present in the new version but missing in the reference.

- Data comparison result :

ERROR =>[['gw_discharge.out', '21.0%'], ['total_overlandflow.out', '48.0%'], ['qhorton.out', '35.0%']]

* Case 2 :
  - General issues :

There are extra data in the file subsurface_runoff.out for the new version.
There are extra data in the file potetranspiration.out for the new version.
There are extra data in the file total_overlandflow.out for the new version.

- Data comparison result :

OK

Figure 2: *Result text file example.*

This file is organized with a little introduction to explain what we will read and then a section per case of study. Each section has two parts:

- **General issues:** it contains the general issues of the case. That means if there are empty files, missing files in a version, extra data in a file for one version, this will be highlighted in this part.

  Note: the files mentioned in this section won't have their data compared.

- **Data comparison result:** it returns for each case *OK* if all the output files of the case are identical. If they are not, it returns *ERROR* and specifies which files are different by specifying the file name and the error associated which is obviously over the threshold.

Finally, regarding the general organization of the code, we decided to define functions to have a clear presentation and to split the code in two parts: a first part with the functions definition and an other part with the calculations.

Before continuing, we precise the python modules we import at the beginning of the code and that we will need :

```
import os
import pandas as pd
import math
```

2. Functions description

In this part we will list the functions we use and then describe them.

The functions are :

- RUNexe
- CreatePath
- FileNamesList
- CreateDf1
- CreateDf2
- TFcomparison
- ErrorCalculation
- ComparisonThreshold
- GeneralComparison
- IssueFilePresence
- IssueEmptyFile
- ComparisonCheck
- ReadDf
- MainLoop

➡ RUNexe :

This function runs the executable *wasa.exe* for one case of study (which can be 1 to 16). We go into the repository (which has been dowloaded in the folder Documents on the computer I used as we can see with the path 'G:\Documents\wasa_tests') and fill in the command which contains where we find the executable, the *do.dat* file and where we want to write the resulting output folder. Then, thanks to the python function **os.system()**, it executes the command.

- Entry : **case_number**, integer, this is the number of the case we test (from 1 until 16).
- Return : no return.

Note1: The command is given by the line « **command** = 'exeRunFolder\wasa.exe %d\do.dat > Outputs/Output%d/Protocol.txt'%**(case_number, case_number)** ». The two %d into the string part refer twice to a number which is **case_number**. It permits to adapt the command to the case number and add a variable to the string.

Note2: The output folders of the cases (so the folders Output1, Output2, …) have to exist before using this function. Otherwise we will get an error into the Python console.

```
def RUNexe(case_nb):
    os.chdir('G:\Documents\wasa_tests')
    command = 'exeRunFolder\wasa.exe %d\do.dat > Outputs/Output%d/Protocol.txt'%(case_nb,case_nb)
    os.system(command)
```

➡ CreatePath :

This function creates two paths to have access to the two versions we compare for one case of study.

- Entry : **case_number**, integer, this is the number of the case we test (from 1 until 16).
- Return : **path_ref**, string, corresponding to the path to go into the output folder of a specific case for the reference ; **path_new**, string, corresponding to the path to go into the output folder of a specific case for the new version.

```
def CreatePath(case_number):
    path_ref = 'G:/Documents/wasa_tests/OutputsREFfake/Output%d'% (case_number)
    path_new = 'G:/Documents/wasa_tests/OutputsFake/Output%d'% (case_number)
    return path_ref, path_new
```

➡ FileNamesList :

From a path to a case output folder, the function first creates a list of all the file names contained into the folder. Then, it sorts out the files to keep only the ones we are interested in. Indeed, it removes the files *parameter.out* and *Protocol.txt* which are not useful to compare and .DS_Store or ._.DS_Store which are files which can appear when using Python with MacOS.

- Entry : **path**, string, corresponding to the path of a case output folder (e.g. 'G:/Documents/wasa_tests/Outputs/Output1' if we want to go into the first case of the new version).
- Return : **file_list**, list of strings, contains the file names we test for the case considered.

