11/4/2020

CLEERS Transient Data

User's Guide to Formatting and Python Code

Austin Ladshaw, Josh Pihl
OAK RIDGE NATIONAL LABORATORY

Contents

Introduction

All transient data is formatted as "tab separated" text files without a specified file extension. You can open the files from any text reading computer program and/or import the data directly into a spreadsheet program such as excel. However, the files contain on the order of ~10,000s of rows of data each, so it is not recommended to use excel to read or interpret the data without some form of pre-processing. Pre-processing algorithms are provided in the form of python scripts that can be used to read and interpret data, produce images and graphs, and/or reduce the data volume for post-processing in spreadsheet software such as excel.

In the Transient Data Formatting section of this guide, information on how each file is formatted, the naming conventions for files and headers, and the units of data values in each column is provided. The Python Code section discusses the python class objects, their functionality and usage, and a specialized python script for analyzing the NH_3 SCR storage data. Much of the python objects are reusable for other transient data CLEERS produces, while the script is specific to just the NH_3 SCR storage data.

Transient Data Formatting

All CLEERS transient data is intended to have similar formatting guidelines to ensure that the data collected can be read, parsed, and interpreted in the same or very similar manner. This section discusses the standards and conventions used in the data.

Folder Naming Conventions

A significant amount of information about the experiments from which the data was collected is contained within the names of the folders that hold a specific subset of the data. Generally, there are 3 distinct pieces of information that can be obtained.

- The catalyst material used
- The aging condition of the material
- The purpose of the experimental data set

Each piece of information in the folder name will be separated by a "-" to identify these specific characteristics.

For example, the folder named: "BASFCuSSZ13-700C4h-NH3storage" is parsed as…

- BASFCuSSZ13 = catalyst material name
  - This is a BASF Cu-SSZ-13 commercial catalyst
- 700C4h = aging conditions
  - This is for a catalyst hydrothermally aged at 700 oC for 4 hours
  - NOTE: this specific condition is considered "De-greened" or "Unaged"
- NH3storage = purpose of the experimental data
  - This indicates the data is for determining NH_3 storage capacities

All data folders will follow a similar naming convention.

File Naming Conventions

Similar to the folder names, the files contained within each folder will also follow a naming convention wherein each piece of data used to identify a particular experimental run will be a part of the name of the file separated by dashes ("-"). Typical types of information contained within the file name include (but are not limited to):

- Date of data collection
- Catalyst material used
- Aging conditions
- Type of experiment
- Gas flow rate in space-volumes per hour
- Concentrations (or concentration ranges) of important gas species
- Isothermal temperature of the experiment (or whether or not the run was a "by-pass")

The file names may contain some additional information as well, but these are the most critical pieces. Some of the concentration range data can be difficult to discern since dashes are also used to determine a range of concentrations. Provided below are several examples of file names and their interpretations.

Example 1: 20160205-CLRK-BASFCuSSZ13-700C4h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-150C

- 0^{th} position: 20160205 ➔ Experiment date: 02/05/2016
- 1^{st} position: CLRK ➔ Abbreviation of research group name
- 2^{nd} position: BASFCuSSZ13 ➔ Catalyst material: BASF Cu-SSZ-13
- 3^{rd} position: 700C4h ➔ Aging conditions: 700 ^{o}C for 4 hrs (De-greened)
- 4^{th} position: NH3DesIsoTPD ➔ Experiment type: Desorption and TPD data for NH_3
- 5^{th} position: 30k ➔ Flow Rate: 30,000 volumes per hour
- 6^{th} position: 0_2pctO2 ➔ Oxygen Concentration: 0.2 % O_2 in gas stream
- 7^{th} position: 5pctH2O ➔ Water Concentration: 5 % H2O in gas stream
- Last position: 150C ➔ Isothermal temperature: 150 ^{o}C

Example 2:

20160229-CLRK-BASFCuSSZ13-800C4h-NH3H2Ocomp-30k-0_2pctO2-11-3pctH2O-400ppmNH3-200C

- 0^{th} position: 20160229 ➔ Experiment date: 02/29/2016
- 1^{st} position: CLRK ➔ Abbreviation of research group name
- 2^{nd} position: BASFCuSSZ13 ➔ Catalyst material: BASF Cu-SSZ-13
- 3^{rd} position: 800C4h ➔ Aging conditions: 800 ^{o}C for 4 hrs
- 4^{th} position: NH3H2Ocomp ➔ Experiment type: Impact of H_2O competition on NH_3
- 5^{th} position: 30k ➔ Flow Rate: 30,000 volumes per hour
- 6^{th} position: 0_2pctO2 ➔ Oxygen Concentration: 0.2 % O_2 in gas stream
- 7-8^{th} position: 11-3pctH2O ➔ Water Conc. Range: Start @ 11 %, End @ 3 %
- 9^{th} position: 400ppmNH3 ➔ Ammonia Concentration: 400 ppmv
- Last position: 200C ➔ Isothermal temperature: 200C

Example 3: 20160205-CLRK-BASFCuSSZ13-700C4h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-bp

- 0^{th} position: 20160205 ➔ Experiment date: 02/05/2016
- 1^{st} position: CLRK ➔ Abbreviation of research group name
- 2^{nd} position: BASFCuSSZ13 ➔ Catalyst material: BASF Cu-SSZ-13
- 3^{rd} position: 700C4h ➔ Aging conditions: 700 ^{o}C for 4 hrs (De-greened)
- 4^{th} position: NH3DesIsoTPD ➔ Experiment type: Desorption and TPD data for NH_3
- 5^{th} position: 30k ➔ Flow Rate: 30,000 volumes per hour
- 6^{th} position: 0_2pctO2 ➔ Oxygen Concentration: 0.2 % O_2 in gas stream
- 7^{th} position: 5pctH2O ➔ Water Concentration: 5 % H2O in gas stream
- Last position: bp ➔ Indicates the “by-pass” run for these isothermal sets

NOTE on "bp": The last position of every file name gives either the isothermal temperature for the experimental run or gives "bp" to indicate that this is the "by-pass" experiment for these experiment types (i.e., a run of the gases through the reactor and equipment without the catalyst present). All data sets at a series of isothermal temperatures will also have 1 "bp" file that those experiments pair with. The by-pass file is used as the baseline concentration record without reactions and storage taking place. Integrating the difference between the by-pass concentration record and a specific isothermal temperature concentration record gives a measure for the amount of a gas species stored on the catalyst. Thus, the by-pass files are necessary to estimate NH_3 storage in each experiment.

Types of Files

Each data folder contains data files for experiments collected at a specific isothermal temperature and a paired data file representing the by-pass data for the experimental runs. You can determine which file is which by looking at the last position of the file name (see File Naming Conventions for examples). Each by-pass file will have the exact same file name as the files it pairs with except for that last position entry in the file name. If a folder is missing the appropriate by-pass file, then the data is incomplete.

