netl-ap-map-flow Documentation

Release 0.1.0

Matthew Stadelman

Jul 14, 2017

Contents

netl-ap-map-flow is a modeling suite written in Fortran and Python3 to perform local cubic law (LCL) simulations of single phase flow through a discrete fracture and analyze the data. Several tools written in Python provide added functionality are packaged in the **apmapflow** module. Dependencies are managed using Anaconda through conda-forge. Paraview is the recommended program to visualize the output using the legacy vtk files. The CSV output files can be visualized in ImageJ, Excel, etc. However, depending on how your chosen program reads in the image matrix, the image may appear inverted. The first value in the CSV files corresponds to bottom left corner of the fracture. After installation several scripts are avilable under with the prefix `apm_`.

Table 1: **Summary of apmapflow submodules**

data_processing	Provides an easy to use and extendable platform for post-processing a set of simulation data.
openfoam	Implements an interface to create simulation cases for OpenFoam.
run_model	Run the LCL model and manipulate input files programmatically.
unit_conversion	Provides a unit conversion API powered by pint

CHAPTER 1

Installation

Quick Install Using Anaconda

First install the Python3 version of Anaconda or Miniconda for your given platform and allow it to modify your `PATH` variable. Then run the following set of commands in a terminal. You can use the module directly in scripts by running `import apmapflow` or simply work with the scripts provided. A full list of scripts and basic usage is shown in the documentation section below.

```
conda config --add channels conda-forge
conda config --add channels stadelmanma
conda update -y conda
conda update -y python
conda install netl-ap-map-flow
```

Install as a Developer

To develop the package you will need to download Anaconda or Miniconda as above. Additionally, you will need to ensure that git, a Fortran compiler (I use gfortran) and the make program are installed and available on your path. When using Windows it is recommended you make a copy of the `mingw32-make.exe` (or similarly named) executable and rename it `make.exe`.

The following set of commands can be used in a terminal window to download and setup the package once the aforementioned programs have been installed.

```
conda config --add channels conda-forge
conda update -y conda
conda update -y python
git clone https://github.com/stadelmanma/netl-ap-map-flow.git
cd netl-ap-map-flow
conda install --file requirements.txt
pip install -r test_requirements.txt
python setup.py develop
```

```
python ./bin/build_model -h
python ./bin/build_model
```

CHAPTER 2

Basic Usage of LCL Model

Running the model in a terminal:

```
apm_run_lcl_model  model_initialization_file
```

Full usage instructions can be found in examples/running-the-flow-model.html.

CHAPTER 3

Documentation

3.1 Usage Examples

Contents:

Running the Flow Model

Contents

Intro

The Local Cubic Law (LCL) flow model by default is named `apm-lcl-model.exe`, the added extension doesn't affect UNIX systems and allows Windows to recognize it as executable. There are two methods to run the model, first is directly on the command line via the script `apm_run_lcl_model` specifying your input parameters file(s) and the second is creating custom scripts using the `run_model` sub-module in `apmapflow`. The model axes are on the X-Z plane where +Z is vertical and +X is to the right. The Y direction is the aperture variation and the model assumes a planar mid surface. This implies that the fracture is symmetric with respect to the X-Z plane. If you envision a spiral bound notebook with the bottom left corner as the origin. The Z-axis follows the metal spiral upward, positive X is the direction moving away from the spiral. Y is the thickness of the notebook.

The Input Parameters File

A template of the input parameters file can be found in here. The input file is parsed by a special routine in the model that treats all text after a semicolon, `;`, as comments and one or more spaces as delimiters. A value enclosed in double quotations will have any internal spaces ignored. apmapflow also reads the input files when needed and for consistency it is recommended you append a colon `:` onto the end of keywords. Nearly all of the input parameters have default values and they are defined in APERTURE_MAP_FLOW.F in the subroutine `INITIALIZE_RUN`

Important Notes

- There must be at least one space separating keyword and value.
- The names of the keywords can not be changed without editing the model's source code.
- Physical quantities must have a unit associated with them.
- The LCL model's unit conversion is independent of the Python module and considerably more restrictive.

To use a diferent version of the LCL model executable than packaged with the module you can add a commented out line reading `;EXE-FILE: (exe-file-name)`. The code will assume the executable path is relative to the location of the input file itself, **not** the current working directory if they differ.

The input file can be broken down into four main components:

File Paths

This is where the path of input and output files are defined. Files are opened relative to the current working directory, **not** the input file itself. The use the absolute paths can prevent any ambiguity. If the line `OVERWRITE EXISTING FILES` is uncommented, any existing files sharing the same name will be replaced. If the line is commented out or absent an error will be raised if a pre-existing file with the same name is encountered. The model assumes the aperture map data being input is stored as a 2-D grid with the first value being the origin and the last value being the top right corner. The data maps output by the model also follow this convention. The output files accept the extension the user provides however the expected extensions are noted below. File names are not required and any omitted files will be saved using a default name.

Input Files

- `APER-MAP`: File that stores the aperture map data.

Output Files

- `SUMMARY-FILE`: Logs the information printed to the screen (.txt expected)
- `STAT-FILE`: Stores statistics calculated and certain simulation parameters. Provided extension is replaced by `.csv` and `.yaml`
- `APER-FILE`: Stores a copy of the input aperture map that is converted to the desired output units and has any applied roughness factored in (.csv expected)
- `FLOW-FILE`: Used as the root name for the three flow files output: X-component, Z-component and magnitude. The files have -X, -Z, -M suffixes appended to the root name before the extension. (.csv expected)
- `PRESS-FILE`: Stores a pressure distribution map of the fracture(.csv expected)
- `VTK-FILE`: A legacy formatted input file for Paraview which combines all of the former input files and includes several other data maps. Provided extension is replaced by `.vtk`

Boundary Conditions

Defines the boundary conditions for the simulation. The model does no internal checking to see if the BCs supplied create a valid solvable problem. It is the user's resposiblity to ensure a valid combination is supplied; two Dirichlet (pressure) conditions or one Dirichlet and one Neumann (flow rate) condition on the inlet and outlet.

- `INLET-PRESS`: value unit ;The pressure value to use at the inlet
- `INLET-RATE`: value unit ;The flow rate to apply at the inlet
- `OUTLET-PRESS`: value unit ;The pressure value to use at the outlet
- `OUTLET-RATE`: value unit ;The flow rate to apply at the outlet.
- `OUTLET-SIDE`: [LEFT, RIGHT, TOP, BOTTOM] sets the outlet side, the inlet is assumed to be the opposite face. i.e. top is outlet, bottom is inlet

Model Properties

- `FLUID-DENSITY`: value unit ;Density of liquid to use in Reynolds number calculation
- `FLUID-VISCOSITY`: value unit ;Viscosity of liquid
- `MAXIMUM MAP DIMENSION`: value ;Maximum number of blocks along either axis (default 2000)

Other Parameters

- `MAP AVERAGING FACTOR`: value ;The number of voxels required to span an edge of a grid block along the X or Z direction. Grid blocks are assumed square in the X-Z plane.
- `VOXEL SIZE`: value unit ;Specifies the voxel to meter conversion factor

- `ROUGHNESS REDUCTION:` value ;**The value is in voxels** Amount to symmetrically bring the front and back fracture surfaces together by.
- `CALCULATE PERCENTILES:` value1,value2,value3, ..., valueN ;A comma separated list of percentiles to calculate for various quantities during runtime. Commenting this line out tells it to not calculate them at all
- `HIGH-MASK:` value ;**The value is in voxels** All data values in the aperture map above this value will be reduced to this value.
- `LOW-MASK:` value ;**The value is in voxels** All data values in the aperture map below this value will be raised to this value

This tells the model what units you want the data output in. Commenting out or omitting this line will output everything in SI (pascals, meters and meters^3/second)

- `OUTPUT-UNITS:` pressure unit, distance unit, flow rate unit

Running the Model

This guide will assume you setup the modeling package using setup.py which would have installed the script `apm_run_lcl_model` on the console PATH and build the model from source code. Alternatively, you can use the actual executable in place of the script for these examples. Before we actually run the model it will be helpful to have a place to store the output files generated. We need to define an input file to use with the model and in this case we will take advantage of many of the predefined defaults. Running the following code in a terminal while in the top level directory (netl-ap-map-flow) will get things started, and show the usage information for the script.

```
mkdir model-testing
cd model-testing
touch model-input-params.inp
# display usage info
apm_run_lcl_model -h
```

Open model-input-params.inp with your favorite text editor and copy and paste the following block.

```
;
; FILE PATHS AND NAMES
APER-MAP: ../examples/Fractures/Fracture1ApertureMap-10avg.txt
;SUMMARY-FILE:
;STAT-FILE:
;APER-FILE:
;FLOW-FILE:
;PRESS-FILE:
;VTK-FILE:
;OVERWRITE EXISTING FILES
;
; BOUNDARY CONDITIONS
INLET-PRESS: 100 PA
OUTLET-PRESS: 0 PA
OUTLET-SIDE: TOP
;
; MODEL PROPERTIES
FLUID-DENSITY: 1000.0 KG/M^3
FLUID-VISCOSITY: 0.890 CP
;
; OTHER PARAMETERS
MAP AVERAGING FACTOR: 10.0
```

```
VOXEL SIZE: 25.0 MICRONS
CALCULATE PERCENTILES: 0,1,5,10,15,20,25,30,40,50,60,70,75,80,85,90,95,99,100
;
; DEFINE SPECIFIC OUTPUT UNITS TO USE
; REQUIRED FIELD ORDER: PRESSURE,DISTANCE,FLOW RATE
OUTPUT-UNITS: PA,MM,MM^3/SEC
```

Running Directly

With the above steps complete running the model is as simple as this:

```
apm_run_lcl_model model-input-params.inp
```

You will notice that several output files have been generated in the current directory. They are saved under the default names because we did not specified our own filenames in the input file. You can view the VTK file in Paraview and the other CSV data maps in your viewer of choice. The STATS file is not a data map but being saved as a CSV file allows for quick calculations in excel or similar software. The YAML version of the stats file provides an easy to parse format for programmic manipulation, such as using the `apm_combine_yaml_stat_files` script to coalesce the results of multiple simulations. If we try to run the model a second time as before line again you will see an error is generated and execution is terminated. This is because the line `;OVERWRITE EXISTING FILES` is preceded by a semicolon meaning it is commented out and by default existing files will not be overwritten.

Running in Python Scripts

The run_model sub-module allows for much more power and convenience when running the model or multiple instances of the model. The sub-module also houses the BulkRun class which can be used to automate and parallelize the running of many simulations. Usage of the BulkRun class is outside the scope of this example file and is gone over in depth in this file.

The core components of the RunModule consist of one class used to manipulate an input parameters files and two functions to handle running of the model. Code snippets below will demonstrate their functionality. The examples here assume you are working with the files created at the beginning of the section *Running the Model*. The first step is to run the Python interpreter and import them from the parent module. You will only be able to import the module if you used setup.py to install the it, or manually added it to a location on the Python path, i.e. site-packages.

```
import os
from apmapflow.run_model import InputFile
from apmapflow.run_model import estimate_req_RAM, run_model
```

The InputFile Class

The InputFile class is used to read, write and manipulate an input parameters file. It provides an easy to use interface for updating parameters and can dynamically generate filenames based on those input parameters. Use of the class is simplest when you have the code read a file with all of the possible parameters already entered and the unneeded ones commented out.

Notes:

- The keywords of the input file class are the first characters occurring before *any* spaces on a line. The keyword for parameter `OVERWRITE EXISTING FILES path/to/filename` is `OVERWRITE`

Argument - Type - Description

- infile - String or InputFile - The path to the file you want to read or the variable storing the InputFile object you want to recycle.
- filename_formats (optional) - dict - A dict containing filename formats to use when creating outfile names and the save name of the input file itself based on current params. If none are provided then the original names read in will be used.

```
# Creating an InputFile object
inp_file = InputFile('model-input-params.inp', filename_formats=None)

# updating arguments can be done three ways
#inp_file['param_keyword'] = value
#inp_file['param_keyword'] = (value, commented_out)
#inp_file.update(dict_of_param_values)

# Directly updating the viscosity value
inp_file['FLUID-VISCOSITY'] = '1.00'

# updating a set of parameters using a dictionary
new_param_values = {
    'OVERWRITE': 'OVERWRITE FILES',
    'INLET-PRESS': '150.00'
}
inp_file.update(new_param_values)

# printing the InputFile object shows the changes
print(inp_file)
```

You will notice that the line `OVERWRITE EXISTING FILES` has been changed and uncommented. The class by default will uncomment any parameter that is updated. To update a value and set the commented_out boolean use a tuple `(value, True/False)`. This will also work when creating a dictionary of values to update. Parameters are stored in their own class called ArgInput which can be directly manipulated by accessing the keyword of an InputFile object like so, `inp_file['FLUID-VISCOSITY']`. New parameters can not be created by simple assignment, instead you must call the `add_parameter` method and pass in the full line intended to go in the file, or a suitable placeholder line.

```
# commenting out percentile parameter
inp_file['CALCULATE'].commented_out = True

# changing the unit and value of density
inp_file['FLUID-DENSITY'].unit = 'LB/FT^3'
inp_file['FLUID-DENSITY'] = '62.42796'

# adding a new parameter to the input file
inp_file.add_parameter('NEW-PARAMETER: (value) (unit)')

# changing the new param and making the line commented out
inp_file['NEW-PARAMETER'] = ('NULL', True)

#
print(inp_file)
```

In addition to updating arguments you can also apply a set of filename formats to the InputFile class. These allow the filenames to be dynamically created based on the argument parameters present. Using the `update` method of the InputFile class you can also add a special set of args not used as parameters but instead to format filenames. Any args passed into `update` that aren't already a parameter are added to the `filename_format_args` attribute of the class.

```
# setting the formats dict up
# Format replacements are recognized by {KEYWORD} in the filename
name_formats = {
    'SUMMARY-FILE': '{apmap}-summary-visc-{fluid-viscosity}cp.txt',
    'STAT-FILE': '{apmap}-stat-visc-{fluid-viscosity}cp.csv',
    'VTK-FILE': '{apmap}-vtk-visc-{fluid-viscosity}cp.vtk'
}

# recycling our existing input file object
inp_file = InputFile(inp_file, filename_formats=name_formats)
inp_file.update({'apmap': 'avg-frac1'})

# showing the changes
print(inp_file)
```

The name of the input file can also be altered with formatting by adding an `input_file` entry to the filename_formats_dict. An entry in the filename_formats_dict will overwrite any changes directly make to the `.outfile_name` attribute of the InputFile class. The default outfile name is the name of the parameters file being read, so the original file would be overwritten.