Note: In the current function we also remove some additional files because they are organized in a different way than the others and I did not find a way to read them. These files are: river_storage.stat, river_storage.stat_start, susp_sediment_storage.stat and susp_sediment_storage.stat_start. To improve the code and the comparison we have to solve this issue by finding a way to read these files.

```
def FileNamesList(path):
    os.chdir(path)
    file_list = os.listdir(path)
    for file in file_list :
        if file == 'Protocol.txt':
            file_list.remove(file)
        elif file == 'parameter.out':
            file_list.remove(file)
        elif file == '._.DS_Store':
            file_list.remove(file)
    for f in file_list:
        if f=='.DS_Store':
            file_list.remove(f)
    for f2 in file_list:
        if f2=='river_storage.stat':
            file_list.remove(f2)
    for f3 in file_list:
        if f3=='river_storage.stat_start':
            file_list.remove(f3)
    for f4 in file_list:
        if f4=='susp_sediment_storage.stat':
            file_list.remove(f4)
    for f5 in file_list:
        if f5=='susp_sediment_storage.stat_start':
            file_list.remove(f5)
    return file_list
```

➡ CreateDf1 and CreateDf2 :

From a path to a case output folder and a file name, the functions read the corresponding file and create a data-frame. The difference between the two functions is the way that the files are read. We will prefer **CreateDf1** to get float data after the comparison between two data-frames but sometimes the comparison will not work with this reading mode so we will have to read the files with the function **CreateDf2**.

- Entry : **path**, string, the path of a case output folder ; **file_name**, string, the name of the file we want to convert into a data-frame.
- Return : **df**, object, the resulting data-frame.

```
def CreateDf1(file_name,path):
    os.chdir(path)
    df = pd.read_csv(file_name, 'b', engine='python', delimiter='\t', header=1)
    return df

def CreateDf2(file_name,path):
    os.chdir(path)
    df = pd.read_csv(file_name, 'b', engine='python', delimiter='\t')
    return df
```

➡ TFcomparison :

From two data-frames it checks if the data-frames are exactly identical or not.

- Entry : **df1** and **df2**, objects, two data-frames we want to compare.
- Return : **comp**, boolean. **True** means that the two data-frames are identical and **False** means that they are different.

```
def TFcomparison(df1, df2): #True/False comparison
    comp=df1.equals(df2)
    return comp
```

➡ ComparisonThreshold :

From two values, a calculated error and a threshold (under which we consider the error as acceptable), it compares them and return **True** if the error is under the threshold and **False** if it is above.

- Entry : **error_exact** and **error_acceptable**, floats, the error we have calculated and the threshold we have decided to apply.
- Return : a boolean, **True** if the error is under the threshold and **False** if it is above.

```
def ComparisonThreshold(error_exact, error_acceptable):
    if error_exact <= error_acceptable :
        return True
    else :
        return False
```

➡ ErrorCalculation :

From two values, the reference which is the value from the reference file and the value from the new version file, it calculates the error between them. We write the error with 2 decimal places.

- Entry : **ref** and **value**, floats, two values we want to compare.
- Return : **error**, float with 2 decimal places.

Note1: When the reference is equal to zero we can not divide by zero so we just keep the simple difference between the two data.

Note2: In this note we explain the two first lines of this function. First, we have to precise that, when we compare two data-frames (with the python function dataframe.compare()), we will get a new data-frame with the data which are different from a version to another. An example is presented by figure 3.

```
df1 =
      Year  Day  Timestep    15    69
0    2000    1         1  0.108  0.011
1    2000    1         2  0.060  0.011
2    2000    1         3  0.001  0.011
3    2000    1         4  0.101  0.013
4    2000    1         5  0.022  0.013

df2 =
      Year  Day  Timestep    15    69
0    2000    1         1  0.108  0.013
1    2000    1         2  0.060  0.013
2    2000    1         3  0.009  0.013
3    2000    1         4  0.108  0.013
4    2000    1         5  0.022  0.013

>>> dfComp = df2.compare(df1)

dfComp =
       15          69
   self  other   self  other
0   NaN    NaN  0.013  0.011
1   NaN    NaN  0.013  0.011
2 0.009  0.001  0.013  0.011
```