File Structure

Regardless of the file type and/or file name, the data contained within the files follows a common formatting and structure. That formatting is as follows:

Rows

- The 1st row of each file is a header that just states where the data came from and the reactor in which the experiment took place.
- The 2nd row of each file contains the names of the data in each column. *For example*:
  - Time = Actual time stamp for when the measurement was made
  - Elapsed Time (min) = Time (in min) that has passed since the experiment started
  - NH3 (300) = Measure NH3 concentration at the exit in ppmv
  - (More information provided in Columns and Data Units sections)
- Starting from the 3rd row, the data corresponding to the column name is provided, i.e., the actual values for elapsed time, ammonia concentration, etc.
- **PERIODICALLY**: After the 3rd row, there will occasionally be a repeat of the column names (i.e., the 2nd row information). This repeat is used to denote when an experimental input change in the gases and/or temperature has occurred.
  - For example, in the NH_3 desorption and TPD data sets, the 2nd row repeats are used to identify the times at which the inlet concentrations of NH_3 have changed to start the adsorption, desorption, or TPD section of the experiment.
  - This is an important indicator in the data as it signals where and when this experimental data set needs alignment with the corresponding by-pass data set.

Columns

Each column contains the set of data gathered for the specific measurement or corresponding column name from the 2nd row of the file. Plotting any column set versus the "Elapsed Time (min)" column shows the time evolution of that data set during the experiment. Not all columns contain useful information for an experimental run or contain information that is necessary for analysis. The following is an explanation of the data contained within each column as identified by the column name:

- Time:
  - Actual time stamp for the measurement (mm/dd/yy hh:mm:ss)
- Elapsed Time (min):
  - Amount of time (in min) that has passed since measurements started
- NH3 (300):
  - Measured exit concentration of NH_3 in ppmv
  - Number in parenthesis indicates the sensitivity of the instrument
    - i.e., good for measuring up to ppmv of 300
- NH3 (3000):
  - Measured exit concentration of NH_3 in ppmv
  - Number in parenthesis indicates the sensitivity of the instrument
    - i.e., good for measuring up to ppmv of 3000

NOTE: When two columns measure the same thing (like above), they are often done with equipment/instruments that have a different sensitivity. Use the data column that most closely corresponds to the instrument sensitivity (without exceeding the sensitivity limit).

- Ethylene (100,3000):
  - Measured exit concentration of Ethylene in ppmv
  - Numbers in parenthesis that are separated by commas indicate the combined sensitivity of the instruments taking measurements to the maximum value
    - i.e., good for measuring up to a ppmv of 3000
- H2O% (20):
  - Measured exit concentration of H_2O in volume %
  - Number in parenthesis indicates the sensitivity of the instrument
    - i.e., good for measuring up to volume % of 20
- TC bot sample in (C):
  - Temperature measured at the front/inlet and centerline to the catalyst in ^oC
- TC bot sample mid 1 (C):
  - Temperature measured in the middle and centerline of the catalyst in ^oC
- TC bot sample mid 2 (C):
  - Temperature measured in the middle and wall of the catalyst in ^oC
- TC bot sample out 1 (C):
  - Temperature measured at the back/outlet and centerline of the catalyst in ^oC
- TC bot sample out 2 (C):
  - Temperature measured at the back/outlet and wall of the catalyst in ^oC
- P bottom in (bar):
  - Inlet pressure at front of catalyst in bar

- DP bottom (bar):
  - Pressure drop at exit of catalyst in bar
- P bottom out (bar):
  - Outlet pressure at exit of catalyst in bar
- valve 0:
  - Position/state of the 0^{th} valve
  - Either "TRUE" or "FALSE" for the "opened" or "closed" position
  - Same for all other "valve #" and "4-way valve #" columns
- MFC1: 100 % N2 in N2 (carrier) [6140 sccm]:
  - Mass Flow Controller 1
  - Provides info on the gas species and carrier gas the controller provides
  - Numbers in square brackets are the flow rates of the gas for this controller
    - Units of flow rate are in Standard cubic centimeters per minute
  - Same information for all other "MFC#:" columns
- vapor from pump1 [393 sccm]:
  - Vapor pump flow controllers provide same information as "MFC#" columns
  - Same for all "vapor from pump#" columns

NOTE: Not every column name is discussed above, but you can use the explanation of each column name as a guide to determine what other column names of a similar structure provide.

NOTE: Many of the other thermo-couple (TC) columns are left out as they are associated with the system controls and would not have a direct impact or meaning to the concentration observations.

Data Units

Each column has its own units dictated by the column names. In general, the concentrations will be in either ppmv or volume %, the temperatures will be in °C, pressures will be in bars, and time will be in minutes. Occasionally, the units can be obtained from the column names (i.e., provided in parenthesis as part of the column names). See Columns for detailed information on column names, units, and interpretations.

Python Code

A series of python data/class objects and python scripts are provided to aid in the reading, parsing, compiling, reducing, and interpretation of the CLEERS transient data sets. The python objects are intended to be generic enough to work with any transient data set produced by CLEERS work, while the NH3_H2O_data_processing.py script is specific for data interpretation of the series of NH3 SCR storage experiments. Users can produce their own scripts using the developed python objects (TransientData, PairedTransientData, TransientDataFolder, and TransientDataFolderSets) to parse or process the data in other ways if desired.

All the python code is documented using Doxygen, which is used to create a developer's reference manual is distributed with the code. That reference manual (refman.pdf) contains more details regarding all the object functions, options, and methods that is supplementary to this document. Therefore, the information about the python code in this document is limited to basic user functionality and usage examples. Please refer to the reference manual (refman.pdf) that should also be distributed with these codes. You may also find a permanent link to that reference manual here:

https://bitbucket.org/gitecosystem/ecosystem/src/master/scripts/CLEERS/TimeSeriesData/doc/refman.pdf

Recommended Python Version

The python codes provided have been tested and work with Python 3.7, thus it is recommended you use this version or newer.

Required Libraries

The python scripts use the following "import" statements, thus those libraries are required:

- transient_data.py
  - math
  - os
  - sys
  - statistics
  - random
  - matplotlib.pyplot
  - scipy.optimize
  - scipy.stats

- transient_data_set.py
  - os
  - sys
  - getopt
  - mpl_toolkits.mplot3d
  - matplotlib
  - numpy

- NH3_H2O_data_processing.py
  - os
  - sys
  - getopt
  - time

TransientData Object

This class object is used to read in a single transient data file, without pairing that data file to a particular by-pass run. It will automatically extract information from the file name, including: (i) the catalyst name, (ii) the aging conditions, (iii) the experimental run type, (iv) the total gas flow rate, and (v) the isothermal temperature (or identify whether the given file is a by-pass run). See File Naming Conventions for how this information is organized. For data that needs to be paired to a by-pass data file, use the PairedTransientData object instead of this object.

The constructor for this class object requires the user provide the file name that is to be read in. That file, if it exists, is then automatically read by the constructor (i.e., there is no need to call the "readFile" or "closeFile" functions for the object after its construction). Once that file is read, all the data contained therein is stored inside of a python "dictionary" structure named "data_map". This map can be directly queried by the user or the user can use other object functions to access specific map information. Each key in the map is the name of a column in the data file. NOTE: those key names can change when the data is processed or compressed. Under each key is a list of all the data in that column in the order it appears in the original data file. Access into maps in python is given by the key name, while access into lists in python is given by offset positions indexed starting from 0.