The estimate_req_RAM Function

The estimate_req_RAM function estimates the maximum amount of RAM the model will use while running. This is handy when running large maps on a smaller workstation or when you want to run several maps asynchronously.

Argument - Type - Description:

- input_maps - list - A list of filenames of aperture maps.
- avail_RAM (optional) - float - The amount of RAM the user wants to allow for use, omission implies there is no limit on available RAM.
- suppress (optional) - boolean - If set to True and too large of a map is read only a message is printed to the screen and no Exception is raised. False is the default value.

Returns a list of required RAM per map.

```
# setting the maps list
maps = [
    os.path.join('..', 'examples', 'Fractures', 'Fracture1ApertureMap-10avg.
↪txt'),
    os.path.join('..', 'examples', 'Fractures', 'Fracture2ApertureMap-10avg.
↪txt'),
    os.path.join('..', 'examples', 'Fractures', 'Fracture1ApertureMap.txt'),
    os.path.join('..', 'examples', 'Fractures', 'Fracture2ApertureMap.txt'),
]

#checking RAM required for each
estimate_req_RAM(maps, 4.0, suppress=True)
```

```
#raises EnvironmentError
estimate_req_RAM(maps, 4.0)
```

Because suppress was true we only received a message along with the amount of RAM each map would require. However the last line generates an error.

The run_model Function

The run_model function combines some higher level Python functionality for working with the system shell into a simple package. The model can be both run synchronously or asynchronously but in both cases it returns a Popen object. Running the model synchronously can take a long time when running large aperture maps.

Argument - Type - Description

- input_file_obj - InputFile - the input file object run with the model. Note: This file has to be written to disk, be careful to not overwrite existing files by accident
- synchronous (optional) - boolean - If True the function will halt execution of the script until the model finishes running. The default is False.
- show_stdout (optional) - boolean - If True then stdout and stderr will be printed to the screen instead of being stored on the Popen object as stdout_content and stderr_content

```
# running our current input file object
# synchronous is True here because we need the process to have completed␣
↪for
# all of stdout to be seen.
proc = run_model(inp_file, synchronous=True, show_stdout=False)

# proc is a Popen object and has several attributes here are a few␣
↪useful ones
print('PID: ', proc.pid) # could be useful for tracking progress of␣
↪async runs
print('Return Code: ', proc.returncode) # 0 means successful
print('Standard output generated:\n', proc.stdout_content)
print('Standard err generated:\n', proc.stderr_content)
```

Another instance where running the model synchronously is helpful would be running data processing scripts after it completes.

Using the BulkRun Class

Contents

Intro

The BulkRun class housed in the run_model submodule allows the user to setup a test matrix where all combinations of a parameter set can be tested taking advantage of multiple cores on the computer. It relies heavily on the core methods and classes in the run_model submodule and it is recommended that you go through the example running-the-flow-model before trying to use the script to be familiar with how the code will work behind the scenes. In addition to the core methods a special class is used to facilitate running a test matrix, BulkRun. It is also recommended you view the source of the class to understand the flow of the program. Lastly the script apm_bulk_run.py can be used to process a bulk run for you by supplying one or more YAML formatted input files to it. An example YAML file is displayed below.

Using the apm_bulk_run Script

Usage

The usage of the apm_bulk_run script is very simple because it only needs to parse YAML parameter files using the yaml module which can be installed through pip via the `pyyaml` package. YAML files are read in as series of nested dictionaries which the BulkRun module is designed to use. Any number of YAML files can be read in at once however, the first YAML file sets up the BulkRun class and the others are only used to generate additional InputFile instances from.

Full usage information can be obtained by using the `-h` flag, `./apm_bulk_run -h`. More runtim information can be obtained by adding the `-v` flag

By default the script does a dry run, the `--start` flag needs to be added to actually begin simulations. Below is an example of calling the script

```
# doing a dry run first
apm_bulk_run -v group-run1.yml group-run2.yml

# actually running the simulations
apm_bulk_run -v --start group-run1.yml group-run2.yml
```

Example YAML File

```
# initial model input file to use as a template
initial_input_file: './model-input-file.inp'

# keyword arguments passed onto the BulkRun __init__ method
bulk_run_keyword_args:
    spawn_delay: 1.0  # delay in starting new individual simulations
    retest_delay: 5.0  # time to wait between checks for completed sims
```

```
    sys_RAM: 8.0 # amount of RAM allocated for simulations
    num_CPUs: 4 # number of CPUs allocated for simulations

# filename formats to use when building filenames based on input parameters
default_file_formats:
    APER-MAP: './maps/{stage}_ApertureMapRB{map_type}.txt'
    SUMMARY-FILE: './{sim-type}/{stage}/{stage}{map_type}-inlet_rate_{INLET-
↪RATE}ml_min-log.txt'
    STAT-FILE: './{sim-type}/{stage}/{stage}{map_type}-inlet_rate_{INLET-
↪RATE}ml_min-stat.csv'
    APER-FILE: './{sim-type}/{stage}/{stage}{map_type}-inlet_rate_{INLET-
↪RATE}ml_min-aper.csv'
    FLOW-FILE: './{sim-type}/{stage}/{stage}{map_type}-inlet_rate_{INLET-
↪RATE}ml_min-flow.csv'
    PRESS-FILE: './{sim-type}/{stage}/{stage}{map_type}-inlet_rate_{INLET-
↪RATE}ml_min-press.csv'
    VTK-FILE: './{sim-type}/{stage}/{stage}{map_type}-inlet_rate_{INLET-RATE}
↪ml_min.vtk'
    input_file: './{sim-type}/inp_files/{stage}{map_type}-inlet_rate_{INLET-
↪RATE}ml_min.inp'


# parameter lists to combine when generating individual InputFile
default_run_parameters:
    OUTLET-PRESS: [0.00]
    INLET-RATE: [1.00, 2.00, 4.00, 6.00, 10.00]
    MAP: [1]
    ROUGHNESS: [0.0]
    OUTPUT-UNITS: ['PA,M,M^3/SEC']
    VOXEL: [26.8]
    # these are not model inputs and only affect filename formatting
    stage:
    - 'IS-6_0'
    - 'IS-6_1'
    - 'IS-6_2'
    - 'IS-6_3'
    - 'IS-6_4'
    sim-type: ['const-flow']
    map_type: ['-full', '-10avg']

# format string used to identify specific cases based on parameters
case_identifier: '{map_type}'

# parameters for each desired identifier value
case_parameters:
    -10avg:
    MAP: ['10']
    OUTLET-PRESS:   # empty key-value pair used to unset a run parmeter
```

Block or Flow styling may by used based on the user preference, in this example flow style sequences were chosen for parameters and block style for mapppings because of readability and compactness. The block style list for the `stage` keyword was done to denote significance.

Although all values are converted to strings before use in the InputFile instances, values in the YAML file are not *required* to be quoted. However, adding quotes can be safer when a value contains characters that may confuse/error the YAML parser such as the value for `OUTPUT-UNITS`. Without quotes it would be interpreted as a list instead of a string, producing an invalid entry in the InputFile. Additional higher level YAML functionality can likely be used but has not been tested. The basic syntax is used here for a clear

and concise example.

The BulkRun Class

This class wraps up the core functionality contained in the run_model submodule into a format allowing easy processing of a test matrix in parallel. The Local Cubic Law (LCL) model itself is not parallelized however this limitation is overcome by calling the `run_model` function in asynchronous mode and capturing the screen output produced. The class accepts many arguments during instantiation but the only required argument is an initial InputFile instance to clone. The InputFile instance acts as a template for all subsequent runs. All arguments that are being varied need to be present even if they only contain a dummy value. The block below shows accepted arguments and defaults.

```
bulk_run = BulkRun(input_file, # Used as a template to generate simulation␣
↪runs from
                   num_CPUs=2.0, # Maximum number of CPUs to try and use
                   sys_RAM=4.0, # Maximum amount of RAM to use
                   **kwargs)
```

Useful kwargs and defaults are:

- `spawn_delay=5.0`: minimum time between spawning of new processes
- `retest_delay=5.0`: time to wait between checking for completed processes

You can manually supply a list of InputFile instances to the class by assigning them to the `bulk_run.input_file_list` attribute. However the better method is to use the `bulk_run.generate_input_files` method which will be explained in detail next. When running the simulations the program considers the available RAM first and then if there is enough space it will check for an open CPU to utiltize. The RAM requirement of an aperture map is an approximation based on a linear relationship with the total number of grid blocks. The code will only seek to use 90% of the supplied value because the LCL model occasionally carries a small fraction of additional overhead which can not be predicted. The order that simulations are run may differ from the order of the input_file_list. This is because the code will loop through the list looking for a map small enough to fit the available RAM when a CPU is available. Time between tests and simulation spawns are controlled by the keywords listed above. The BulkRun class has three public methods `generate_input_files`, `dry_run` and `start` these will be gone over next.

The generate_input_files Method

The BulkRun class was designed to easily setup and process a large test matrix with no hard limit on the size of the parameter space. For this reason manually creating InputFile instances is not practical and where the `generate_input_files` method comes into play.

The method requries 2 arguments and accepts up to five arguments: `bulk_run.generate_input_files(default_params, default_name_formats, case_identifer='', case_params=None, append=False)`. The first argument, `default_params`, is a dictionary of parameter lists which define the primary parameter space for the bulk run. The second argument, `default_name_formats`, is a dictionary that defines the filename formats. Filename formats are Python format strings used to dynamically generate file names based on the current parameters stored on the InputFile instance. The third argument, `case_identifier`, is also a format string, however it is used to identify a "case" which is a given combination of parameters that you define as significant. A specfic set of parameters can be defined for each case encountered, for example defining your identifier as `{map_type}` and changing the averaging factor for a group of maps based on case parameters. The fourth argument `case_params` is a dictionary of parameter dictionaries where the primary key is the desired case_identifer when formatted with the given parameters. Not all

combinations of an identifier need to be defined, only those you want to specify additional parameters for. The final argument `append` tells the routine whether or not to create a new `input_file_list` or append to the existing list on the bulk_run instance.

Individual InputFile instances have the following parameter resolution order:

1. Hard-coded model defaults
2. Values in the initial input file used to instantiate the class
3. default_params/name_formats passed into generate_input_files
4. case specific parameters defined in the case_params dictionary

The arguments to the method have the following general format:

```
default_params = {
    'param1 keyword': ['p1_value1', 'p1_value2'],
    'param2 keyword': ['p2_value1', 'p2_value2', 'p2_value3'],
    'param3 keyword': ['p3_value1', 'p3_value2', 'p3_value3', 'p3_value4']
}

default_name_formats = {
    'outfile keyword': './path/filename_p1-{param1 keyword}-p2_{param2␣
↪keyword}'
}

my_case_identifier = '{param3 keyword}-{param2 keyword}'

my_case_params = {
    'p3_value1-p2_value2': {'param1 keyword': ['p1_value1', 'p1_value3', 'p1_
↪value3']},
    'p3_value3-p2_value3': {'param4 keyword': ['p4_value1', 'p4_value2'],
↪'param2 keyword': None}
}

bulk_run.generate_input_files(default_params,
                  default_name_formats,
                  case_identifer=my_case_identifier,
                  case_params=my_case_params,
                  append=False)
```

Each parameter range needs to be a list even if it is a single value. Additionally it is recommend that each value already be a string. The value is directly placed into the input file as well in the place of any `{param keyword}` portions of the filename format. Standard Python formatting syntax is used when generating a filename, so non-string arguments may be passed in and will be formatted as defined. Something like `{OUTLET-PRESS:0.4f}` is perfectly valid in the filename formats to handle a floating point number, however no formatting is applied when the value is output to the InputFile object. To disable a parameter defined in the defaults `None` can passed in the case specific parameters for the desired keyword, as shown above with *param2 keyword*. You can manually add InputFile instances to the `bulk_run.input_file_list` before or after running this method, just set the `append` keyword as appropriate. There are no limits to the number of parameters or parameter values to vary but keep in mind every parameter with more than one value increases the total number of simulations multiplicatively. Conflicting parameters will also need to be carefully managed, i.e. varying the boundary conditions through case specific dictionaries. It can safer to comment out all inputs that will be varied and give them a dummy value such as `AUTO` because they will be uncommented when updated with a value. However, ensure the desired and consistent units are being used because they can not be changed from the inital value defined in the InputFile instance.