Figure 3: *Example to explain the function ErrorCalculation.*

From this example, the columns 'self' refer to the data-frame df2 and 'other' to the data-frame df1. We can note that sometimes for both 'self' and 'other' we get NaN values. That means the data are the same but because for the other column (here for the sub-basin 69) the data are not the same, something has to be written to complete the line. And this something is NaN. So the two first lines of the function permit to take into account this scenario by returning an error equal to zero when it occurs.

```
def ErrorCalculation(ref, value):
    if math.isnan(ref)==True and math.isnan(value)==True:
        return 0
    elif ref==0:
        e = abs(ref-value)
    else:
        e = abs(ref-value)/ref
    error = float("{:.2f}".format(e))*100
    return error
```

➡ GeneralComparison :

From two data-frames, the associated file name and the acceptable error, it determines if the two files are identical or not. We can explain this function with different steps:

1. We use the function **TFcomparison** to check if the two data-frames are exactly identical or not. If they are, we stop here and return **True**. If they are not we go to the step 2.
2. We compare the data-frames with the python function **dataframe.compare()** and we get a resulting data-frame with the values which differ from a data-frame to another (see example on figure 3). We convert this new data-frame into a list and we check if the data in the list are float or string (see figure 4 to see the two types of list we can get).
3. If they are floats, we first check if all the values are comparable. That means if there are *** or NaN values (except for the case we have presented earlier with the function **ErrorCalculation**). If it occurs, we write, into the result text file, an error message announcing that there are incomparable values in the file considered and we return **True**. Returning **True** permits, in the future, to ignore this case because we already wrote on the text file the result for the file considered. If all the values are comparable we go to the step 5.
4. If they are string, we need to convert them into float to be able to calculate the error between the data. In this case, the list contains several lists where each as two elements. Each element is a succession of number such as ‘ 1000 2 0.00’. So, we go through each element to convert the numbers we can find to float. The difference of the lists we have between step 3 and step 4 is shown on figure 5. When all the numbers are found and translated into float, we go to the step 5. We can note that the step 5 appears two times on the code: at the end of each step (3 and 4).
5. For each line and each column of the list, we calculate the error between the data with **ErrorCalculation**. Then, we take the maximum of all these errors. This means that the error of the entire file is the maximum of all the errors there are on the file. Then, we compare this error with the threshold thanks to the function **ComparisonThreshold**. If the error is under the threshold the code returns **True**. If it is not, it returns a list with the name of the file and the error associated.

```
#Option 1: reading the files with CreateDf1

dfComp =
      15          69
   self  other   self  other
0   NaN    NaN  0.013  0.011
1  0.009  0.001  0.013  0.011

 L=
[ [NaN, NaN, 0.013, 0.011], [NaN, NaN, 0.013, 0.011], [0.009, 0.001, 0.013, 0.011]


#Option 2: reading the files with CreateDf2

dfComp =
      self                                  other
15  2000    1    1        0.000             15  2000    1    1        0.001
15  2000    1    2        0.000             15  2000    1    2        0.001

L=
 [['   15  2000    1    1        0.000', '   15  2000    1    1        0.001'], ['   15  2000
1    2        0.000', '   15  2000    1    2        0.001']]
```

Figure 4: *Example to different organization of the list we get after the comparison.*

- Entry : **file_name**, string, the name of the file ; **dfRef** and **dfNew**, objects, the two data-frames we compare ; **error_acceptable**, float, the threshold ; **textFile**, the opened text file for the results.
- Return : **TF** or **TFerror**, a boolean, which can only be **True** or it returns a list such as **[file_name, written_error]** which contains two strings : the name of the file and the associated error converted from a float to a string and written in percentage.