Typically, you will not want to use direct access into the "data_map" structure to get specific information from the file. Instead, you should consider using built-in access functions to extract specific information. Below is a listing of the most common and useful data accessing functions.

Data Accessing Functions

- **extractColumns(column_list)**
  - This function returns a map object that contains only the data for the list of specified columns
  - column_list can either be a single column name or a list of column names whose data you want to extract into a new, smaller map
  - If a column name is invalid, the function will print out an error message
- **extractRows(min_time, max_time)**
  - This function returns a map object that contains only the data between a specified range of "Elapsed Time (min)" values for all columns in the original data_map structure
  - min_time is the lower time value, i.e., where to start extraction
  - max_time is the upper time value, i.e., where to stop extraction
- **getDataPoint(time_value, column_name)**
  - This function returns the value in the specified data column that corresponds to the given time value

    - More than likely, this is the primary data access function you will want to use as it will give you exactly the individual piece of data you want
    - time_value is the value for "Elapsed Time (min)" for which you are wanting to extract data
    - column_name is the name of the column that you want time stamped data from
    - If time_value is between two times in the actual data map, then the routine will linearly interpolate the data between the two nearest time stamps.
- **getMaximum(column_name)**
    - Returns the maximum value of the corresponding column
    - column_name is the name of the column you want the maximum value of
- **getMinimum(colum_name)**
    - Returns the minimum value of the corresponding column
    - column_name is the name of the column you want the minimum value of
- **getTimeFrames()**
    - Returns a list of tuples containing the start and end times where the inlet conditions for the experimental run was changed
    - During typical desorption and TPD experiments, the inlet concentrations of the adsorbing gas are stepped down over a specific period of time. These "time frames" are saved internally and can be obtained using this function.

Data Access Examples

In this example, we are reading in a file named "20160205-CLRK-BASFCuSSZ13-700C4h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-150C.dat" to extract the rows of data for the last time frame only. **Note**: the last time frame for TPD runs contains the actual TPD curves.

First, read in the file by using the TransientData constructor and save the object in variable name TPD.

```
TPD = TransientData("20160205-CLRK-BASFCuSSZ13-700C4h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-150C.dat")
```

Next, save the last time frame in a tuple name final_frame. **Note**: the last object in a list can be directly accessed in python using an index value of -1.

```
final_frame = TPD.getTimeFrames()[-1]
```

Then, extract the rows of data for times only within the desired time frame from final_frame. **Note**: start time in the time frame is accessed at index 0 and the end time in the time frame is accessed at index 1.

```
tpd_map = TPD.extractRows(final_frame[0], final_frame[1])
```

The keys of the new "tpd_map" are the same as the original "data_map" structure, so you can use those keys to extract columns. For instance, if you only want the ammonia data for the TPD time frame, you can extract that information into separate lists that you can manipulate later.

```
time_list = tpd_map["Elapsed Time (min)"]

nh3_list = tpd_map["NH3 (300)"]
```

Data Compression Functions

In addition to generic data access functions, the object also has several functions built-in to compress the data into smaller chunks. This is particularly useful when you want to parse and compile a data file to reduce the total number of data points to something more manageable in your productivity software of choice, such as MS Excel. Below are some of the most common data compression functions.

- **compressColumns()**
  - This function will automatically go through all the columns of data and eliminate or combine redundant information.
  - Recall from File Structure → Columns that several measurements are taken with different instruments of different sensitivities. Each column contains the same data, but at different levels of accuracy. This function combines those columns into a single column and always takes the most accurate of each measurement.
  - **NOTE**: if and when columns are compressed, the data is stored in a new column name and the old column names are discarded. The new column name is made of a combination of the column names from which they came.
    - For example, compressing the columns "NH3 (300)" and "NH3 (3000)" will result in a new column named "NH3 (300,3000)"
- **deleteColumns(column_list)**
  - Delete the given columns from the "data_map"
  - column_list can either be a single column name or a list of column names whose data you want deleted
  - Use this function to get rid of columns of information you do not want or need
- **retainOnlyColumns(column_list)**
  - More often than not, there will be more columns to delete than there are to keep. Thus, you can use this function to direct which columns you want to keep, rather than the ones you want deleted.
  - column_list can either be a single column name or a list of column names whose data you want to keep, the rest will be deleted
  - **NOTE**: you should always keep "Elapsed Time (min)" as one of your columns
- **compressRows(factor = 2)**
  - This function will compress the rows of data by a specific factor.
  - The factor represents the level of compression to use.
    - Default = 2
    - Averages every "factor" number of values into a new compressed value
  - **NOTE**: This function changes the names of the columns by adding "(factor)x Compression" to the end of each column name. This is done such that after processing the user will know whether or not the data was compressed and by how much was it compressed.
  - Typically, this should always be the last compression operation you do in your analysis because of the name changes and the loss of precision.

Data Compression Example

In this example, we are reading in a file named "20160205-CLRK-BASFCuSSZ13-700C4h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-150C.dat" and performing the standard data compression techniques to reduce the amount of data we have to deal with.

First, read in the file by using the TransientData constructor and save the object in variable name TPD.

```
TPD = TransientData("20160205-CLRK-BASFCuSSZ13-700C4h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-150C.dat")
```

Next, compress the columns of the data set to remove information redundancy.

```
TPD.compressColumns()
```

Then, for this example, we are only interested in the "Elapsed Time (min)" and "NH3 (300,3000)" data columns. **Note**: That before we called compressColumns(), the ammonia data was stored in 2 separate columns: (i) "NH3 (300)" and (ii) "NH3 (3000)". If at any point, you are unsure of the column names, you can call the function "displayColumnNames()" to get a printout of all currently valid column names in the map.

```
TPD.displayColumnNames()    # This to console all column names that are currently valid

TPD.retainOnlyColumns(["Elapsed Time (min)", "NH3 (300,3000)"])
```

Finally, we want to compress the row information to a more manageable size. For this, it is often useful to know how large the data set is currently. To get the current data size, you can call the "getNumRows()" function to find out the current number of rows in the "data_map". In this example, there are on the order of 10,000 rows of data, so we decide to compress with a factor of 5 to reduce to around 2,000 rows of data.

```
print( TPD.getNumRows() )        # Gives the number of rows in the map and prints to console

TPD.compressRows(5)
```

Data Manipulation Functions

For more advanced users, you might be interested in performing some specific data manipulations prior to printing out information to files or creating plots from the processed data sets. Most common manipulations would be things like unit conversions or summations of columns or integrals over time. Below are some functions that you may find useful.