The dry_run and start Methods

The `dry_run()` method works exactly as its name implies, doing everything except actually starting simulations. It is best if you always run this method before calling the `start()` method to ensure everything checks out. `dry_run` will generate and write out all model input files used allowing you to ensure the input parameters and any name formatting is properly executed. Also, as the code runs it calculates and stores the estimated RAM required for each map. If a map is found to exceed the available RAM an `EnvironmentError/OSError` will be raised halting the program. The BulkRun code does not actually require each input file to have a unique name since the LCL model only references it during initialization. However, if you are overwriting an existing file ensure the spawn_delay is non-zero to avoid creating a race condition or an IO error from simultaneous access. Non-unique output filenames can also cause an IO error in the FORTRAN code if two simulations attempt to access the same file at the same time.

The `start()` method simply begins the simulations. One slight difference from the `dry_run()` method is that input files are only written when a simulation is about to be spawned, instead of writing them all out in the beginning. One additional caveat is that although the BulkRun code takes advantage of the threading and subprocess modules to run simulations asynchronously the BulkRun program itself runs synchronously.

Behind the Scenes

Outside of the public methods used to generate inputs and start a simulation the class does a large portion of the work behind the scenes. Understanding the process can help prevent errors when defining the input ranges. Below is the general flow of the routine after `start()` is called.

1. `_initialize_run()` - processes the aperture maps to estimate requried RAM
2. `_check_processes(processes, RAM_in_use, retest_delay=5, **kwargs)` - Tests to see if any of the simulations have completed
3. `_start_simulations(processes, RAM_in_use, spawn_delay=5, **kwargs)` - Tests to see if additional simulations are able to be started

Running each InputFile

The while loop in `bulk_run.start` operates as long as there is a value left in the `bulk_run.input_file_list`. A non-empty array is treated as a 'True' or 'Truthy' value in Python. The while loop executes two function continuously with a slight delay defined by the user inputs `retest_delay` and `spawn_delay`. The functions it executes are `_check_processes` and `_start_simulations`.

The _check_processes Method

`_check_processes` is a very simple method that essentially pauses the routine until a simulation is completed. It looks through the currently running processes which are stored as an array of `Popen` objects returned by the core method `run_model`. Popen objects are part of the subprocess module in the standard library, they have a method `poll()` which returns `None` if the process has not yet completed. Regardless of the return code when the `poll()` returns a value the corresponding process is removed and its RAM requirement is released before returning from the method. If no processes have completed then the function waits the amount of time specified by `retest_delay` and checks again.

The _start_simulations Method

`_start_simulations` handles the spawning of new processes if certain criteria are met. This method is only entered if `_check_processes` registers that a simulation has completed. It first calculates the amount of free RAM based on the maximum requirement of currently running simulations. Then it enters a while loop to test spawn criteria, if either fail the method returns and the while loop tests its own exit criteria or calls `_check_processes` otherwise. Return conditions are if the number of current processes is greater than or equal to the number of CPUs or if all maps require more RAM than available.

If both criteria are satisfied then a new process is spawned and its RAM requirement and the process are stored. The method then waits for the duration specified by the `spawn_delay` argument and checks to see if it can spawn any additional processes by retesting the same exit criteria defined above. This method and the one above work in conjunction to process all of the InputFiles stored in `bulk_run.input_file_list`.

Using the BlockMeshDict class

Contents

Description

This describes the basic process of how to convert a vertical aperture map into an OpenFoam readable blockMeshDict. Using the BlockMeshDict class is a very simple and all the heavy lifting is done internally unless additional customizations are needed. Additional customizations can include setting up your own edges and mergePatchPairs as well as apply more than just the standard boundary face labels. The routine can also create sub directories OpenFoam expects the mesh file to be in. The template at the bottom of the file can be used to base a simple mesh generation off of by copying and pasting it into a file and running it as a python script. There is additional code commented out below that allows you to generate a thresholded mesh. **Currently there is not a good method to add custom face labels that is in the works.**

There are three steps required to generate a mesh file:

1. Create a DataField object to store the aperture map data
2. Create the BlockMeshDict class
3. Write the mesh to your desired location

Setting up the BlockMeshDict

As mentioned the first step is creating a data field object, the only required argument is the path to the aperture map file. However first we need to import the modules.

```
from apmapflow import DataField
from apmapflow.openfoam import BlockMeshDict
```

Next we can instantiate the data field class.

```
aper_map_file = r'./examples/Fractures/Fracture1ApertureMap-10avg.txt'
aper_map_field = DataField(aper_map_file)
```

With the DataField for the aperture map file created we can now instantiate the BlockMeshDict class. The class accepts three arguments, `BlockMeshDict(field, avg_fact=1.0, mesh_params=None)`. The first argument `field` is a `DataField` object, in this case `aper_map_field` will go there. `avg_fact` is the horizontal averaging or scale factor of the map. It defaults to assume that each cell in the map has a 1 voxel by 1 voxel square base. The final argument `mesh_params` is a dictionary used to populate several aspects of the blockMeshDict file. Below I have listed the default params for the mesh class, these will be overwritten by anything you define.

```
default_mesh_params = {
    'convertToMeters': '1.0',
    'numbersOfCells': '(1 1 1)',
    'cellExpansionRatios': 'simpleGrading (1 1 1)',
    #
    'boundary.left.type': 'wall',
    'boundary.right.type': 'wall',
    'boundary.top.type': 'wall',
    'boundary.bottom.type': 'wall',
    'boundary.front.type': 'wall',
    'boundary.back.type': 'wall'
}
```

`convertToMeters` is where you specify the voxel to meters conversion factor. The `boundary.*.type` entries set the types for each defined face label. Any additional face labels you create would need their type specified here. Next we will create the mesh class for our aperture map defining the parameters required to replicate a 2-D LCL simulation.

```
my_mesh_params = {
    'convertToMeters': '2.680E-5',
    #
    'boundary.left.type': 'empty',
    'boundary.right.type': 'empty',
    'boundary.top.type': 'wall',
    'boundary.bottom.type': 'wall',
    'boundary.front.type': 'wall',
    'boundary.back.type': 'wall'
}

mesh = BlockMeshDict(aper_map_field, avg_fact=10.0, mesh_params=my_mesh_
↪params)
```

The mesh stores the verticies, blocks, faces, edges and mergePatchPairs in scipy ndarrays as attributes. They are accessible by typing `mesh._verticies` or `mesh._edges`, etc. The `._edges` and `._merge_patch_pairs` arrays are not initialized by default. Face labels are stored in a dictionary attribute named `face_labels` each key has the format boundary.side for example `face_labels['boundary.bottom']` would return a boolean array and all indicies that are `True` correspond to a 'bottom' face. If you need to add custom edges or mergePatchPairs then a valid list of strings representing them will need to be stored in the `mesh._edges` and `mesh._merge_patch_pairs` arrays. The mesh does no additional processing on them so what you put

is is exactly what will be output in those sections of the blockMeshDict file. For example to add in arc shaped edges you would need to store strings like this `'arc 1 5 (1.1 0.0 0.5)'` in the `._edges` array. Each entry in the `._edges` array should describe a single edge.

Creating the blockMeshdict File

All of the work mainly takes place in the setup steps and the user just needs to call `mesh.write_foam_file()` to use the defaults and output a mesh file in the local directory. The output function also takes three optional parameters as well, `mesh.write_foam_file(path='.', create_dirs=True, overwrite=False)`. The first allows for an alternate output location, say in the 'run' folder of OpenFoam, relative and absolute paths are valid. *create_dirs* tells the export whether or not to create the `constants/polyMesh` directories for you, if this is true and they already exist the file will be output in that location preserving the contents of those directories. The final parameter *overwrite* prevents or enables the program to replace an existing blockMeshDict file in the chosen location.

Template Code for Simple Mesh Generation

The template below can be used with some minor customization for simple mesh generation. The commented out section below allows generation of a 'thresholded' mesh where all data values less/greater than or equal to the min_value and max_value are removed. When cells are removed internal faces are exposed and assigned an 'internal' patch name which defaults to the 'wall' BC type.

```
import os
from apmapflow import DataField
from apmapflow.openfoam import BlockMeshDict
#
# The path to the aperture map needs to be updated to match the file you␣
↪want to export
aper_map_file = os.path.join('path', 'to', 'aperture_map_file.txt')
aper_map_field = DataField(aper_map_file)
#
# convertToMeters needs to be updated to match your data
# numbersOfCells needs to be updated to match your desired internal block␣
↪meshing
my_mesh_params = {
    'convertToMeters': '1.0',
    'numbersOfCells': '(1 1 1)',
    'cellExpansionRatios': 'simpleGrading (1 1 1)',
    #
    'boundary.left.type': 'wall',
    'boundary.right.type': 'wall',
    'boundary.top.type': 'wall',
    'boundary.bottom.type': 'wall',
    'boundary.front.type': 'wall',
    'boundary.back.type': 'wall'
}
#
mesh = BlockMeshDict(aper_map_field, avg_fact=1.0, mesh_params=my_mesh_
↪params)
mesh.write_foam_file(path='.', create_dirs=True, overwrite=False)
#
#
# the code below generates a thresholded mesh
#
```

```
# mesh.generate_threshold_mesh(min_value=0.0, max_value=1.0e9)
# mesh.write_foam_file(path='.', create_dirs=True, overwrite=False)
```

Using the openfoam Module

Contents

Description

The openfoam Module is designed to allow easy modification and creation of OpenFoam files in a Python interface. The main motivation for writing the code was to create a work flow that would allow results from the Local Cubic Law (LCL) model to be quickly compared to OpenFoam results on the same geometry. The module has routines to load and parse basic OpenFoam files as well as generate a blockMeshDict file from a 2-D aperture map. There are three primary classes and three additional helper classes used in the module, all of which will be gone over next. The apm_open_foam_export script wraps all of the functionality present in the module into a program that can generate a complete OpenFoam simulation by modifying an existing set of OpenFoam files.

OpenFoamFile Class

The OpenFoamFile class is one of the central objects of the openfoam module allowing the reading of an existing OpenFoamFile or creating one from scratch. It inherits from the OpenFoamObject and the OrderedDict classes. All OpenFoam files have a dictionary-like structure and as such data is stored on the OpenFoamFile object in the same format as a regular Python dictionary. It has one method, `write_foam_file(*args, **kwargs)`. The class can be instantiated by directly providing values or giving a filename to read instead. Direct creation of an OpenFoamFile instance has two required positional arguments and 2 optional keyword arguments. `OpenFoamFile(location, object_name, class_name=None, values=None)`. The first three correspond to entries in the FoamFile dictionary at the top of all OpenFoam files and the final argument is a set of key, value pairs to load onto the object. Because the class inherits from the OrderedDict class any valid dictionary iterable in Python can be used however a list of key,value pairs works best because order is maintained.

```
# loading modules
import os
from apmapflow import openfoam as of

# directly creating an OpenFoamFile object
init_vals = [
    ('numberOfSubdomains', '2'),
    ('method', 'scotch'),
    ('distributed', 'no')
]
of_file = of.OpenFoamFile('system', # goes in the system directory
                          'decomposeParDict', # OpenFoam object name
                          class_name='dictionary, # showing default value
                          values=init_vals)

# checking value and resetting
print(of_file['numberOfSubdomains']) # prints '2'
of_file['numberOfSubdomains'] = '4'

# creating instance of OpenFoamFile by reading an existing file
filename = 'path/to/OpenFoamFile'
of_file = of.OpenFoamFile(filename)
```

Once an instance has been created and contains the desired values a file can be easily written to a specified location by calling `write_foam_file(path='.', create_dirs=True, overwrite=False)` instance method. By default a file is written to the following path and name './location/object' where location and object are values stored in the `head_dict` attribute of the class object. In the above example location is the 'system' directory and object is 'decomposeParDict'. An alternative output name, for example 'decomposeParDict-np4' can be defined by setting the `name` attribute of the object. The example below will show a few examples of writing files.

When using an instance produced from an existing file the location and name attribute can differ from the defaults. The code will attempt to pull the 'location' value from the FoamFile dict in the file being read, if that fails then it will use the name of the directory the file was stored in. The initial value of the `name` attribute is always the name of the file being read. This was done to allow different versions of the same file to coexist when creating an export.

```
# writing file to './system/decomposeParDict'
of_file.write_foam_file()

# writing file to a FOAM_RUN case directory (may not work correctly in␣
↪Spyder)
foam_run = os.getenv('FOAM_RUN')
```

```
case_dir = os.path.join(foam_run, 'LCL-test')
of_file.write_foam_file(path=case_dir)

# writing file to './decomposeParDict'
of_file.write_foam_file(create_dirs=False)

# writing file to './system/decomposeParDict-np4'
of_file.name = 'decomposeParDict-np4'
of_file.write_foam_file()
```

OpenFoamObject Class

This class is not intended for direct use and has no methods of its own. It is used to identify any objects descended from it because they have specialized `__str__` methods that need called directly. Any future methods that need applied to the entire gamut of OpenFoam objects will also be added here.

OpenFoamDict Class

Along with the OpenFoamList class this is a primary building block of an OpenFoam file. It is descended from the OpenFoamObject and OrderedDict classes. The primary feature of the class is a specialized `__str__` method that produces a nicely formatted dictionary structure in an OpenFoam file with proper indentation. Instantiation is done in the same way as a regular dict with one exception, the first argument is the 'name' to be used in output and is required. The second argument is optional and can be any valid iterable used to initialize a regular dictionary. Any number of OpenFoamDicts and OpenFoamLists can be mixed and nested into each other.