Example: A returned list can be ['daily_actetranspiration.out', '26%'].

```
def GeneralComparison(file_name, dfRef, dfNew, error_acceptable, textFile):
    TF = TFcomparison(dfRef, dfNew)
    if TF == True:
        return TF
    else:
        CompDf=dfRef.compare(dfNew)
        CompList=CompDf.values.tolist()
        n = len(CompList)
        S = []
        if isinstance(CompList[0][0], float)==True:
            for j in range(n):
                m = len(CompList[j])
                for i in range(0,m,2):
                    if i<m:
                        if isinstance(CompList[j][i], str)==True or isinstance(CompList[j][i+1], str)==True or (math.isnan(Compl
                            textFile.write('Incomparable value in the file '+file_name+'.\n')
                            return True
                        else:
                            cal = ErrorCalculation(CompList[j][i], CompList[j][i+1])
                            S.append(cal)
        else: #that means we have string data
            CompList_Ref=[]
            CompList_New=[]
            for j in range(n):
                CL_Ref_line=[]
                CL_New_line=[]
                if CompList[j][0] != CompList[j][1]:
                    CL_Ref_value = ''
                    CL_New_value = ''
                    for k in range(len(CompList[j][0])-1):
                        if CompList[j][0][k]!=' ':
                            CL_Ref_value = CL_Ref_value + CompList[j][0][k]
                            if k==len(CompList[j][0]) or CompList[j][0][k+1]==' ':
                                CL_Ref_line.append(float(CL_Ref_value))
                                CL_Ref_value = ''
                    for k1 in range(len(CompList[j][1])-1):
                        if CompList[j][1][k1]!=' ':
                            CL_New_value = CL_New_value + CompList[j][1][k1]
                            if k1==len(CompList[j][1]) or CompList[j][1][k1+1]==' ':
                                CL_New_line.append(float(CL_New_value))
                                CL_New_value = ''
                    CompList_Ref.append(CL_Ref_line)
                    CompList_New.append(CL_New_line)
                    m = len(CompList[j])
                    for i in range(m):
                        cal = ErrorCalculation(CompList_Ref[j][i], CompList_New[j][i])
                        S.append(cal)

            max_error = max(S)
            written_error = str(max_error)+'%'
            TFerror = ComparisonThreshold(max_error, error_acceptable)
            if TFerror == True:
                return TFerror
            else:
                return [file_name, written_error]
```

➡ IssueFilePresence :

From two lists of file names, it compares them to check if, for one case, the same output files are present on the Output folder. It returns a list with the comparison result and a list of file names. If there is an issue concerning one of the files, the name of this file will not appear on the returning list of file names. To do this comparison we create a list with what differs from a names list to the other.

- Entry : **namelistRef** and **namelistNew**, lists of strings, the list of the file names which are into the case output folder for the reference and the list of the file names which are into the case output folder for the new version.
- Return : **res**, list of strings, is empty if the file names are the same for both versions and contains error messages if this is not the case ; **namelistRef**, list of strings, the list of the file names we will compare (if there is an error message concerning one file into **res**, the name of this file will not appear into **namelistRef**).

```
def IssueFilePresence(namelistRef, namelistNew):
    diff1=list(set(namelistRef)-set(namelistNew))
    diff2=list(set(namelistNew)-set(namelistRef))
    res = []
    if diff1 != []:
        bonus_file=''
        for name in diff1:
            bonus_file=bonus_file+name+', '
            namelistRef.remove(name)
        res.append('    The file(s) '+bonus_file+' is (are) missing in the new version.')
    if diff2 != []:
        bonus_file2=''
        for name2 in diff2:
            bonus_file2=bonus_file2+name2+', '
        res.append('    The file(s) '+bonus_file2+' is (are) present in the new version but missing i
    return res, namelistRef
```

➡ IssueEmptyFile :

From two data-frames and a file name, it checks if one of the data-frames is empty or if there is extra data in one of the data-frames. It returns possible error messages. This function is based on the comparison of the data-frames size. Indeed, if the size of one of the two data-fames is zero, it means that one of the files is empty. Also, if they do not have the same size, that means one data-frame has more rows so more data than the other one.