- **mathOperation(column_name, operator, value_or_column, append_new = False, append_name = "")**
  - This is a powerful, generic function that allows you to apply vector or scalar operations to a given column and override the values in that column or save the new result in a new column that is added to the "data_map"
  - column_name is the name of the column that you are applying the operation to
  - operator is a string value that is a symbol representing what mathematical operation to perform

        - Options are as follows:
            - "*" = multiply
            - "/" = divide
            - "+" = add
            - "-" = subtract
    - value_or_column can be a scalar value or another column name whose value is applied to column_name with the given operator
        - For instance:
            - if value_or_column is 5 and the operator is "*", then each entry in the given column_name is multiplied by 5
            - if value_or_column is "C1" (i.e., another column name) and the operator is "-", then the given column_name value is reduced by the "C1" column value for each row.
    - append_new is a Boolean argument to determine whether or not to create a new column in the map to store the operation result or just override the original column_name with the result
        - if append_new = False, then the values in column_name are overridden
        - if append_new = True, then a new column is created to hold the result
    - append_name is the name of the new column to hold the result of the operation (if that option is choosen)
        - if append_new = True, and this option is blank, then a default new column name is provided
- **calculateIntegralSum(column_name, min_time=None, max_time=None)**
    - This function calculates a simple numerical integration and returns that total integrated value of the given column name over the specified time range.
    - column_name is the name of the column over which to perform the integral
    - min_time is the starting time for the integration
        - If not given, then assumes the start time is the initial time
    - max_time is the ending time for the integration
        - If not given, then assumes the end time is the final time

Data Manipulation Example

In this example, we are reading in a file named "20160205-CLRK-BASFCuSSZ13-700C4h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-150C.dat" and using the "mathOperation" function to convert a temperature column from °C to K and to convert a pressure column from bar to kPa.

First, read in the file by using the TransientData constructor and save the object in variable name TPD.

```
TPD = TransientData("20160205-CLRK-BASFCuSSZ13-700C4h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-150C.dat")
```

Next, apply a unit conversion to the "TC bot sample in (C)" column to produce a new column named "T in (K)" with temperatures in Kelvin.

```
TPD.mathOperation("TC bot sample in (C)", "+", 273.15, True, "T in (K)")
```

Lastly, apply a unit conversion to the “P bottom in (bar)” column to produce a new column named “P in (kPa)” with pressure in kilopascals.

```
TPD.mathOperation(“P bottom in (bar)”, “*”, 100, True, “P in (K)”)
```

Data Output Functions

The most common and widely used functionality of the python scripts is to perform pre-processing of the transient data so that the data can be parsed, compiled, and simplified in a manner that allows individuals to then use their productivity software of choice (such as MS Excel) to perform other post-processing or analysis of the data. The data output functions are meant to be used in conjunction with the Data Compression and Data Manipulation functions so that all actions performed can be saved and written to output files that can be read in by other software. There are also output functions to create python plots automatically for the data. Below are the most common data output functions:

- **printAlltoFile(file_name = “”)**
  - Prints all the information in the “data_map” object in a similar formatting to how the data was originally stored.
  - **NOTE**: because of the similarity in output formatting, this function is generally used after performing some data manipulations and compressions to reduce the file size down to something more manageable.
  - file_name is the name that you want to give the output file
- **savePlots(range = None, folder = “”, file_type = “.png”)**
  - Saves plots of all data columns for a specified time range, in a specified folder, with a specified image file type.
  - range is a tuple representing the start and end times for the range of data you want to plot versus time
    - if range = None, then the entire time range is used
  - folder is the name of the folder in which to save all the plots
    - if not given, then a folder name will be chosen based on the original file name that was read in by the object
    - Each plot in that folder will be named by their respective column names
  - file_type is a string identifying the image file type to save the plots as
    - Default = “.png” files
    - Valid options:
      - “.png”
      - “.pdf”
      - “.ps”
      - “.eps”
      - “.svg”
- **saveTimeFramePlots(folder = “”, file_type = “.png)**
  - Calls the savePlots() function for all ranges in the “time_frames” list dictated automatically by the data in the original data file
  - file_type is the same as above

Data Output Example

In this example, we will culminate all the above examples to demonstrate how to read in transient data, perform some of the most common preprocessing, and print out all the processed results into an output file.

First, read in the file by using the TransientData constructor and save the object in variable name TPD.

```
TPD = TransientData("20160205-CLRK-BASFCuSSZ13-700C4h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-150C.dat")
```

Next, compress the columns of the data set to remove information redundancy.

```
TPD.compressColumns()
```

Then, for this example, we are only interested in the "Elapsed Time (min)" and "NH3 (300,3000)" data columns. **Note**: That before we called compressColumns(), the ammonia data was stored in 2 separate columns: (i) "NH3 (300)" and (ii) "NH3 (3000)". If at any point, you are unsure of the column names, you can call the function "displayColumnNames()" to get a printout of all currently valid column names in the map.

```
TPD.displayColumnNames()    # This to console all column names that are currently valid

TPD.retainOnlyColumns(["Elapsed Time (min)", "NH3 (300,3000)"])
```

Before we compress the rows, we should perform any other data manipulations we want to the original data set. For instance, the ammonia concentrations are originally in ppmv, however, in this example we want to convert the concentrations to mol/L at standard pressure and the given isothermal temperature (150 ^{o}C ➔ 423.15 K). This conversion involves the usage of ideal gas law and a series of calls to "mathOperations", then a deletion of the old column.

```
TPD.mathOperation("NH3 (300,3000)", "/",1E6)          # ppmv to molefraction

TPD.mathOperation("NH3 (300,3000)", "*",101.35)       # molefraction to kPa at std pressure

TPD.mathOperation("NH3 (300,3000)", "/",8.3144)       # partial pressure divided by gas const

# final conversion factor to mol/L saved into a new column

TPD.mathOperation("NH3 (300,3000)", "/",423.15, True, "NH3 (mol/L)")

TPD.deleteColumns("NH3 (300,3000)")                   # delete the old column
```

Next, we want to compress the row information to a more manageable size. For this, it is often useful to know how large the data set is currently. To get the current data size, you can call the "getNumRows()" function to find out the current number of rows in the "data_map". In this example, there are on the order of 10,000 rows of data, so we decide to compress with a factor of 5 to reduce to around 2,000 rows of data.

```
print( TPD.getNumRows() )       # Gives the number of rows in the map and prints to console

TPD.compressRows(5)
```

Lastly, we will print the pre-processed and newly compressed data to an output file.

```
TPD.printAlltoFile()
```

PairedTransientData Object

The PairedTransientData object has numerous similarities with the TransientData object. Most all functions of TransientData have equivalent functions in this object. Generally, if there is a by-pass file provided for an experimental run, then you should always "pair" that data to a complimentary by-pass data file. This is useful for looking at comparisons between by-pass concentration histories and non-by-pass data histories, which can be used to formulate the storage capacities as a retention integral between the by-pass curve and the data curves.

The constructor for this object requires 2 file names be given: (i) the name of the by-pass file and (ii) the name of the results file. When the constructor is called, the code will automatically check the names of the given files to determine whether or not those files can actually be paired together. If they cannot, then the code reports and error to the console. Each file is then read in using the TransientData object's constructor and each object is stored into sub-objects (bypass_trans_obj and result_trans_obj) for each file.

To reduce redundancy in this manual, any function that exists in TransientData that also exists in this object will not be discussed in great detail. Instead, the primary focus here will be on functions in PairedTransientData that are different from those of TransientData.

Data Accessing Functions

- **extractBypassColumns(column_list)**
  - Same as **extractColumns** from TransientData, but action is specific to the by-pass data only.
- **extractResultColumns(column_list)**
  - Same as **extractColumns** from TransientData, but action is specific to the result data only.
- **extractRows(min_time, max_time)**
  - Same as **extractRows** from TransientData, but action is performed of both the by-pass and result data.
- **getBypassDataPoint(time_value, column_name)**
  - Same as **getDataPoint** from TransientData, but action is specific to the by-pass data only.
- **getResultDataPoint(time_value, column_name)**
  - Same as **getDataPoint** from TransientData, but action is specific to the result data only.
- **getTimeFrames()**
  - Same as **getTimeFrames** from TransientData, but action is performed of both the by-pass and result data. Only one list of time frames is returned.