OpenFoamList Class

This is the second core building block used in OpenFoam files and mainly in blockMeshDict generation. It is descended from the OpenFoamObject and list classes. This class also has a specialized `__str__` method that produces an output considerably different than calling `str()` on a regular Python list and honors indentation from nesting. Instantiation is similar to the OpenFoamDict class where the first parameter is the required named attribute of the class and the second is optional but can be any valid iterable used to initialize a regular list. As above any number of OpenFoamDicts and OpenFoamLists can be mixed and nested into each other.

BlockMeshDict Class

The BlockMeshDict class is used to generate a mesh file from a provided 2-D aperture map data field. It descends from the OpenFoamFile class however has significantly different usage and the only method it shares from the parent class is `write_foam_file`. If `create_dirs=true` then it will automatically generate the 'constant/polyMesh' sub-directories on the desired path. Full explanation of mesh generation is beyond the scope of this example and is covered in depth in the blockMeshDict example

OpenFoamExport Class

The OpenFoamExport class is designed to act as a central location to manage and write OpenFoam files. This class has a few public methods that allow it to interact with OpenFoam objects. It can be initialized either with no arguments or supplying the optional arguments to pass along to the BlockMeshDict

class. The latter case generates and stores a BlockMeshDict class instance on the `block_mesh_dict` attribute other wise that attribute is 'None'. The second primary attribute of the class is a foam_files dictionary where each key-value pair represents an OpenFoamFile instance. Each key has the format of 'location.name' where location is the value of the 'location' key in the file's head_dict and name is the file's 'name' attribute.

BlockMesh Related Methods

As stated above this class is able to call some methods of its internally stored BlockMeshDict instance. An instance of the BlockMeshDict is created by calling the `generate_block_mesh_dict(field, avg_fact=1.0, mesh_params=None)` method. The arguments are exactly the same as and passed on to the BlockMeshDict constructor. These arguments can also be supplied during instantiation of the OpenFoamExport class and a BlockMeshDict instance will automatically be created. The other two methods available are `write_symmetry_plane(path='.', create_dirs=True, overwrite=False)` and `write_mesh_file(path='.', create_dirs=True, overwrite=False)`. Any other BlockMeshDict methods such as `generate_threshold_mesh` need to be called directly from the `block_mesh_dict` attribute. Additionally the `write_foam_files` method of the OpenFoamExport call will not write a mesh file, one of the above methods will need to be called to output one.

OpenFoamFile Related Methods

There are two methods associated with OpenFoamFile instances, they are `generate_foam_files(*args)` and `write_foam_files(path='.', overwrite=False)`. The first method accepts any number of arguments where each argument is one of the following three forms.

1. An existing instance of the OpenFoamFile class
2. A filename to read from
3. A valid dictionary iterable with at minimum a 'location' and 'object' key.

The two required keys and an optional 'class_name' key are passed onto the OpenFoamFile constructor. Any remaining values are sent along in the 'values' keyword argument. All OpenFoamFile instances are stored on the 'foam_files' attribute of the OpenFoamExport class. That attribute is a python dictionary and the format of keywords are 'location.name' where location is the 'location' key in the stored file's head dict and 'name' is the value of the 'name' attribute of the stored file.

The final method is `write_foam_files(path='.', overwrite=False)` which writes every file stored in the foam_files dict to `path/location/name`. The blockMeshDict is not generated using this method, one of the methods listed in the above section needs to be used as well.

The apm_open_foam_export Script

Usage

Located in the scripts directory, it is designed to help automate the process of taking input files from the LCL model and creating a full OpenFoam simulation case to run. The script is meant to be run from the command line and accepts several arguments and flags. The script only modifies or creates four files including the blockMesh, all other files will either need to be created or exist in the directory read from.

Arguments and Flags

The export command line utility accepts several flags and a useful help message can be generated by running the command `apm_open_foam_export --help`. The command needs to be run from the scripts folder unless a valid path to the script is used or you make the scripts folder visible on the `$PATH` environment variable. Basic syntax of the command is:

```
apm_open_foam_export [-ivf] [-r READ_DIR] [-o OUTPUT_DIR] [input_file]
```

All arguments are optional, the default values for READ_DIR and OUTPUT_DIR is the current working directory. When reading from a directory the command will not recursively search all subdirectories, only the constant, system and 0 directories will be searched if they exist. Sub directories of those directories are also not searched, for example nothing in the constant/polyMesh directory will be found unless that is explicitly stated as the directory to read from. Note, blockMeshDict files are not directly excluded but should not be intentionally read due to their length and likelihood of parsing errors due to different formatting.

Flag and Argument description

- -i, –interactive : interactive mode, explained below
- -v, –verbose : verbose logging messages
- -f, –force : overwrite mode, allows any existing files to be replaced
- -r [dir], –read-dir [dir] : specifies a directory to read files from
- -o [dir], –output-dir [dir] : specifies a directory to output the files to.
- input_file : LCL Model input file to pull information from

Automatic BlockMesh Generation

By suppling a LCL input_file in the command's args an instance of the BlockMeshDict class is created and automatically linked to the internal export variable. Several pieces of information are pulled from the input_file. However, the five of note for blockMeshDict generation are APER-MAP, AVG_FACT, VOXEL, ROUGHNESS, HIGH-MASK and LOW-MASK. The latter three are applied as geometric adjustments to the aperture map data read from the APER-MAP keyword. The value of AVG_FACT is passed on to the BlockMeshDict constructor as the avg_fact argument, voxel-size and the BC's determined are stored in the mesh_params dictionary. Also, a default cell-grading of (5 5 5) is used.

Automatic File Generation

The three files below are generated by the script or modified from existing files found and read. A U file is only created/modified if an aperture map is found at the path specified by the APER-MAP keyword in the input file.

p File

The script will check the foam_files dict for a '0.p' key to update and if one is not found then it will create a new OpenFoamFile instance for that key. The only dictionary updated in the p file is the boundaryField dict. It will attempt to pull patch names from the blockMeshDict object but if it does not exist then the standard 6 faces are used: left, right, bottom, top, front and back. Initially all patches have a 'zeroGradient' boundary condition defined. If any pressure BC's were found in the LCL model input file then a kinematic pressure condition is created for that BC.

U File

The U file is generated in much the same way as the p file, if there is not a value for the '0.U' key in the foam_files dictionary one is created. All patches are initially defined as no-slip walls. If a pressure BC was defined for a side then that side is changed from a no-slip wall to the zeroGradient BC type. If a fixed rate condition was defined for the LCL model then average cross-sectional area of the BC side is used to calculate a uniform U field value from the volumetric flow.

transportProperties File

The transport properties file only has two keys updated 'nu' and 'rho' with the values defined in the input file. Any additional coefficients defined will need to be manually updated. If a viscosity or density is not supplied by the LCL model input file then the standard values for water are used 1.0 cP and 1000 kg/m^3. Only a file stored on the 'constant.transportProperties' key of the foam_files dictionary is modified or created if it doesn't exist.

Interactive Mode

Interactive mode is activated by using the -i flag. When interactive mode is used the script recalls itself using `python3 -i (script and args passed)` which essentially causes the script to be executed in an interactive python interpreter. Anything that exists on the main namespace in the script is visible and defined in the interactive session such as variables, functions, module imports, etc. Using the command `apm_open_foam_export -iv` will begin an interactive mode session.

Files are not automatically written in interactive mode. To write all files based on the command line args used call the function `write_all_files(overwrite=False)`. If the -f flag was used `overwrite=False` is ignored, alternatively `overwrite=True` can be used to mimic the effects of the -f flag. This command will write the blockMeshDict file as well.

Several global variables are defined to ease the use of interactive mode. For a more complete understanding of what is available in terms of functions and globals it recommended to review the code in apm_open_foam_export.py.

Unit Conversion

Contents

Overview

Unit conversions are facilitated with three functions, `register_voxel_unit`, `convert_value`, `get_conversion_factor`. All of the functionality is build on top of pint with the functions act-

ing as a basic layer of abstraction. Additionally the 'micron' unit which is equivalent to a micrometer is automatically defined. Access to the pint `UnitRegistry` instance is available through the `unit_conversion.unit_registry` attribute.

register_voxel_unit

`register_voxel_unit(length, unit)` is used as expected to define the 'voxel' unit in the registry because it is not a standard unit of measurement. Once registered conversions can be applied as usual. The unit describing the length must be a standard unit of measurement such as millimeters, inches, microns, etc.

convert_value

`convert_value(value, unit_in, unit_out='SI')` converts a value in the current unit to the unit specified by unit_out. If unit_out is not provided then the value is converted to the proper SI units.

get_conversion_factor

`get_conversion_factor(unit_in, unit_out='SI')` returns a conversion factor from the current unit to the unit specified by unit_out. If unit_out is not provided then a conversion factor to the SI unit is returned.

Module Reference

Data Processing

Data procesing classes for 2-D data maps.

Written By: Matthew Stadelman
Date Written: 2016/02/26
Last Modifed: 2016/02/26

Base Processor

This is a template that is used to add additional processors to
the package.

Written By: Matthew Stadelman
Date Written: 2016/02/26
Last Modifed: 2016/10/25

class apmapflow.data_processing.base_processor.**BaseProcessor**(*field*)
Only required parameter is a data field object, initializes properties defined by subclassses.

__delattr__
Implement delattr(self, name).

__dir__() → list
default dir() implementation

__eq__
Return self==value.

__format__()
default object formatter

__ge__
Return self>=value.

__getattribute__
Return getattr(self, name).

__gt__
Return self>value.

__hash__
Return hash(self).

__le__
Return self<=value.

__lt__
Return self<value.

__ne__
Return self!=value.

__new__()
Create and return a new object. See help(type) for accurate signature.

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__
Return repr(self).

__setattr__
Implement setattr(self, name, value).

__sizeof__() → int
size of object in memory, in bytes

__str__
Return str(self).

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

classmethod _add_subparser(*subparser*)
Adds a specific action based sub-parser to the supplied arg_parser instance.

_output_data(*filename=None*, *delim=', '*)
Not implemented

_process_data(***kwargs*)
Not implemented

copy_processed_data(*data_dict*, *alt_key=False*)
Copys the current processed data array to a dict object using a key defined in the subclass initialization or a key supplied by the alt_key keyword.

gen_output(***kwargs*)
Calls the subclassed routine output_data to create outfile content

print_data()
Writes the data processor's data the screen

process(***kwargs*)
Calls the subclassed routine process_data to create outfile content

setup(***kwargs*)
Sets or resets arguments

write_data(*path='/home/docs/checkouts/readthedocs.org/user_builds/netl-ap-map-flow/checkouts/latest/docs'*)
Writes the data processor's data to its outfile

Percentile

Calculates a set of percentiles for a dataset

Written By: Matthew Stadelman
Date Written: 2016/02/26
Last Modifed: 2016/10/25

class apmapflow.data_processing.percentiles.**Percentiles**(*field*, ***kwargs*)
Automatic method to calculate and output a list of data percentiles. kwargs include:

percentiles : list of percentiles to calculate (required) key_format : format to write percentile dictionary keys in (optional) value_format : format to write percentile values in (optional)

__delattr__
Implement delattr(self, name).

__dir__() → list
default dir() implementation

__eq__
Return self==value.

__format__()
default object formatter

__ge__
Return self>=value.

__getattribute__
Return getattr(self, name).

__gt__
Return self>value.

__hash__
Return hash(self).

__le__
Return self<=value.

__lt__
Return self<value.

__ne__
Return self!=value.

__new__()
Create and return a new object. See help(type) for accurate signature.

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__
Return repr(self).

__setattr__
Implement setattr(self, name, value).

__sizeof__() → int
size of object in memory, in bytes

__str__
Return str(self).

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

classmethod **_add_subparser**(*subparsers*, *parent*)
Adds a specific action based sub-parser to the supplied arg_parser instance.

_output_data(*filename=None*, *delim=', '*)
Creates the output content for percentiles

_process_data()
Takes a list of percentiles specified in self.args and generates the corresponding set of values.

copy_processed_data(*data_dict*, *alt_key=False*)
Copys the current processed data array to a dict object using a key defined in the subclass initialization or a key supplied by the alt_key keyword.

gen_output(***kwargs*)
Calls the subclassed routine output_data to create outfile content

print_data()
Writes the data processor's data the screen

process(***kwargs*)
Calls the subclassed routine process_data to create outfile content

setup(***kwargs*)
Sets or resets arguments

write_data(*path='/home/docs/checkouts/readthedocs.org/user_builds/netl-ap-map-flow/checkouts/latest/docs'*)
Writes the data processor's data to its outfile

Evaluate Channelization

Evaluates channelization in flow data based on the number and widths of channels

Written By: Matthew Stadelman
Date Written: 2016/02/26
Last Modifed: 2016/10/25

class apmapflow.data_processing.eval_channels.**EvalChannels**(*field*, ***kwargs*)
Evaluates channelization in flow data based on the number and widths of channels. More work needs to be done on this class to make it not dependent on a specfic direction. kwargs include:

thresh - minimum numeric value considered to be part of a flow channel axis - (x or z) specifies which axis to export along

__delattr__
Implement delattr(self, name).

__dir__() → list
default dir() implementation

__eq__
Return self==value.

__format__()
default object formatter

__ge__
Return self>=value.

__getattribute__
Return getattr(self, name).

__gt__
Return self>value.

__hash__
Return hash(self).

__le__
Return self<=value.

__lt__
Return self<value.

__ne__
Return self!=value.

__new__()
Create and return a new object. See help(type) for accurate signature.

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__
Return repr(self).