- Entry : **dfRef** and **dfNew**, objects, the data-frames from the reference and the new version for one file ; **file_name**, string, the name of the considered file.
- Return : **res**, list of strings, is empty if the data-frames are not empty and have the same number of data or contains errors messages if this is not the case.

```
def IssueEmptyFile(dfRef, dfNew, file_name):
    R = len(dfRef)
    N = len(dfNew)
    res = []
    if R>N:
        if N==0:
            res.append('    The file '+file_name+' is empty in the new version.')
        else:
            res.append('    There are missing data in the file '+file_name+' for the new version.')
    elif R<N:
        if R==0:
            res.append('    The file '+file_name+' is empty in the reference.')
        else:
            res.append('    There are extra data in the file '+file_name+' for the new version.')
    return res
```

➡ ComparisonCheck :

From two data-frames this function checks if it is possible to use the python function **dataframe.compare()** or if using this function gives an error.

- Entry : **dfA** and **dfB**, objects, the data-frames we want to compare.
- Return : boolean, **True** if we can use the function **dataframe.compare()** with the two data-frames and **False** if we can not.

```
def ComparisonCheck(dfA, dfB):
    try:
        dfA.compare(dfB)
        return True
    except ValueError:
        return False
```

➡ ReadDf :

From a file name and two paths, one to go into an output folder of the reference and the other to go into an output folder of the new version, the function gives two data-frames for which the python function **dataframe.compare()** works. Indeed, at the beginning the data-frames are created with the function **CreateDf1** and then we check if we can use the comparison python function with **ComparisonCheck**. If there is no problem, we return these data-frames. If there is an issue, we create new data-frames with the function **CreateDf2** and we return them if it is possible to compare them.

- Entry : **file_name**, string, the name of the files we want to convert into data-frames ; **pathA** and **pathB**, strings, the paths of the case output folders (one for the reference and the other for the new version).
- Return : **dfA(2)** and **dfB(2)**, objects, the data-frames we are able to compare.

```
def ReadDf(file_name, pathA, pathB):
    dfA = CreateDf1(file_name, pathA)
    dfB = CreateDf1(file_name, pathB)
    if ComparisonCheck(dfA, dfB)==True:
        return dfA, dfB
    else:
        dfA2 = CreateDf2(file_name, pathA)
        dfB2 = CreateDf2(file_name, pathB)
        if ComparisonCheck(dfA2, dfB2)==True:
            return dfA2, dfB2
```

➡ MainLoop :

This is the main function of the code which will refer to some the other functions. The aim is to give the comparison result for one case of study. This function has different steps:

1. We create two lists: one which will keep the results (*result*), another one (*filesToDelete*) to take the name of the files we may have to delete because of some possible error messages given by the function **IssueEmptyFile**.
2. We go through the list of names and for each name, we use the function **ReadDf** to create the data-frames (one for the reference and one for the new version) that we will be able to compare.

3. Then, for each file and couple of data-frames we check if there is an empty data-frame or extra data with the function **IssueEmptyFile**. We remind that this function gives us a list. If this list is not empty, it means there is an issue regarding the file we consider. In this case, we write the error message into the result text file and we append the file name to the list *filesToDelete*. If the result of the function **IssueEmptyFile** is empty, we just go to the next step.
4. After went through the entire list of file names, we take the names of the files we want to compare. That means we delete from the initial list of file names, the file names written into *filesToDelete*. So we get a list of file names we will be able to compare.
5. We go through this new list of file names, for each file, we use **ReadDf** to create the data-frames and then use **GeneralComparison** to compare them. If it does not gives **True** that means it gives a list (cf the explanation of the function **GeneralComparison**). In this case we append what we get to the list *result*.
6. Finally, the code returns the list *result*. If the list is empty that means all the files of this case of study are considering identical. Otherwise, that means there is an error regarding at least one of the files.