Data Accessing Example

In this example, we are reading in a file named "20160205-CLRK-BASFCuSSZ13-700C4h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-150C.dat" and pairing that data to the corresponding file named "20160205-CLRK-BASFCuSSZ13-700C4h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-bp.dat". **Note**: the constructor will automatically know whether or not these data sets are pairable based on the file names. The by-pass file will have all the same name except for the last argument before the file type.

First, read in the file by using the PairedTransientData constructor and save the object in variable name pTPD. **Note**: the by-pass file name must be given first.

```
pTPD = TransientData("20160205-CLRK-BASFCuSSZ13-700C4h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-bp.dat", "20160205-CLRK-BASFCuSSZ13-700C4h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-150C.dat")
```

All accessing data functions for the paired data are used exactly the same as the unpaired data. Please see TransientData → Data Accessing Example for usage examples.

Data Compression Functions

Nearly all of the data compression functions for the paired object are identical to TransientData. However, in addition to the standard compression functions, there is a data alignment function that is responsible for making sure that the by-pass data is aligned in time and concentration levels with the result data it is being paired with. This function is extremely important and MUST be called BEFORE doing any Data Manipulations. The object has a natural check flag that knows whether or not the data has been aligned and will report errors if you try to manipulate data before performing the alignment.

- **alignData(addNoise = True)**
  - This function will perform the arduous task of aligning all the data in time and relative concentration levels. Data alignment is a necessary step because the by-pass runs often take less time that the actual experimental runs. Thus, the results act as the master set or anchor in time by which the by-pass information must be set and aligned to.
  - addNoise is a Boolean that determines whether or not you want to add a simulated instrument noise to the by-pass data set. This is set to True by default since the simulated noise does well to fill in data gaps in the by-pass set.
  - **Note**: this function can take a few minuets to run if the data set is large and you specify to add in the simulated instrument noise
  - **Note:** After alignment, all by-pass data will have column names changed to column_name+"[bypass]" and stored into the data_map for the result object.
    - For example, if both the by-pass data and result data have a column named "NH3", then after alignment the result column name remains "NH3", but the by-pass column name becomes "NH3[bypass]"
- **compressColumns()**
  - This function is identical to **compressColumns** from TransientData.
- **deleteColumns(column_list)**
  - This function is identical to **deleteColumns** from TransientData.

- **retainOnlyColumns(column_list)**
  - This function is identical to **retainOnlyColumns** from TransientData.
- **compressRows(factor = 2)**
  - This function is identical to **compressRows** from TransientData.

Data Compression Example

In this example, we are reading in a file named "20160205-CLRK-BASFCuSSZ13-700C4h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-150C.dat" and pairing that data to the corresponding file named "20160205-CLRK-BASFCuSSZ13-700C4h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-bp.dat". We will compress the columns, then specify a set of columns that we want to retain and align the remaining data.

First, read in the file by using the PairedTransientData constructor and save the object in variable name pTPD. **Note**: the by-pass file name must be given first.

```
pTPD = TransientData("20160205-CLRK-BASFCuSSZ13-700C4h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-bp.dat", "20160205-CLRK-BASFCuSSZ13-700C4h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-150C.dat")
```

Next, we will compress the columns and request to retain only the "Elapsed Time (min)" and "NH3 (300,3000)" columns. **Note**: That before we called compressColumns(), the ammonia data was stored in 2 separate columns: (i) "NH3 (300)" and (ii) "NH3 (3000)". If at any point, you are unsure of the column names, you can call the function "displayColumnNames()" to get a printout of all currently valid column names in the map.

```
pTPD.compressColumns()

pTPD.displayColumnNames()    # This to console all column names that are currently valid

pTPD.retainOnlyColumns(["Elapsed Time (min)", "NH3 (300,3000)"])
```

Lastly, before doing anything else, we MUST call the alignData() function. After alignment, we will have three distinct column names in the data map: (i) "Elapsed Time (min)", (ii) "NH3 (300,3000)", and (iii) "NH3 (300,3000)[bypass]". **ALSO NOTE**: after this function call, all data is stored in the result data object (i.e., result_trans_obj), thus, you will have no further use for any functions that are specific to the by-pass data.

```
pTPD.alignData()
```

Data Manipulation Functions

Prior to performing any data manipulation with PairedTransientData, you must call the **alignData()** function for the object (see Data Compression Functions above). All data manipulation is performed on the result data, not the by-pass data. The most common data manipulation functions are provided below.

- **mathOperation(column_name, operator, value_or_column, append_new = False, append_name = “”)**
    - This function is the same as **mathOperation** from TransientData.
- **calculateIntegralSum(column_name, min_time=None, max_time=None)**
    - This function is the same as **calculateIntegralSum** from TransientData.
- **calculateRetentionIntegral(column_name, normalized = False, conv_factor = 1, input_col_name=””)**
    - This function computes a new column for the mass retained in the reactor (M_R) based on the integration of the following function:
        - $\frac{d(M_R)}{dt} = Q \cdot (M_{in} - M_{out})$
            - M_R = mass retained in system
            - M_{in} = mass into the system
            - M_{out} = mass out of the system
            - Q = space-volume flow rate ($time^{-1}$) through system
    - column_name is the name of the column that represents the mass or concentration at the exit of the system (M_{out})
    - normalized is a Boolean to determine whether or not to normalize the new computed mass retained (M_R) column normalized to a maximum value of 1
    - conv_factor is a multiplicative factor you can applied to the calculated result column (M_R) to convert units of the result to a desired unit basis
    - input_col_name is the name of the column that represents the mass or concentration at the inlet of the system (M_{in})
        - If left blank, then the inlet column is assumed to be column_name+”[bypass]”
    - You can use this function to compute the mass or concentration time series of a gas species adsorbed by the catalyst
        - The resulting column will have the same units as those of M_{in} and M_{out}, unless you normalize the column or provide a conversion factor
    - **Note**: the resulting integral column will be added to the result’s data_map and given a name after the column_name as follows:
        - column_name+”-Retained (normalized)”, if normalized = True
        - column_name+”-Retained”, if normalized = False

Data Manipulation Example

This data manipulation example will follow directly after the example from Data Compression Example. After we have aligned the data, we have a column for the exit ammonia concentration history (“NH3 (300,3000)”) and the inlet concentration history (“NH3 (300,3000)[bypass]”). Thus, we have the information we need to perform a retention integral calculation to know how much mass of ammonia was retained in the catalyst, i.e., was adsorbed.

First, we will calculate the basic retention integral. **Note**: the result of the integral will have the same units as the units of the columns we used to calculate the integral. For “NH3 (300,3000)” columns, this means our mass retained will be calculated in ppmv.

```
pTPD.calculateRetentionIntegral("NH3 (300,3000)")
```

This now produces a new column in our data named "NH3 (300,3000)-Retained" with units of ppmv. We can perform additional operations to this column to convert to other units if desired. For instance, we may want to have the mass retained in something like moles of NH3 adsorbed per volume of catalyst. This can be done in a series of **mathOperation** steps.