__setattr__
Implement setattr(self, name, value).

__sizeof__() → int
size of object in memory, in bytes

__str__
Return str(self).

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

classmethod **_add_subparser**(*subparsers*, *parent*)
Adds a specific action based sub-parser to the supplied arg_parser instance.

_output_data(*filename=None*, *delim=','*, ***kwargs*)
creates the output content for channelization

_process_data(***kwargs*)
Examines the dataset along one axis to determine the number and width of channels.

copy_processed_data(*data_dict*, *alt_key=False*)
Copys the current processed data array to a dict object using a key defined in the subclass initialization or a key supplied by the alt_key keyword.

gen_output(***kwargs*)
Calls the subclassed routine output_data to create outfile content

print_data()
Writes the data processor's data the screen

process(***kwargs*)
Calls the subclassed routine process_data to create outfile content

setup(***kwargs*)
Sets or resets arguments

write_data(*path='/home/docs/checkouts/readthedocs.org/user_builds/netl-ap-map-flow/checkouts/latest/docs'*)
Writes the data processor's data to its outfile

Histogram

Calculates a simple histogram for a data map. This also serves as the
super class for variants of a simple histogram.

Written By: Matthew Stadelman
Date Written: 2016/02/29
Last Modifed: 2016/10/25

class apmapflow.data_processing.histogram.**Histogram**(*field*, ***kwargs*)
Performs a basic histogram of the data based on the number of bins desired. The first bin contains all values below the 1st percentile and the last bin contains all values above the 99th percentile to keep axis scales from being bloated by extrema. kwargs include:

num_bins - integer value for the total number of bins

__delattr__
Implement delattr(self, name).

__dir__() → list
default dir() implementation

__eq__
Return self==value.

__format__()
default object formatter

__ge__
Return self>=value.

__getattribute__
Return getattr(self, name).

__gt__
Return self>value.

__hash__
Return hash(self).

__le__
Return self<=value.

__lt__
Return self<value.

__ne__
Return self!=value.

__new__()
Create and return a new object. See help(type) for accurate signature.

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__
Return repr(self).

__setattr__
Implement setattr(self, name, value).

__sizeof__() → int
size of object in memory, in bytes

__str__
Return str(self).

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

classmethod **_add_subparser**(*subparsers*, *parent*)
Adds a specific action based sub-parser to the supplied arg_parser instance.

_output_data(*filename=None*, *delim=', '*)
Creates the output content for histograms

_process_data(*preserve_bins=False*)
Calculates a histogram from a range of data. This uses the 1st and 99th percentiles as limits when defining bins

copy_processed_data(*data_dict*, *alt_key=False*)
Copys the current processed data array to a dict object using a key defined in the subclass initialization or a key supplied by the alt_key keyword.

define_bins()
This defines the bins for a regular histogram

gen_output(***kwargs*)
Calls the subclassed routine output_data to create outfile content

print_data()
Writes the data processor's data the screen

process(***kwargs*)
Calls the subclassed routine process_data to create outfile content

setup(***kwargs*)
Sets or resets arguments

write_data(*path='/home/docs/checkouts/readthedocs.org/user_builds/netl-ap-map-flow/checkouts/latest/docs'*)
Writes the data processor's data to its outfile

Range Histogram

Calculates a histogram over a defined percentile range for a data map.

Written By: Matthew Stadelman
Date Written: 2016/03/07
Last Modifed: 2016/10/25

class apmapflow.data_processing.histogram_range.**HistogramRange**(*field*, ***kwargs*)

Performs a histogram where the minimum and maximum bin limits are set by percentiles and all values outside of that range are excluded. The interior values are handled the same as a basic histogram with bin sizes evenly spaced between the min and max percentiles given.

kwargs include: num_bins - integer value for the total number of bins range - list with two numeric values to define the minimum and maximum

data percentiles.

__delattr__
Implement delattr(self, name).

__dir__() → list
default dir() implementation

__eq__
Return self==value.

__format__()
default object formatter

__ge__
Return self>=value.

__getattribute__
Return getattr(self, name).

__gt__
Return self>value.

__hash__
Return hash(self).

__le__
Return self<=value.

__lt__
Return self<value.

__ne__
Return self!=value.

__new__()
Create and return a new object. See help(type) for accurate signature.

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__
Return repr(self).

__setattr__
Implement setattr(self, name, value).

__sizeof__() → int
size of object in memory, in bytes

__str__
Return str(self).

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

classmethod _add_subparser(*subparsers*, *parent*)
Adds a specific action based sub-parser to the supplied arg_parser instance.

_output_data(*filename=None*, *delim=', '*)
Creates the output content for histograms

_process_data(*preserve_bins=False*)
Calculates a histogram from a range of data. This uses the 1st and 99th percentiles as limits when defining bins

copy_processed_data(*data_dict*, *alt_key=False*)
Copys the current processed data array to a dict object using a key defined in the subclass initialization or a key supplied by the alt_key keyword.

define_bins(***kwargs*)
This defines the bins for a range histogram

gen_output(***kwargs*)
Calls the subclassed routine output_data to create outfile content

print_data()
Writes the data processor's data the screen

process(***kwargs*)
Calls the subclassed routine process_data to create outfile content

setup(***kwargs*)
Sets or resets arguments

write_data(*path='/home/docs/checkouts/readthedocs.org/user_builds/netl-ap-map-flow/checkouts/latest/docs'*)
Writes the data processor's data to its outfile

Logscaled Histogram

Calculates a logarithmically spaced histogram for a data map.

Written By: Matthew Stadelman
Date Written: 2016/03/07
Last Modifed: 2016/10/20

class apmapflow.data_processing.histogram_logscale.**HistogramLogscale**(*field*, ***kwargs*)

Performs a histogram where the bin limits are logarithmically spaced based on the supplied scale factor. If there are negative values then the first bin contains everything below 0, the next bin will contain everything between 0 and 1. kwargs include:

scale_fact - numeric value to generate axis scale for bins. A scale fact of 10 creates bins: 0-1, 1-10, 10-100, etc.

__delattr__
Implement delattr(self, name).

__dir__() → list
default dir() implementation

__eq__
Return self==value.

__format__()
default object formatter

__ge__
Return self>=value.

__getattribute__
Return getattr(self, name).

__gt__
Return self>value.

__hash__
Return hash(self).

__le__
Return self<=value.

__lt__
Return self<value.

__ne__
Return self!=value.

__new__()
Create and return a new object. See help(type) for accurate signature.

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__
Return repr(self).

__setattr__
Implement setattr(self, name, value).

__sizeof__() → int
size of object in memory, in bytes

__str__
Return str(self).

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

classmethod **_add_subparser**(*subparsers*, *parent*)
Adds a specific action based sub-parser to the supplied arg_parser instance.

_output_data(*filename=None*, *delim=', '*)
Creates the output content for histograms

_process_data(*preserve_bins=False*)
Calculates a histogram from a range of data. This uses the 1st and 99th percentiles as limits when defining bins

copy_processed_data(*data_dict*, *alt_key=False*)
Copys the current processed data array to a dict object using a key defined in the subclass initialization or a key supplied by the alt_key keyword.

define_bins(***kwargs*)
This defines the bins for a logscaled histogram

gen_output(***kwargs*)
Calls the subclassed routine output_data to create outfile content

print_data()
Writes the data processor's data the screen

process(***kwargs*)
Calls the subclassed routine process_data to create outfile content

setup(***kwargs*)
Sets or resets arguments

write_data(*path='/home/docs/checkouts/readthedocs.org/user_builds/netl-ap-map-flow/checkouts/latest/docs'*)
Writes the data processor's data to its outfile

Profile

Outputs a set of data profiles along either the X or Z axis

Written By: Matthew Stadelman
Date Written: 2016/03/07
Last Modifed: 2016/10/25

class apmapflow.data_processing.profile.**Profile**(*field*, ***kwargs*)
Automatic method to export data vectors for a range of distances along a specified axis. Locations are given as percentages from the bottom or left edge of the 2-D data map

locations - list of numbers, ex: [10, 25, 50, 75, 90] axis - (x or z) specifies which axis to export along

__delattr__
Implement delattr(self, name).

__dir__() → list
default dir() implementation

__eq__
Return self==value.

__format__()
default object formatter

__ge__
Return self>=value.

__getattribute__
Return getattr(self, name).

__gt__
Return self>value.

__hash__
Return hash(self).

__le__
Return self<=value.

__lt__
Return self<value.

__ne__
Return self!=value.

__new__()
Create and return a new object. See help(type) for accurate signature.

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__
Return repr(self).

__setattr__
Implement setattr(self, name, value).

__sizeof__() → int
size of object in memory, in bytes

__str__
Return str(self).

__subclasshook__()
Abstract classes can override this to customize issubclass().

This is invoked early on by abc.ABCMeta.__subclasscheck__(). It should return True, False or NotImplemented. If it returns NotImplemented, the normal algorithm is used. Otherwise, it overrides the normal algorithm (and the outcome is cached).

__weakref__
list of weak references to the object (if defined)

classmethod **_add_subparser**(*subparsers*, *parent*)
Adds a specific action based sub-parser to the supplied arg_parser instance.

_output_data(*filename=None*, *delim=', '*)
Creates the output content for data profiles

_process_data(***kwargs*)
Takes a list of percentiles specified in **kwargs and generates the corresponding set of values.

copy_processed_data(*data_dict*, *alt_key=False*)
Copys the current processed data array to a dict object using a key defined in the subclass initialization or a key supplied by the alt_key keyword.

gen_output(***kwargs*)
Calls the subclassed routine output_data to create outfile content

print_data()
Writes the data processor's data the screen

process(***kwargs*)
Calls the subclassed routine process_data to create outfile content

setup(***kwargs*)
Sets or resets arguments

write_data(*path='/home/docs/checkouts/readthedocs.org/user_builds/netl-ap-map-flow/checkouts/latest/docs'*)
Writes the data processor's data to its outfile

class apmapflow.data_processing.**EvalChannels**(*field*, ***kwargs*)
Evaluates channelization in flow data based on the number and widths of channels. More work needs to be done on this class to make it not dependent on a specfic direction. kwargs include:

thresh - minimum numeric value considered to be part of a flow channel axis - (x or z) specifies which axis to export along

class apmapflow.data_processing.**Histogram**(*field*, ***kwargs*)
Performs a basic histogram of the data based on the number of bins desired. The first bin contains all values below the 1st percentile and the last bin contains all values above the 99th percentile to keep axis scales from being bloated by extrema. kwargs include:

num_bins - integer value for the total number of bins

define_bins()
This defines the bins for a regular histogram

class apmapflow.data_processing.**HistogramLogscale**(*field*, ***kwargs*)
Performs a histogram where the bin limits are logarithmically spaced based on the supplied scale factor. If there are negative values then the first bin contains everything below 0, the next bin will contain everything between 0 and 1. kwargs include:

scale_fact - numeric value to generate axis scale for bins. A scale fact of 10 creates bins: 0-1, 1-10, 10-100, etc.

define_bins(***kwargs*)
This defines the bins for a logscaled histogram

class apmapflow.data_processing.**HistogramRange**(*field*, ***kwargs*)
Performs a histogram where the minimum and maximum bin limits are set by percentiles and all values outside of that range are excluded. The interior values are handled the same as a basic histogram with bin sizes evenly spaced between the min and max percentiles given.

kwargs include: num_bins - integer value for the total number of bins range - list with two numeric values to define the minimum and maximum

data percentiles.

define_bins (***kwargs*)
This defines the bins for a range histogram

class apmapflow.data_processing.**Percentiles** (*field*, ***kwargs*)
Automatic method to calculate and output a list of data percentiles. kwargs include:

percentiles : list of percentiles to calculate (required) key_format : format to write percentile dictionary keys in (optional) value_format : format to write percentile values in (optional)

class apmapflow.data_processing.**Profile** (*field*, ***kwargs*)
Automatic method to export data vectors for a range of distances along a specified axis. Locations are given as percentages from the bottom or left edge of the 2-D data map

locations - list of numbers, ex: [10, 25, 50, 75, 90] axis - (x or z) specifies which axis to export along

OpenFoam

Module storing the public interface to generate OpenFoam cases from LCL simulations and aperture maps.

Written By: Matthew Stadelman
Date Written: 2016/03/22
Last Modifed: 2016/08/08

Block Mesh Dict

A class build to handle generation of a blockMeshDict from an aperture map.