- Entry : **pathRef** and **pathNew**, strings, corresponding to the paths to go into the output folder of the specified case of study for the reference and for new version ; **namesList**, list of strings, containing the names of the output file for the considered case ; **errorAcceptable**, float, the threshold we choose and use for the data comparison ; **textFile**, the opened text file for the results.
- Return : **result**, list of strings list, it can be empty or can contained one list per file for which the error is higher than the threshold.

Example: A returned list can be [[‘daily_actetranspiration.out’, ’26%’], [‘actetranspiration.out’, ’16%’]].

```
def MainLoop(pathRef, pathNew, namesList, errorAcceptable, textFile):
    result = []
    filesToDelete = []
    for name in namesList:
        df_ref,  df_new = ReadDf(name, pathRef, pathNew)
        IEF= IssueEmptyFile(df_ref, df_new, name)
        if IEF != []:
            print(IEF[0],'\n')
            textFile.writelines([IEF[0], '\n'])
            filesToDelete.append(name)
    resultingNamesList=list(set(namesList)-set(filesToDelete))
    for nameB in resultingNamesList:
        df_ref2,  df_new2 = ReadDf(nameB, pathRef, pathNew)
        GC=GeneralComparison(nameB, df_ref2, df_new2, errorAcceptable, textFile)
        if GC != True:
            result.append(GC)
    return result
```

3. Calculation part

This section presents the calculation part of the code **WASAoutputSimilarityTest_v2**. To have a clearer explanation we split this part into 3 steps: the preamble, the loop and the conclusion.

- **Preamble :** In this part of the calculation part, we define the number of cases we have studied. In our case this is 16. We also define the threshold which has to be in percentage ('threshold = 20' means 20%). Then, we open the file in which we will write the results and delete what is written inside. If this file does not exist it will be created. We start by writing an explanation of what the file is into the text file.

- **Loop :** This loop is a for loop where an index takes the case numbers. In our study, the cases are numbered from 1 to 16 so the loop will go from 1 to 16. We have to precise that, in the code, the index go from 1 to nb_cases+1 (so 17) because in Python the loop for i in range(n) takes n-1 as final index. The aim of this loop is to get the result of the comparison between the versions of the code WASA-SED for each case of study. The for loop permits to deal with all the cases. So, for each case, the following steps are followed:

  1. We first use the function **RUNexe** to get the output files of the new version.
  2. Then, we use the function **CreatePath** to get the path of the reference and the new version folders and **FileNamesList** to get the list of file names which are in the Output folders. Then, by using **IssueFilePresence**, we check if the two lists of file name are identical or not.

     If they are not identical, we will get a list of error messages and then the list of file names that we will finally compare.

     If they are identical, the only difference is that the list of error messages will be empty and the final list of file names will be identical to the ones got from the function **FileNamesList**.
  3. We write into the text file the name of the case we are studying and a paragraph named 'General issues' under which we will be able to read the possible error messages (cf Figure 2 in *1. General)*.
  4. We check if the error messages list from the function **IssueFilePresence**, is empty or not.

     If this is not empty, that means there is at least one error message, we first go through the list and write each error message into the text file. Then, we use the function **MainLoop** to get the comparison result of this case.

     If the list is empty, we directly use the function **MainLoop** to get the comparison result of this case.
  5. Now, we start to write the results in the text file, this is why we write a new paragraph 'data comparison result' into it. We analyze what the **MainLoop** function gave us. If we not get an empty list, that means there is an error in some files. So we write in the text file that there is an ERROR for this case and we precise for which files. If we get an empty list, that means there is no error for this case and we can write that everything is OK for this case.