First series of steps, convert ppmv to total moles of NH3 retained using ideal gas law and the total volume of the reactor. For these experiments, there is a total reactor volume of 0.015708 L (based on the total dimensions of the system).

```
pTPD.mathOperation("NH3 (300,3000)-Retained", "/",1E6)        # ppmv to molefraction
pTPD.mathOperation("NH3 (300,3000)-Retained", "*",101.35)    # molefraction to kPa
pTPD.mathOperation("NH3 (300,3000)-Retained", "/",8.3144)    # divide by gas const
pTPD.mathOperation("NH3 (300,3000)-Retained", "/",423.15)    # convert to mol/L
pTPD.mathOperation("NH3 (300,3000)-Retained", "*",0.015708) # multiply by total volume
```

Last step, divide the total moles of ammonia retained in system by the volume of just the catalyst. In this example, the catalyst takes up about 66.9 % of the total reactor volume, we divide total moles by 0.01051 L (the catalyst volume). In addition, we will store this new result in a new column name and delete the old column name.

```
# divide by catalyst volume and save result in new column
pTPD.mathOperation("NH3 (300,3000)-Retained", "/",0.01051, True, "NH3 ads (mol/L)")
# delete the old retention column, since it is not needed anymore
pTPD.deleteColumns("NH3 (300,3000)-Retained")
```

Data Output Functions

The output functions in PairedTransientData are all the same as those of TransientData. The only difference is that they will also output, or create plots, for all the columns with their paired by-pass data columns.

- **printAlltoFile(file_name = "")**
  - Same as in TransientData
- **savePlots(range = None, folder = "", file_type = ".png")**
  - Same as in TransientData
- **saveTimeFramePlots(folder = "", file_type = ".png)**
  - Same as in TransientData

Data Output Example

All output functions are the same. See TransientData for examples.

TransientDataFolder Object

Recall that the transient data is distributed in folders (Folder Naming Conventions) that contains several data files all with some common thread (i.e., same materials, same aging conditions, and same experimental purpose). Thus, it will be advantageous to be able to parse, compile, and process all data contained within a single data folder. This object is designed specifically for the purpose of performing data compression, data manipulations, and producing outputs from all data within a single common folder. You can even perform specific data access by providing the name of the file whose data you want to access.

This object's constructor is invoked by providing the name of the folder that contains all the data files that you want to compress, manipulation, or compile. For instance, you can invoke the constructor as follows:

```
folder_data = TransientDataFolder("BASFCuSSZ13-700C4h-NH3storage")
```

This command will open the folder named "BASFCuSSZ13-700C4h-NH3storage" and automatically iterate through and read all the data files contained within. The constructor is designed in such a way that it will know whether or not there is a result data file that is to be paired with a corresponding by-pass file without any additional user intervention. In addition, the constructor will also automatically perform the **compressColumns()** and **alignData()** functions (See PairedTransientData for details) to immediately prepare all the data for further compression or operations.

After calling the constructor, all the data is now stored into your local variable (folder_data from above) and you can then use any other access, manipulation, or compression functions desired.

Data Accessing Functions

In general, there are no direct data access functions for this object. This is because data access is different for each file in the folder. However, you can gain access to sub-class data structure for those files, then call any data access function for that object. Below are the helper functions that allow you to gain access to a specific file's data object, which can then be used to call other data accessing functions.

- **grabFileList()**
  - Returns a list of file names in the folder
  - Access to specific data is done through the **grabDataObj()** function below, which requires you provide the name of the file you want to access information from
- **grabDataObj(file)**
  - Returns a reference to a PairedTransientData object or TransientData object depending on the data type for the given file
  - file is the name of the file you want to access

Data Accessing Example

To access any specific data from a file, you use the **grabDataObj()** function followed by a call to whatever other data access function from PairedTransientData or TransientData you want. For example, we read in all the files from a folder named "BASFCuSSZ13-700C4h-NH3storage" and want to extract out

the time frames for the TPD sections (i.e., the last time frame) for the file in that folder named "20160205-CLRK-BASFCuSSZ13-700C4h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-150C".

First, call the TransientDataFolder constructor.

```
folder_data = TransientDataFolder("BASFCuSSZ13-700C4h-NH3storage")
```

Then, grab the last time frame from the desired file.

```
final_frame = folder_data.grabDataObj("20160205-CLRK-BASFCuSSZ13-700C4h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-150C").getTimeFrames()[-1]
```

Note: Since the **grabDataObj()** function returns a reference to the sub-class objects of PairedTransientData and TransientData, this gives you access to any and all functions from those two objects (i.e., any manipulations, compressions, or output functions you want).

Data Compression Functions

The TransientDataFolder constructor automatically calls the **compressColumns()** and **alignData()** functions for all data files, however, there are several more data compressions that you may want to perform. While you can use the **grabDataObj()** function (see Data Access above) to perform compressions for individual files, you can also use the functions below to perform similar compressions *en masse* to all files in the folder.

- **deleteColumns(column_list)**
  - This function is identical to **deleteColumns** from TransientData, but performs the action for all files
- **retainOnlyColumns(column_list)**
  - This function is identical to **retainOnlyColumns** from TransientData, but performs the action for all files
- **compressAllRows(num_rows_target = 1000, max_factor = 10)**
  - This function is similar to the **compressRows** function from TransientData, but performs the action for all files and automatically chooses a compression factor for each file based on the arguments for num_rows_target and max_factor
  - num_rows_target represents approximately how many rows of data you want each file to have after performing the data compression
    - Default = 1000 rows
  - max_factor is the maximum compression factor which is not to be exceeded for any file, regardless of the num_rows_target
    - Default = 10x compression

Data Compression Example

This example will continue from Data Access Example and will demonstrate how to reduce the columns we want to just "Elapsed Time (min)", "NH3 (300,3000)", "H2O% (20)", and "TC bot sample out 1 (C)". See File Structure → Columns for more information on the interpretation of the column names.

These columns represent the experimental time, ammonia concentration, water concentration, and exit temperature histories, respectively. Then, we will compress that data for further analysis later.

First, direct the folder_data object to retain only the columns we want.

```
folder_data.retainOnlyColumns(["Elapsed Time (min)", "NH3 (300,3000)", "H2O% (20)", "TC bot sample out 1 (C)"])
```

Then, use the row compression option to compress the data down to around 2000 rows each with a maximum compression factor of 5.

```
folder_data.compressAllRows(2000,5)
```

Note: Generally, you want to hold off on compressing data rows till after data manipulation, but before data output.

Data Manipulation Functions

All the data manipulation functions are used the same as those from PairedTransientData, however, they may have slightly different names or return slightly different arguments.

- **mathOperations(column_name, operator, value_or_column, append_new = False, append_name = "")**
  - This function is the same as **mathOperation** from PairedTransientData.
- **calculateIntegralSum(column_name, min_time=None, max_time=None)**
  - This function is the same as **calculateIntegralSum** from PairedTransientData.
  - The one difference is that it returns a map of the integral sums rather than a single value. The keys of the map are the names of the files in the folder.
- **calculateRetentionIntegrals(column_name, normalized = False, conv_factor = 1, input_col_name="")**
  - This function is the same as **calculateRetentionIntegral** from PairedTransientData, except is has a slightly different name and performs that action for all files in the folder.