Written By: Matthew Stadelman
Date Written: 2016/08/08
Last Modifed: 2016/08/08

class apmapflow.openfoam.block_mesh_dict.**BlockMeshDict** (*field*, *avg_fact=1.0*, *mesh_params=None*, *offset_field=None*)
This is a special subclass of OpenFoamFile used to generate and output a blockMeshDict for OpenFoam

__init__ (*field*, *avg_fact=1.0*, *mesh_params=None*, *offset_field=None*)
Takes a field object and a set of mesh params to set the properties of the blockMeshDict

field : DataField, field that is storing the aperture map data avg_fact : float, optional, number of voxels along the X and Z axes. mesh_params : dict, optional, a dictionary containing parameters to use instead of the defaults offset_field : DataField, field object that stores the offset data for an aperture map

_create_blocks(*cell_mask*)
Sets up the vertices and blocks.
•cell_mask is a boolean array in the shape of the data_map
telling the function what blocks to include.

vert_map stores the 4 vertex indices that make up the back surface and the front surface is '+ 1' the index of the corresponding lower point. In terms of the very first block vert_map[i,j,0] is at the orgin (0,0), bottom left corner vert_map[i,j,1] is bottom right corner (1,0) vert_map[i,j,2] is the top right corner (1,1) vert_map[i,j,3] is the top left corner (0,1)

_generate_masked_mesh(*cell_mask=None*)
Generates the mesh based on the cell mask provided

generate_mesh_file()
Populates keys on itself based on geometry data to output a mesh file

generate_simple_mesh()
Generates a simple mesh including all cells in the data map

generate_threshold_mesh(*min_value=0.0*, *max_value=1000000000.0*)
Generates a mesh excluding all blocks below the min_value arg. Regions that are isolated by the thresholding are also automatically removed.

set_boundary_patches(*boundary_blocks*, *reset=False*)
Sets up boundary patches based on the dictionary passed in. Overlapping declarations are overwritten by the last patch to use that face. The boundary blocks dictionary contains a dictionary entry for each patch name.
•**boundary_blocks dictionary has the format of:**
{patch_name: {
<side>: [block-list], <side>: [block-list], ...
} where <side> is left, right, bottom, top, front or back and block list is a list of blocks to add that patch to the side of.
•**reset - boolean** [if True then the face labels dictionary] and _faces array are re-initialized to default values

write_foam_file(*path='.'*, *create_dirs=True*, *overwrite=False*)
Writes a full blockMeshDict file based on stored geometry data

write_mesh_file(*path='.'*, *create_dirs=True*, *overwrite=False*)
Passes args off to write_foam_file

write_symmetry_plane(*path='.'*, *create_dirs=True*, *overwrite=False*)
Exports the +Y half of the mesh flattening out everything below 0 on the Y axis

OpenFoam Export

A class build to handle mass exportation of OpenFoam files.

Written By: Matthew Stadelman
Date Written: 2016/08/08
Last Modifed: 2016/08/08

class apmapflow.openfoam.openfoam_export.**OpenFoamExport**(*field=None*, *avg_fact=1.0*, *mesh_params=None*)

A class to handle generation and exporting of OpenFoam files

__init__(*field=None*, *avg_fact=1.0*, *mesh_params=None*)
Handles generation and exporting of OpenFoam files

__weakref__
list of weak references to the object (if defined)

generate_block_mesh_dict(*field*, *avg_fact=1.0*, *mesh_params=None*)
Passes arguments off to BlockMeshDict init method.

generate_foam_files(**args*)
Generates, reads and adds OpenFoam files to the export to provide a centralized location to perform modifications and write files. Any number of 'file' arguments may be passed into this function but they are required to have one of the following three forms:

1. An already existing OpenFoamFile instances
2. A string representing the path to a valid OpenFoamFile
3. **An iterable acceptable to the dictionary constructor with at** minimum 2 keys: location and object; 'class' is an optional key. Those 2/3 keys are removed and the rest of the iterable is passed along to the OpenFoamFile __init__ method as the 'values' keyword.

Files are stored on the export object in a dictionary attribute called 'foam_files'. Keys in this dictionary have the format of 'location.name'. Where 'name' is 'name' attribute on the OpenFoamFile.

write_foam_files(*path='.'*, *overwrite=False*)
Writes the files generated by 'generate_foam_files' to their associated directories on the supplied path. If a directory doesn't exist then it is created

write_mesh_file(*path='.'*, *create_dirs=True*, *overwrite=False*)
Passes arguments off to the BlockMeshDict method

write_symmetry_plane(*path='.'*, *create_dirs=True*, *overwrite=False*)
Passes arguments off to the BlockMeshDict method

OpenFoam Core

This stores the basic classes and functions needed to interact with OpenFoam files.

Written By: Matthew Stadelman
Date Written: 2016/03/22
Last Modifed: 2016/08/08

class apmapflow.openfoam.openfoam.**OpenFoamDict**(*name*, *values=None*)
Class used to build the dictionary style OpenFoam input blocks

__init__(*name*, *values=None*)

Creates an OpenFoamDict: name - string printed at top of dictionary in files values - any valid iterable that can be used to initialize

a dictionary

__str__(*indent=0*)
Prints a formatted output readable by OpenFoam

class apmapflow.openfoam.openfoam.**OpenFoamFile**(**args*, ***kwargs*)
Class used to build OpenFoam input files

__init__(**args*, ***kwargs*)
Creates and instance of the class passing the first three arguments to the FileFile header dict and the final argument can be used to initialize the OpenFoamFile object with entries.

location [string, sets the subdirectory location of the file] during output.

object_name [string, sets initial value of self.name attribute used] to name the output file and the 'object' key in the FoamFile dict

class_name [string, optional, sets the 'class' key in the FoamFile] dict, it defaults to 'dictionary'

values [iterable, optional, any valid iterable that can be used to] initialize a regular Python dictionary

__str__()
Prints a formatted OpenFoam input file

static _init_from_file(*filename*)
Reads an existing OpenFoam input file and returns an OpenFoamFile instance. Comment lines are not retained.

write_foam_file(*path='.'*, *create_dirs=True*, *overwrite=False*)
Writes out the foam file, adding proper location directory if create_dirs is True

class apmapflow.openfoam.openfoam.**OpenFoamList**(*name*, *values=None*)
Class used to build the output lists used in blockMeshDict.

__init__(*name*, *values=None*)

Creates an OpenFoamList: name - string printed at top of dictionary in files values - any valid iterable that can be used to initialize

a list

__str__(*indent=0*)
Prints a formatted output readable by OpenFoam

__weakref__
list of weak references to the object (if defined)

class apmapflow.openfoam.openfoam.**OpenFoamObject**
General class used to recognize other OpenFoam objects

__weakref__
list of weak references to the object (if defined)

Parallel Mesh Generation

A class build to handle generation of a blockMeshDict in parallel using the
mergeMeshes and stitchMesh OpenFoam utilities.

Written By: Matthew Stadelman
Date Written: 2016/08/09
Last Modifed: 2016/08/15

class apmapflow.openfoam.parallel_mesh_gen.**BlockMeshRegion**(*region*, *avg_fact*, *x_shift*, *z_shift*, *mesh_params*, *offset_reg*)

Used to handle a sub-set of field point data for parallel mesh generation

__init__(*region*, *avg_fact*, *x_shift*, *z_shift*, *mesh_params*, *offset_reg*)

Takes a field object and a set of mesh params to set the properties of the blockMeshDict. The x_shift and z_shift values are multipled by avg_fact.

data_region : DataFieldRegion object avg_fact : float, number of voxels along the X and Z axes. x_shift : int, number of voxels shifted from orgin z_shift : int, number of voxels shifted from origin mesh_params : dict, a dictionary containing parameters to offsets : DataFieldRegion object use instead of the defaults

_create_blocks(*cell_mask=None*)

Generates blocks and verticies the same as the parent class and then applies the x and z shift

run_block_mesh(*mesh_type*, *path*, *system_dir*, *t_name*, *overwrite*)

Writes the blockMeshDict and runs blockMesh on a region

class apmapflow.openfoam.parallel_mesh_gen.**DataFieldRegion**(*data*, *point_data*)

Used to manipulate a specfic data region of a DataField. In order to maintain data integrity point data is not able to be recalculated here.

class apmapflow.openfoam.parallel_mesh_gen.**MergeGroup**(*region_id*, *external_patches*, *path*)

Handles merging of meshes and stitching any patches that become internal

__init__(*region_id*, *external_patches*, *path*)

Sets up the initial merge group as a single region. External patches is a dictionary with an entry for each side, {side: 'patch_name'}. The attribute 'external_patches' has the same format except its entries for each side are lists.

__weakref__

list of weak references to the object (if defined)

static _clean_polymesh(*path*)

Removes files left over from merging and stitching meshes together

merge_regions(*region_in*, *direction*, *t_name*)

Merges two regions updating the external_patches dict and stitching and patches that become internal. It is assumed the region_in will be a higher 'id' than this region.

stitch_patches(*internal_patches*, *t_name*)

Stitches all internal patches in the region together

class apmapflow.openfoam.parallel_mesh_gen.**ParallelMeshGen**(*field*, *system_dir*, *nprocs=4*, ***kwargs*)

Handles creation of a large mesh in parallel utilizing the OpenFoam utilties mergeMesh and

stitchMesh.

__weakref__
list of weak references to the object (if defined)

static _create_merge_queue(*grid*, *direction*)
Determines the region merge queue based on the grid supplied

_create_regions_thread(*region_queue*, *t_name*, *kwargs*)
Handles processing of the queue in it's own thread

_create_subregion_meshes(*ndivs*, ***kwargs*)
Divides the data map into smaller regions and creates a BlockMeshRegion object for each one.

_merge_submeshes(*grid*)
Handles merging and stitching of meshes based on alternating rounds of horizontal pairing and then vertical pairing.

static _remove_leftover_patches(*path*)
Removes all left over merge patches from theboundary file

_setup_region(*region_id*, *z_slice*, *x_slice*, ***kwargs*)
sets up an individual mesh region

generate_mesh(*mesh_type='simple'*, *path='.'*, *ndivs=8*, ***kwargs*)
Generates multiple mesh types and outputs them to a specific path. Valid mesh_types are: simple, threshold and symmetry. Additional kwargs need to be supplied for the given mesh type if it needs additional keywords.

class apmapflow.openfoam.**OpenFoamDict**(*name*, *values=None*)
Class used to build the dictionary style OpenFoam input blocks

class apmapflow.openfoam.**OpenFoamList**(*name*, *values=None*)
Class used to build the output lists used in blockMeshDict.

class apmapflow.openfoam.**OpenFoamFile**(**args*, ***kwargs*)
Class used to build OpenFoam input files

write_foam_file(*path='.'*, *create_dirs=True*, *overwrite=False*)
Writes out the foam file, adding proper location directory if create_dirs is True

class apmapflow.openfoam.**BlockMeshDict**(*field*, *avg_fact=1.0*, *mesh_params=None*, *offset_field=None*)
This is a special subclass of OpenFoamFile used to generate and output a blockMeshDict for OpenFoam

generate_mesh_file()
Populates keys on itself based on geometry data to output a mesh file

generate_simple_mesh()
Generates a simple mesh including all cells in the data map

generate_threshold_mesh(*min_value=0.0*, *max_value=1000000000.0*)
Generates a mesh excluding all blocks below the min_value arg. Regions that are isolated by the thresholding are also automatically removed.

set_boundary_patches(*boundary_blocks*, *reset=False*)
Sets up boundary patches based on the dictionary passed in. Overlapping declarations are overwritten by the last patch to use that face. The boundary blocks dictionary contains a dictionary entry for each patch name.

- **boundary_blocks dictionary has the format of:**

{patch_name: {

<side>: [block-list], <side>: [block-list], ...

} where <side> is left, right, bottom, top, front or back and block list is a list of blocks to add that patch to the side of.

- **reset - boolean** [if True then the face labels dictionary] and _faces array are re-initialized to default values

write_foam_file(*path='.'*, *create_dirs=True*, *overwrite=False*)
Writes a full blockMeshDict file based on stored geometry data

write_mesh_file(*path='.'*, *create_dirs=True*, *overwrite=False*)
Passes args off to write_foam_file

write_symmetry_plane(*path='.'*, *create_dirs=True*, *overwrite=False*)
Exports the +Y half of the mesh flattening out everything below 0 on the Y axis

class apmapflow.openfoam.**OpenFoamExport**(*field=None*, *avg_fact=1.0*, *mesh_params=None*)
A class to handle generation and exporting of OpenFoam files

generate_block_mesh_dict(*field*, *avg_fact=1.0*, *mesh_params=None*)
Passes arguments off to BlockMeshDict init method.

generate_foam_files(**args*)
Generates, reads and adds OpenFoam files to the export to provide a centralized location to perform modifications and write files. Any number of 'file' arguments may be passed into this function but they are required to have one of the following three forms:

1. An already existing OpenFoamFile instances
2. A string representing the path to a valid OpenFoamFile
3. **An iterable acceptable to the dictionary constructor with at** minimum 2 keys: location and object; 'class' is an optional key. Those 2/3 keys are removed and the rest of the iterable is passed along to the OpenFoamFile __init__ method as the 'values' keyword.

Files are stored on the export object in a dictionary attribute called 'foam_files'. Keys in this dictionary have the format of 'location.name'. Where 'name' is 'name' attribute on the OpenFoamFile.

write_foam_files(*path='.'*, *overwrite=False*)
Writes the files generated by 'generate_foam_files' to their associated directories on the supplied path. If a directory doesn't exist then it is created

write_mesh_file(*path='.'*, *create_dirs=True*, *overwrite=False*)
Passes arguments off to the BlockMeshDict method

write_symmetry_plane(*path='.'*, *create_dirs=True*, *overwrite=False*)
Passes arguments off to the BlockMeshDict method

class apmapflow.openfoam.**ParallelMeshGen**(*field*, *system_dir*, *nprocs=4*, ***kwargs*)
Handles creation of a large mesh in parallel utilizing the OpenFoam utilties mergeMesh and stitchMesh.

generate_mesh(*mesh_type='simple'*, *path='.'*, *ndivs=8*, ***kwargs*)
Generates multiple mesh types and outputs them to a specific path. Valid mesh_types are: simple, threshold and symmetry. Additional kwargs need to be supplied for the given mesh type if it needs additional keywords.