- **Conclusion :** This calculation part ends by closing the text file and by writing into the Python console a message to inform where to find the results file.

```
""" CALCULATIONS """

nb_cases = 16 #number of cases we study
threshold = 20 #%


#os.chdir('G:/Documents/wasa_tests/WASA_pyProject') #already there because this is where the code is
resultTextFile = open("test_results", mode = 'w+')
resultTextFile.truncate()
resultTextFile.write('\nThis file presents the results of the similarity checking of the output files f
resultTextFile.write('\nNOTE: The file names which appear in the section -General issues- are not compa

for k in range(1,nb_cases+1):
    #RUNexe(k) #it runs the WASA-SED for each case of study
    pathRef, pathNew = CreatePath(k)
    listNamesRef = FileNamesList(pathRef)
    listNamesNew = FileNamesList(pathNew)
    IFPmessage, listOfName = IssueFilePresence(listNamesRef, listNamesNew)
    resultTextFile.writelines(['\n\n* Case ', str(k), ' :\n'])
    resultTextFile.write('   - General issues : \n\n')
    if IFPmessage != []:
        for l in IFPmessage:
            resultTextFile.writelines([l, '\n\n'])
        res = MainLoop(pathRef, pathNew,listOfName, threshold, resultTextFile)
    else:
        res=MainLoop(pathRef, pathNew, listOfName, threshold, resultTextFile)

    resultTextFile.write('\n   - Data comparison result : \n')
    if res != []:
        resultTextFile.writelines(['\n     ERROR =>', str(res), '\n'])
    else :
        resultTextFile.writelines(['\n     OK \n'])

resultTextFile.close()
print('__ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __\n')
print('The results of this similarity checking are presented is the file test_results saved in the fol
print('\n__ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __ __')
```

III. Future work

In this part I will present what we can do more for this code or what we can improve. The first thing regards the reading files and the second the creation of the Output folders.

1. Reading files

The principal issue I had was on the file reading. Indeed, the output files are not always organized in the same way which means we have to use different ways to read the files. I have solved the biggest part of the problem. Indeed, at the beginning the code was not able to read the files related to the lakes, the .stat files and others. Now, the current code is just not able to read the four followings files :

river_storage.stat
river_storage.stat_start
susp_sediment_storage.stat
susp_sediment_storage.stat_start

Indeed, they have more than two columns of data for only two column names which lead to an issue. I did not find a way to read them clearly. But, I have noticed that this is possible to read them by specifying some column names on the function ***pandas.read_csv()***. Let's have a look at the figure 5 to understand the point.

Example of a file *river_storage.stat*:

```
UHG routing: values of routed discharge per timestep of unit hydrograph
Subbasin        [n_h x timestep]
15      16.186  15.397  13.450  10.782  8.354   6.224   4.402   2.886   1.673   0.769
0.204   0.006   0.000   0.000   0.000   0.000   0.000   0.000   0.000
69      16.808  17.673  18.295  18.637  18.241  15.918  13.695  11.581  9.587   7.731
6.031   4.509   3.185   2.074   1.190   0.545   0.148   0.007   0.000
```

We can note that there is 20 columns for just two column names. So a way to read this file is to defines 20 column names. For example:

```
fileName = 'river_storage.stat'
columnNames = ['subbasin', 'c2', 'c3', 'c4', 'c5', 'c6', 'c7', 'c8', 'c9', 'c10', 'c11', 'c12', 'c13', 'c14', 'c15',
'c16', 'c17', 'c18', 'c19', 'c20']
df = pd.read_csv(fileName, 'b', engine='python', delimiter='\t', names=columnNames)
```

Figure 5: Idea number 1 to read the files.

Another idea, is to check if we can read the files in the row direction and so taking as column names '15' and '69'. I don't know if it is possible but it can be an idea.

2. Output folders creation

As noticed at the beginning of this report, before running the executable to get the output files for the new version for each case of study, the output folder has to exist. An improvement of the code can be to ensure that we can run the **WASAoutputSimilarityTest_v2** code even if the output folder do not exist.