Data Manipulation Example

The usage of these functions are essentially identical to the usage in PairedTransientData. See those manipulation examples for reference. Pay attention to the above names of the functions. They may be different from those of PairedTransientData.

Data Output Functions

There a host of new output functions for this object that give new levels of control to the types of plots you can produce from the data. All outputs are saved in folders that you can provide a custom name for or allow the code to choose a folder name based on the name of the folder you read the data from and the types of data you are outputting to the output folder.

- **printAlltoFile(subdir = “”)**
  - This function will print all output from all files of the folder that was read in to another folder of the given name.
  - subdir is the name of the output folder
    - If left blank, the name of the output folder will be the name of the input folder + “-Output”
  - An output file is placed in the output folder for each input file in the input folder, except for the by-pass files since they are paired.
- **savePlots(range = None, folder = “”, file_type = “.png”)**
  - Same as in PairedTransientData, except action is performed for all files in the input folder.
  - folder is the name of the output folder to save the plots in
    - If left blank, the name of the output folder will be the name of the input folder + “-Plots”
- **saveTimeFramePlots(folder= “”, file_type = “.png”)**
  - This function just performs the same action as **savePlots** from above, but does so for each time frame individually
    - Useful for viewing individual adsorption desorption curves and/or the TPDs by themselves
  - Both the folder and file_type arguments are same as above
- **saveOverlayPlots(column_name, second_column=None, folder= “”, base=None, condition = “iso_temp”, file_type = “.png”)**
  - This function saves a series of “overlay plots” for the given column name based on a variable condition given.
    - An “overlay plot” will plot all columns given across the different related files in the folder in the same plot. It can be used to visualize how the given column changes based on the given condition.
    - **For example**, we may want to view the TPD curves for ammonia at all isothermal temperatures on the same figure to see how the variations in isothermal temperature impact the TPD profiles
  - column_name is the name of the column we want the plots to be of
  - second_column is the name of the column that we want to plot the first given column_name against
    - **For example**, we may want to plot ammonia concentration versus temperature in the TPD, thus we would provide the name of the temperature column for this argument.
    - If left blank/None, then the code will automatically plot the column_name values versus the time column
  - folder is the name of the output folder to save the plots in
    - If left blank, the name will automatically be the name of the input folder + “-OverlayPlots”

- base is the “base name” of the files for which we are plotting the overlays
  - Data folders may contain more than 1 type of data files. For example, the same folder may hold “NH3DesIsoTPD” type files and “NH3H2Ocomp” type files. See File Naming Conventions for examples.
  - If base is left as None, then the same actions are performed for all file types in the folder
- condition is the experimental condition that each of the common file types have as a difference between their collected data
  - **For example**, in the ammonia storage data folders, each data file is suffixed with the isothermal temperature of the experiment. Thus, that represents the different experimental condition for each file. See File Naming Conventions for details.
  - Valid condition options include:
    - “iso_temp” = isothermal temperature
    - “material” = catalyst material
    - “aging_time” = hours the material was aged for
    - “aging_temp” = temperature at which it was aged
    - “flow_rate” = space-volume flow rate of the experiments
  - You should only provide a condition option that is actually representative of the condition that changes in the data
- file_type is the file extension for the plots
  - Valid options include:
    - “.png”
    - “.pdf”
    - “.ps”
    - “.eps”
    - “.svg”

Data Output Example

This example continues from the Data Compression Example and demonstrates how to create and save overlay plots for ammonia TPD curves. Here, we want to plot the ammonia concentration versus the reactor exit temperature for all the different isothermal experimental conditions. The other options in the function call are left to their defaults, so the output folder name will be chosen automatically.

Note: the default condition for the overlay plot function is the isothermal temperature condition (see above Data Output Functions for details).

```
folder_data.saveOverlayPlots(“NH3 (300,3000)”, “TC bot sample out 1 (C)”)
```

TransientDataFolderSets Object

The final python class object for CLEERS transient data build on all prior objects. It is used to automatically read and interpret data in a series of CLEERS data folders that all have some common thread. For instance, the ammonia storage data is saved in a series of folders named the following:

- "BASFCuSSZ13-700C4h-NH3storage"
- "BASFCuSSZ13-800C2h-NH3storage"
- "BASFCuSSZ13-800C4h-NH3storage"
- "BASFCuSSZ13-800C8h-NH3storage"
- "BASFCuSSZ13-800C16h-NH3storage"

All of the above data folders contain the same types of data, but for different hydrothermal aging conditions. Because of that commonality in data, each folder can be treated as from the same base set of experimental work and thus be manipulated, parsed, and interpreted using common python functions and actions.

To use this specialized object, each of the above indicated folders of data should be saved into a common subdirectory. Then, we can pass a list of the names of the folders from above to the TransientDataFolderSets constructor to automatically read all data from the above folders. Below demonstrates how to call that constructor:

```
all_data = TransientDataFolderSets(["BASFCuSSZ13-700C4h-NH3storage", "BASFCuSSZ13-800C2h-NH3storage", "BASFCuSSZ13-800C4h-NH3storage", "BASFCuSSZ13-800C8h-NH3storage", "BASFCuSSZ13-800C16h-NH3storage"])
```

After calling the constructor, all data in all files has been read into a series of TransientDataFolder objects and automatically had all columns compressed and all data aligned. At this point, you can now start calling the data access, data compression, data manipulation, and data output functions.

Data Accessing Functions

The access to the data in this object can be granted in the same way as TransientDataFolder, as long as you know the names of the files in each folder. If the names of the files are unknown, you can grab the file names by first using the **grabFolderObj()** function (passing the folder name of interest), then calling the **grabFileList()** function from the corresponding TransientDataFolder object that gets returned.

- **grabFolderObj(folder)**
  - This function returns reference to the TransientDataFolder object corresponding to the folder that contains the data that was read in.
  - folder is the name of the folder that you want to grab information from
    - It is presumed that the user should know the folder names that are valid as they would have to have been given during object construction
  - Use this function in conjunction with the **grabFileList()** or other data access functions from TransientDataFolder object.

- **grabDataObj(file)**
    - Returns a reference to a PairedTransientData object or TransientData object depending on the data type for the given file
    - file is the name of the file you want to access, which should be a unique name in each folder that was read in
        - The function will automatically search through all folders for the given file name and return reference to the appropriate object

Data Accessing Example

After calling the constructor (see TransientDataFolderSets for constructor call), you can use the specific access functions to get into the data for a specific folder and/or specific file. Each of those access functions returns a reference to another TransientData or PairedTransientData object (or TransientDataFolder object in the case of **grabFolderObj**), which then gives you the ability to save that object into a temporary data structure or directly call other corresponding data object functions.