Run Model

This stores the basic classes and functions needed to the run model

Written By: Matthew Stadelman
Date Written: 2016/06/16
Last Modifed: 2017/04/05

Bulk Run

This stores BulkRun class used for running multiple concurrent simulations

Written By: Matthew Stadelman
Date Written: 2016/06/16
Last Modifed: 2016/06/16

class apmapflow.run_model.bulk_run.**BulkRun**(*init_input_file*, *num_CPUs=2*, *sys_RAM=4.0*, ***kwargs*)

Handles generating a collection of input files from the provided parameters and then running multiple instances of the LCL model concurrently to produce simulation data for each simulation parameter combination. A comprehensive example of this class and the associated script is avilable under the Usage Examples page.

Parameters

- **init_input_file** (apmapflow.run_model.InputFile) – An inital InputFile instance to define the static parameters of the bulk run.
- **num_CPUs** (*int, optional*) – The maximum number of CPUs to utilize
- **sys_RAM** (*float, optional*) – The maximum amount of RAM avilable for use.
- ****kwargs** (*multiple*) –
  - **delim** [string] The expected delimiter in the aperture map files
  - **spawn_delay** [float] The minimum time between spawning of new LCL instances in seconds
  - **retest_delay** [float] The time to wait between checking for completed processes.

Examples

```
>>> from apmapflow import BulkRun, InputFile
>>> inp_file = InputFile('./input-file-path.inp')
>>> blk_run = BulkRun(inp_file, num_CPUs=16, sys_RAM=32.0, spawn_
↪delay=10.0)
```

Notes

`spawn_delay` is useful to help ensure shared resources are not accessed at the same time.

__init__(*init_input_file*, *num_CPUs=2*, *sys_RAM=4.0*, ***kwargs*)
Setting properties of the class.

__weakref__
list of weak references to the object (if defined)

static _check_processes(*processes*, *RAM_in_use*, *retest_delay=5*, ***kwargs*)
Checks the list of currently running processes for any that have completed removing and them from a list. If no processes have completed then the routine sleep for a specified amount of time before checking again.

Parameters

- **processes** (*list of Popen instances*) – The list of processes to curate.
- **RAM_in_use** (*list of floats*) – The list of maximum RAM each process is estimated to use.
- **retest_delay** (*floats*) – The time delay between testing for completed processes.

static _combine_run_params(*run_params*)
Generates all possible unique combinations from a set of parameter arrays.

Parameters **run_params** (*dictionary*) – A dictionary of parameter lists to combine together

Returns **parameter combinations** – A list of dictionaries where each parameter only has a single value

Return type dictionary

_initialize_run()
Assesses RAM requirements of each aperture map in use and registers the value with the InputFile instance. This RAM measurement is later used when determining if there is enough space available to begin a simulation.

_start_simulations(*processes*, *RAM_in_use*, *spawn_delay=5*, ***kwargs*)
This starts additional simulations if there is enough free RAM and avilable CPUs.

Parameters

- **processes** (*list of Popen instances*) – The list of processes to add any new simulations to.
- **RAM_in_use** (*list of floats*) – The list of maximum RAM to a new simulations requirement to.
- **spawn_delay** (*floats*) – The time delay between spawning of processes.

dry_run()
Steps through the entire simulation creating directories and input files without actually starting any of the simulations. This Allows the LCL input files to be inspected before actually starting the run.

Examples

```
>>> from apmapflow import BulkRun, InputFile
>>> inp_file = InputFile('./input-file-path.inp')
>>> blk_run = BulkRun(inp_file, num_CPUs=16, sys_RAM=32.0, spawn_
↪delay=10.0)
>>> blk_run.dry_run()
```

See also:

start()

generate_input_files(*default_params*, *default_name_formats*, *case_identifer=''*, *case_params=None*, *append=False*)

Generates the input file list based on the default parameters and case specific parameters. An InputFile instance is generated for each unique combination of model parameters which is then written to disk to be run by the LCL model.

Parameters

- **default_params** (*dictionary*) – A dictionary containing lists of parameter values to use in the simulations.
- **default_name_formats** (*dictionary*) – A dictionary containing the infile and outfile name formats to use.
- **case_identifer** (*string, optional*) – A format string used to identify cases that need special parameters
- **case_params** (*dictionary, optional*) – A dictionary setup where each key is an evaluation of the case_identifer format string and the value is a dictionary containing lists of parameter values used to update the default params for that case.
- **append** (*boolean, optional*) – When `True` the BulkRun.input_file_list attribute is appended to instead of reset by this method.

Notes

The `default_name_formats` parameter is passed directly to the InputFile instance initialization and is no modified in any way. When using a `case_identifier` only the evaluations that matter need to be added to the `case_params` dictionary. Missing permuatations of the identifer are

start()

Starts the bulk run, first creating the input files and then managing the multiple processes until all input files have been processed. The input file list must have already been generated prior to calling this method.

See also:

generate_input_files()

Run Model Core

This stores the basic classes and functions needed to the run model

Written By: Matthew Stadelman
Date Written: 2016/06/16
Last Modifed: 2017/04/05

class apmapflow.run_model.run_model.**ArgInput**(*line*)

Stores the value of a single input line of a LCL model input file. Instances of this class are stored in an InputFile instance's dict to manage each parameter.

Parameters **line** (*string*) – input line to parse.

__init__(*line*)

Parses the line for the input key string and value.

__str__()

Allows direct printing of an ArgInput object in output format.

Examples

```
>>> from apmapflow.run_model.run_model import ArgInput
>>> param = ArgInput('test-param: 123 ft')
>>> print(param)
test-param: 123 ft
```

See also:

line()

__weakref__

list of weak references to the object (if defined)

_parse_line(*line*)

Parses a line to set attributes of the class instance.

Parameters **line** (*string*) – input line to parse.

keyword

Returns the keyword used to register this instance to an InputFile instance.

line

Return a formatted line meant for use when writing an InputFile isntance to disk. The line is prefixed by `;` if the parameter is supposed to be commented out.

Examples

```
>>> from apmapflow.run_model.run_model import ArgInput
>>> param = ArgInput('test-param: 123 ft')
>>> param.line
'test-param: 123 ft'
>>> param.commented_out = True
>>> param.line
';test-param: 123 ft'
```

See also:

__str__

unit
Returns the given units a value is in or None, and can also be used to set the units of a value.

Parameters **value** (*string, optional*) – If supplied with a value then the units of a instance are set to it.

Examples

```
>>> from apmapflow.run_model.run_model import ArgInput
>>> param = ArgInput('test-param: 123 ft')
>>> param.unit
'ft'
>>> param.unit = 'mm'
>>> param.unit
'mm'
```

value
Returns the value of the parameter stored by the class instance or sets the value of the instance. When setting the value if a tuple is passed instead of a scalar the first entry is used to set the value and the second's truth value sets the .commented out attribute of the instance.

Parameters **value** (*scalar or tuple, optional*) – When value is supplied the property is set

Examples

```
>>> from apmapflow.run_model.run_model import ArgInput
>>> param = ArgInput('test-param: value-123')
>>> param.value
'value-123'
>>> param.commented_out
False
>>> param.value = 'new-value'
>>> param.value
'new-value'
>>> param.value = ('commented-out', True)
>>> param.value
'commented-out'
>>> param.commented_out
True
```

class apmapflow.run_model.run_model.**AsyncCommunicate**(*popen_obj*)
Allows an executable to be run in a separate thread so it does not block the main Python process.

Parameters **popen_obj** (*Popen instance*) – An instance containing a process that is currently executing.

run()
Calls the communicate method on the Popen object registered to the class which blocks the thread until the process terminates. The total execution time, stdout and stderr of the process are passed back to the Popen object.

class apmapflow.run_model.run_model.**InputFile**(*infile*, *filename_formats=None*)
Used to read and write and manipulate LCL model input files. Each key-value pair stored on an instance of this class is actually an instance of the ArgInput class.

Parameters

- **infile** (*string or InputFile instance*) – Either is a filepath to read an input file from or an existing instance to copy.
- **filename_formats** (*dictionary, optional*) – A dictionary of filename formats which use Python format strings to dynamically generate names for the LCL model input and output files.

Examples

```
>>> from apmapflow.run_model import InputFile
>>> inp_file = InputFile('input-file-path.inp')
>>> fmts = {'VTK-FILE': '{apmap}-data.vtk'}
>>> inp_file2 = InputFile(inp_file, filename_formats=fmts)
```

Notes

Any `filename_formats` defined will overwrite a parameter that was manually defined by directly setting the value. The `__setitem__` method of has been subclassed to transparently update the value attribute of the ArgInput instance for the corresponding key instead of the value itself.

__setitem__(*key*, *value*, *new_param=False*)

Subclassed to pass the value directly to the value attribute of the ArgInput instance stored on the provided key unless the `new_param` argument evaluates to `True`.

Parameters

- **key** (*string*) – The key on the dictionary to update
- **value** (*string or ArgInput instance*) – The value to set the given key to
- **new_param** (*boolean, optional*) – If `True` then the value is not passed on to the ArgInput instance and the actual value on the InputFile instance is changed.

Examples

```
>>> from apmapflow.run_model import InputFile
>>> inp_file = InputFile('input-file-path.inp')
>>> inp_file['parameter'] = '123'
>>> inp_file['parameter']
<apmapflow.run_model.run_model.ArgInput object at #########>
>>> inp_file['parameter'].value
'123'
>>> inp_file.__setitem__('parameter', 'param-value', new_param=True)
>>> inp_file['parameter']
'param-value'
```

Notes

If `new_param` is falsy and the key does not already exist a KeyError exeception is raised. The `add_parameter` method is the standard way to add new ArgInput instances to the InputFile instance

See also:

add_parameter()

__str__()
Writes out the input file as if it was being written to disk.

_construct_file_names(*make_dirs=False*)
This updates the instance's outfile names to match current arguments.

Parameters **make_dirs** (*boolean, optional*) – If `make_dirs` evaluates to True then all parent directories for each output file are created as well.

add_parameter(*line*)
Adds a parameter to the input file by parsing a line into an ArgInput class instance. The line supplied needs to be the same as if it were being manually typed into the actual input file.

Parameters **line** (*string*) – The provided line to parse and append to the InputFile instance.

Examples

```
>>> from apmapflow.run_model import InputFile
>>> inp_file = InputFile('input-file-path.inp')
>>> inp_file.add_parameter('NEW-PARAM: 1337 ;elite param')
>>> inp_file['NEW-PARAM'].value
'1337'
```

Notes

The InputFile inheirits from an OrderedDict so new parameters get added to the bottom of the file.

clone(*file_formats=None*)
Creates a new InputFile instance populated with the current instance data. New ArgInput instances are created to prevent mutation.

Parameters **filename_formats** (*dictionary, optional*) – A dictionary of filename formats which use Python format strings to dynamically generate names for the LCL model input and output files.

Returns **input_file** – The cloned input file instance

Return type *apmapflow.run_model.InputFile*

Examples

```
>>> from apmapflow.run_model import InputFile
>>> fmts = {'VTK-FILE': '{apmap}-data.vtk'}
>>> inp_file = InputFile('input-file-path.inp', filename_
↪formats=fmts)
>>> inp_file_copy = inp_file.clone()
>>> inp_file_copy.filename_formats
{'VTK-FILE': '{apmap}-data.vtk'}
>>> inp_file_copy = inp_file.clone(file_formats={})
>>> inp_file_copy.filename_formats
{}
```

Notes

If the `filename_formats` parameter is omitted then the formats from the current instance are copied over.

get_uncommented_values()
Generate and return all uncommented parameters as an OrderedDict.

Returns uncommented_values – An OrderedDict containing all ArgInput instances that were not commented out

Return type OrderedDict

parse_input_file(*infile*)
Populates the InputFile instance with data from a file or copies an existing instance passed in.

Parameters infile (*string or InputFile instance*) – Either is a filepath to read an input file from or an existing InputFile instance to copy.

See also:

InputFile(), *clone()*

set_executable()
Sets the path to an LCL model executable based on the `EXE-FILE` parameter for the current InputFile instance. The path is checked relative to the InputFile instance's `infile` attribute. If no file is found a warning is issued and the executable path defaults to the version packaged with the module.

Examples

```
>>> from apmapflow.run_model import InputFile
>>> inp_file = InputFile('./input-file-path.inp')
>>> inp_file['EXE-FILE'] = 'my-locally-compiled-model.exe'
>>> inp_file.set_executable()
>>> inp_file.executable
'./my-locally-compiled-model.exe'
```

Notes

This method needs to be called if the `EXE-FILE` parameter is added, changed or removed.

update (**args*, ***kwargs*)

Updates the InputFile instance, passing any unknown keys to the filename_format_args dictionary instead of raising a KeyError like in __setitem__

Parameters ****kwargs** (**args,*) – The resulting dictionary formed internally is used to update the instance

Examples

```
>>> from apmapflow.run_model import InputFile
>>> fmts = {'VTK-FILE': '{apmap}-data.vtk'}
>>> inp_file = InputFile('input-file-path.inp', filename_
↪formats=fmts)
>>> new_vals = {'OUTLET-SIDE': 'LEFT', 'apmap': 'fracture-1'}
>>> inp_file.update(new_vals)
>>> inp_file['OUTLET-SIDE'].value
'LEFT'
>>> inp_file.filename_format_args
{'apmap': 'fracture-1'}
```

write_inp_file (*alt_path=None*)

Writes an input file to the `outfile_name` attribute applying any formats defined based the current parameter values.