In this example, we will demonstrate how to access a specific file object from a specific folder. To do this, we will perform a cascading call of functions to grab a data object for a specific file, then save that data object in a temporary variable called file_obj.

```
file_obj = all_data.grabFolderObj("BASFCuSSZ13-800C8h-NH3storage").grabDataObj("20160303-CLRK-BASFCuSSZ13-800C8h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-300C")
```

Note: The above code goes into the "BASFCuSSZ13-800C8h-NH3storage" folder and returns the "20160303-CLRK-BASFCuSSZ13-800C8h-NH3DesIsoTPD-30k-0_2pctO2-5pctH2O-300C" data object, which is an instance of the PairedTransientData object and stores that object into file_obj.

At this point, we can now use other Data Access Functions from TransientData or PairedTransientData to go deeper into the files to grab specific information of interest. For instance, we can return the list of time frames from that particular file and save that list of tuples into a variable named tuple_list.

```
tuple_list = file_obj.getTimeFrames()
```

Data Compression Functions

All of the data compression functions are identical to those of TransientDataFolder, but are applied to all folders in the given set of folders.

- **deleteColumns(column_list)**
    - This function is identical to **deleteColumns** from TransientDataFolder
- **retainOnlyColumns(column_list)**
    - This function is identical to **retainOnlyColumns** from TransientDataFolder
- **compressAllRows(num_rows_target = 1000, max_factor = 10)**
    - This function is identical to **retainOnlyColumns** from TransientDataFolder

Data Compression Example

All calls to the data compression functions for this object are identical to TransientDataFolder, but they get applied to all files in all folders given.

Data Manipulation Functions

All data manipulation functions for this object are identical to TransientDataFolder (including the names of those functions), but they get applied to all files in all folders given.

- **mathOperations(column_name, operator, value_or_column, append_new = False, append_name = “”)**
  - This function is the same as **mathOperations** from TransientDataFolder.
- **calculateIntegralSum(column_name, min_time=None, max_time=None)**
  - This function is the same as **calculateIntegralSum** from TransientDataFolder.
- **calculateRetentionIntegrals(column_name, normalized = False, conv_factor = 1, input_col_name=””)**
  - This function is the same as **calculateRetentionIntegrals** from TransientDataFolder.

Data Manipulation Example

All calls to the data manipulation functions for this object are identical to TransientDataFolder, but they get applied to all files in all folders given.

Data Output Functions

Most all data output functions for this object are identical to TransientDataFolder, but they get applied to all files in all folders given. In addition, there are some “cross overlay plot” functions also added to allow the user to plot curves of the same column in each data file across all data folders against each other. For instance, the folders that were read in from the above example were all at different aging times, ranging from “De-greened” to 16 hours of hydrothermal aging. Thus, it may be useful to create a plot to show the ammonia TPD curves at a specific isothermal condition (like 150 ^{o}C) for all aged and unaged catalyst samples. The “cross overlay plot” functions allow for this functionality.

- **printAlltoFile(subdir = “”)**
  - This function is the same as **printAlltoFile** for TransientDataFolder, but is applied to all folders given.
- **savePlots(range = None, folder = “”, file_type = “.png”)**
  - This function is the same as **savePlots** for TransientDataFolder, but is applied to all folders given.
- **saveTimeFramePlots(folder= “”, file_type = “.png”)**
  - This function is the same as **saveTimeFramePlots** for TransientDataFolder, but is applied to all folders given.

- **saveOverlayPlots(column_name, second_column=None, folder= “”, base=None, condition = “iso_temp”, file_type = “.png”)**
    - This function is the same as **saveOverlayPlots** for TransientDataFolder, but is applied to all folders given.
- **saveCrossOverlayPlots(column_name, subdir= “”, rtype=None, const_cond = “iso_temp”, var_cond = “aging_cond”)**
    - This function will plot the given column_name versus the time column at the given const_cond for each different var_cond across all folders
    - column_name is the name of the column data you want plotted
    - subdir is the name of the output folder to store the results in
        - If left blank, then a name will automatically be chosen and given a suffix of “-CrossOverlayPlots”
    - rtype is the experimental run type of data that you want plotted
        - **For example**, recall from the File Naming Convention that the file name item in the 4th position indicates the type of run.
            - From above, this includes “NH3DesIsoTPD” and “NH3H2Ocomp”
            - By providing one of these string identifiers, the code will know which common data set to apply this action to.
        - If left blank, this action is performed for all run types
    - const_cond is the condition that is to be constant or common among all curves being plotted.
        - **For example**, provide the option “iso_temp” if you want all the curves plotted to be at the same isothermal temperature
        - Valid options include:
            - “iso_temp”
            - “material”
            - “aging_temp”
            - “aging_time”
            - “flow_rate”
            - “aging_cond”
        - Make sure that the condition you give is valid for the data sets your are reading and interpreting.
            - **For instance**, in the ammonia data sets, the only really valid option to consider is “iso_temp” since all data for all other conditions is already the same across all data
    - var_cond is the condition that you want to vary for the sets of plots
        - **For example**, if you want to visualize how the aging conditions change the TPD profiles for ammonia capture at 150 oC, then you would select “aging_cond” for this option (and “iso_temp” for const_cond).
        - Valid options → Same as const_cond above

Data Output Example

In this example, we continue from our constructor call (see TransientDataFolderSets) to demonstrate how to produce "cross overlay plots" for all isothermal temperatures plotting the ammonia TPDs at different aging conditions for each isothermal temperature in each folder. Those plots will be saved in a default output folder by passing a blank string for the subdir argument. The string for rtype is set to "NH3DesIsoTPD" to denote that we are only interested in TPD curves to plot. See **saveCrossOverlayPlots** from Data Output Functions for additional option details.

```
all_data.saveCrossOverlayPlots("NH3 (300,3000)", "", "NH3DesIsoTPD")
```

NH3_H2O_data_processing.py

The TransientData, PairedTransientData, TransientDataFolder, and TransientDataFolderSets objects are generic objects intended to be used by an end user to read in any CLEERS transient data provided. Usage of these objects can be accomplished in any live python session or your own scripts for future data. Make sure you use the proper import statements to gain access to these objects and their methods within your own codes.

For less adept python users, the NH3_H2O_data_processing.py script is our in-house script using the above objects to perform some standard data processing for any transient ammonia storage data we produce. This script is to be run using command line options that provide instructions on the names and locations of data folders, as well as the names and locations for output folders after data processing. This script is used to produce the following output:

- All data in all files is reduced to just experimental time, ammonia concentration, water vapor concentrations, temperature probe data, and inlet/outlet pressures.
- Retention integrals for total ammonia storage histories is computed in moles ammonia storage per liter of catalyst.
- Water retention is also calculation, however, this data is noisy and not very informative.
- Data is compressed to roughly 1000 rows for each file.
- All types of plots are created and saved into separate folders
- All output is printed to new output text files post-compression, which allows users to import the reduced data into productivity software like MS Excel

Usage of the Script

The script is invoked from command line or a bash terminal as follows…

```
bash >:        python NH3_H2O_data_processing.py -i input_dir/ -o output_dir/
```

The "input_dir/" is a path and folder name that contains all the folders of the ammonia storage data (i.e., the folders whose names were given in TransientDataFolderSets must all be contained within one folder named "input_dir"). The "output_dir/" is an optional argument which is the path and name of output folder that will contain all the output results from the script.

NOTE: This script takes roughly 30 min – 1 hr to complete, depending on the amount of data.