Parameters

- **alt_path** (*string,*) – An alternate path to preappend to the generated filename
- **from apmapflow.run_model import InputFile** (*>>>*) –
- **inp_file = InputFile('input-file-path.inp')** (*>>>*) –
- **inp_file.write_inp_file(alt_path='.')** (*>>>*) –

apmapflow.run_model.run_model.**estimate_req_RAM**(*input_maps*, *avail_RAM=inf*, *suppress=False*, ***kwargs*)

Reads in the input maps to estimate the RAM requirement of each map and to make sure the user has alloted enough space. The RAM estimation is a rough estimate based on a linear regression of several simulations of varying aperture map size.

Parameters

- **input_maps** (*list*) – A list of filepaths to read in
- **avail_RAM** (*float, optional*) – The maximum amount of RAM available on the system to be used. When exceeded an EnvironmentError is raised and an error message is generated.
- **suppress** (*boolan, optional*) – If it evaluates out to True and a map exceeds the `avail_RAM` the EnvironmentError is suppressed.
- ****kwargs** (*optional*) – Additional keyword args to pass on to the DataField initialization.

Returns **ram_values** – A list of floats with the corresponding RAM estimate for each of the maps passed in.

Return type list

Examples

```
>>> from apmapflow.run_model import estimate_req_RAM
>>> maps = ['fracture-1.txt', 'fracture-2.txt', 'fracture-3.txt']
>>> estimate_req_RAM(maps, avail_RAM=8.0, suppress=True)
[6.7342, 8.1023, 5.7833]
```

apmapflow.run_model.run_model(*input_file_obj*, *synchronous=False*, *show_stdout=False*)

Runs an instance of the LCL model defined by the InputFile instance passed in.

Parameters

- **input_file_obj** (apmapflow.run_model.InputFile) – An InputFile instance with the desired simulation parameters to run.
- **synchronous** (*boolean, optional*) – If True then run_model will block the main Python execution thread until the simulation is complete.
- **show_stdout** (*boolean, optional*) – If True then the stdout and stderr produced during the simulation run are printed to the screen instead of being stored on the Popen instance

Returns **model_popen_obj** – The Popen instance that contains the LCL model process, which may or man not be finished executing at upon return.

Return type Popen

Examples

```
>>> from apmapflow.run_model import InputFile, run_model
>>> inp_file = InputFile('input-file-path.inp')
>>> proc = run_model(inp_file) # asynchronous run process isn't␣
↪completed yet
>>> proc.returncode
None
>>> # process is finished upon return when using synchronous=True
>>> proc2 = run_model(inp_file, synchronous=True)
>>> proc2.returncode
0
```

Notes

This writes out the inputfile at the perscribed path, a pre-existing file will be overwritten.

class apmapflow.run_model.**InputFile**(*infile*, *filename_formats=None*)

Used to read and write and manipulate LCL model input files. Each key-value pair stored on an instance of this class is actually an instance of the ArgInput class.

Parameters

- **infile** (*string or InputFile instance*) – Either is a filepath to read an input file from or an existing instance to copy.
- **filename_formats** (*dictionary, optional*) – A dictionary of filename formats which use Python format strings to dynamically generate names for the LCL model input and output files.

Examples

```
>>> from apmapflow.run_model import InputFile
>>> inp_file = InputFile('input-file-path.inp')
>>> fmts = {'VTK-FILE': '{apmap}-data.vtk'}
>>> inp_file2 = InputFile(inp_file, filename_formats=fmts)
```

Notes

Any `filename_formats` defined will overwrite a parameter that was manually defined by directly setting the value. The `__setitem__` method of has been subclassed to transparently update the value attribute of the ArgInput instance for the corresponding key instead of the value itself.

add_parameter(*line*)

Adds a parameter to the input file by parsing a line into an ArgInput class instance. The line supplied needs to be the same as if it were being manually typed into the actual input file.

Parameters **line** (*string*) – The provided line to parse and append to the InputFile instance.

Examples

```
>>> from apmapflow.run_model import InputFile
>>> inp_file = InputFile('input-file-path.inp')
>>> inp_file.add_parameter('NEW-PARAM: 1337 ;elite param')
>>> inp_file['NEW-PARAM'].value
'1337'
```

Notes

The InputFile inheirits from an OrderedDict so new parameters get added to the bottom of the file.

clone(*file_formats=None*)

Creates a new InputFile instance populated with the current instance data. New ArgInput instances are created to prevent mutation.

Parameters **filename_formats** (*dictionary, optional*) – A dictionary of filename formats which use Python format strings to dynamically generate names for the LCL model input and output files.

Returns **input_file** – The cloned input file instance

Return type *apmapflow.run_model.InputFile*

Examples

```
>>> from apmapflow.run_model import InputFile
>>> fmts = {'VTK-FILE': '{apmap}-data.vtk'}
>>> inp_file = InputFile('input-file-path.inp', filename_
↪formats=fmts)
>>> inp_file_copy = inp_file.clone()
```

```
>>> inp_file_copy.filename_formats
{'VTK-FILE': '{apmap}-data.vtk'}
>>> inp_file_copy = inp_file.clone(file_formats={})
>>> inp_file_copy.filename_formats
{}
```

Notes

If the `filename_formats` parameter is omitted then the formats from the current instance are copied over.

get_uncommented_values()
Generate and return all uncommented parameters as an OrderedDict.

Returns uncommented_values – An OrderedDict containing all ArgInput instances that were not commented out

Return type OrderedDict

parse_input_file(*infile*)
Populates the InputFile instance with data from a file or copies an existing instance passed in.

Parameters infile (*string or InputFile instance*) – Either is a filepath to read an input file from or an existing InputFile instance to copy.

See also:

InputFile(), *clone()*

set_executable()
Sets the path to an LCL model executable based on the `EXE-FILE` parameter for the current InputFile instance. The path is checked relative to the InputFile instance's `infile` attribute. If no file is found a warning is issued and the executable path defaults to the version packaged with the module.

Examples

```
>>> from apmapflow.run_model import InputFile
>>> inp_file = InputFile('./input-file-path.inp')
>>> inp_file['EXE-FILE'] = 'my-locally-compiled-model.exe'
>>> inp_file.set_executable()
>>> inp_file.executable
'./my-locally-compiled-model.exe'
```

Notes

This method needs to be called if the `EXE-FILE` parameter is added, changed or removed.

update(**args*, ***kwargs*)
Updates the InputFile instance, passing any unknown keys to the filename_format_args dictionary instead of raising a KeyError like in __setitem__

Parameters **kwargs (**args,*) – The resulting dictionary formed internally is used to update the instance

Examples

```
>>> from apmapflow.run_model import InputFile
>>> fmts = {'VTK-FILE': '{apmap}-data.vtk'}
>>> inp_file = InputFile('input-file-path.inp', filename_
↪formats=fmts)
>>> new_vals = {'OUTLET-SIDE': 'LEFT', 'apmap': 'fracture-1'}
>>> inp_file.update(new_vals)
>>> inp_file['OUTLET-SIDE'].value
'LEFT'
>>> inp_file.filename_format_args
{'apmap': 'fracture-1'}
```

write_inp_file(*alt_path=None*)

Writes an input file to the `outfile_name` attribute applying any formats defined based the current parameter values.

Parameters

- **alt_path** (*string,*) – An alternate path to preappend to the generated filename
- **from apmapflow.run_model import InputFile** (*>>>*) –
- **inp_file = InputFile('input-file-path.inp')** (*>>>*) –
- **inp_file.write_inp_file(alt_path='.')** (*>>>*) –

class apmapflow.run_model.**BulkRun**(*init_input_file*, *num_CPUs=2*, *sys_RAM=4.0*, ***kwargs*)

Handles generating a collection of input files from the provided parameters and then running multiple instances of the LCL model concurrently to produce simulation data for each simulation parameter combination. A comprehensive example of this class and the associated script is avilable under the Usage Examples page.

Parameters

- **init_input_file** (apmapflow.run_model.InputFile) – An inital InputFile instance to define the static parameters of the bulk run.
- **num_CPUs** (*int, optional*) – The maximum number of CPUs to utilize
- **sys_RAM** (*float, optional*) – The maximum amount of RAM avilable for use.
- ****kwargs** (*multiple*) –
  - **delim** [string] The expected delimiter in the aperture map files
  - **spawn_delay** [float] The minimum time between spawning of new LCL instances in seconds
  - **retest_delay** [float] The time to wait between checking for completed processes.

Examples

```
>>> from apmapflow import BulkRun, InputFile
>>> inp_file = InputFile('./input-file-path.inp')
>>> blk_run = BulkRun(inp_file, num_CPUs=16, sys_RAM=32.0, spawn_
↪delay=10.0)
```

Notes

`spawn_delay` is useful to help ensure shared resources are not accessed at the same time.

dry_run()

Steps through the entire simulation creating directories and input files without actually starting any of the simulations. This Allows the LCL input files to be inspected before actually starting the run.

Examples

```
>>> from apmapflow import BulkRun, InputFile
>>> inp_file = InputFile('./input-file-path.inp')
>>> blk_run = BulkRun(inp_file, num_CPUs=16, sys_RAM=32.0, spawn_
↪delay=10.0)
>>> blk_run.dry_run()
```

See also:

start()

generate_input_files(*default_params*, *default_name_formats*, *case_identifer=''*, *case_params=None*, *append=False*)

Generates the input file list based on the default parameters and case specific parameters. An InputFile instance is generated for each unique combination of model parameters which is then written to disk to be run by the LCL model.

Parameters

- **default_params** (*dictionary*) – A dictionary containing lists of parameter values to use in the simulations.
- **default_name_formats** (*dictionary*) – A dictionary containing the infile and outfile name formats to use.
- **case_identifer** (*string, optional*) – A format string used to identify cases that need special parameters
- **case_params** (*dictionary, optional*) – A dictionary setup where each key is an evaluation of the case_identifer format string and the value is a dictionary containing lists of parameter values used to update the default params for that case.
- **append** (*boolean, optional*) – When `True` the BulkRun.input_file_list attribute is appended to instead of reset by this method.

Notes

The `default_name_formats` parameter is passed directly to the InputFile instance initialization and is no modified in any way. When using a `case_identifier` only the evaluations that matter need to be added to the `case_params` dictionary. Missing permuatations of the identifer are

start()
Starts the bulk run, first creating the input files and then managing the multiple processes until all input files have been processed. The input file list must have already been generated prior to calling this method.

See also:

generate_input_files()

apmapflow.run_model.**estimate_req_RAM**(*input_maps*, *avail_RAM=inf*, *suppress=False*, ***kwargs*)
Reads in the input maps to estimate the RAM requirement of each map and to make sure the user has alloted enough space. The RAM estimation is a rough estimate based on a linear regression of several simulations of varying aperture map size.

Parameters

- **input_maps** (*list*) – A list of filepaths to read in
- **avail_RAM** (*float, optional*) – The maximum amount of RAM available on the system to be used. When exceeded an EnvironmentError is raised and an error message is generated.
- **suppress** (*boolan, optional*) – If it evaluates out to True and a map exceeds the avail_RAM the EnvironmentError is suppressed.
- ****kwargs** (*optional*) – Additional keyword args to pass on to the DataField initialization.

Returns **ram_values** – A list of floats with the corresponding RAM estimate for each of the maps passed in.

Return type list

Examples

```
>>> from apmapflow.run_model import estimate_req_RAM
>>> maps = ['fracture-1.txt', 'fracture-2.txt', 'fracture-3.txt']
>>> estimate_req_RAM(maps, avail_RAM=8.0, suppress=True)
[6.7342, 8.1023, 5.7833]
```

apmapflow.run_model.**run_model**(*input_file_obj*, *synchronous=False*, *show_stdout=False*)
Runs an instance of the LCL model defined by the InputFile instance passed in.

Parameters

- **input_file_obj** (apmapflow.run_model.InputFile) – An InputFile instance with the desired simulation parameters to run.
- **synchronous** (*boolean, optional*) – If True then run_model will block the main Python execution thread until the simulation is complete.
- **show_stdout** (*boolean, optional*) – If True then the stdout and stderr produced during the simulation run are printed to the screen instead of being stored on the Popen instance

Returns **model_popen_obj** – The Popen instance that contains the LCL model process, which may or man not be finished executing at upon return.

Return type Popen

Examples

```
>>> from apmapflow.run_model import InputFile, run_model
>>> inp_file = InputFile('input-file-path.inp')
>>> proc = run_model(inp_file) # asynchronous run process isn't␣
↪completed yet
>>> proc.returncode
None
>>> # process is finished upon return when using synchronous=True
>>> proc2 = run_model(inp_file, synchronous=True)
>>> proc2.returncode
0
```

Notes

This writes out the inputfile at the perscribed path, a pre-existing file will be overwritten.

Unit Conversion

Handles unit conversions utilizing the pint module.

Written By: Matthew Stadelman
Date Written: 2017/03/05
Last Modifed: 2017/04/30

apmapflow.unit_conversion.**convert_value**(*value*, *unit_in*, *unit_out='SI'*)

Returns a converted value, in the specified output units or SI.

Parameters

- **value** (*float*) – edge length of a voxel cube.
- **unit_in** (*string*) – the current units of the value
- **unit_out** (*string, optional*) – the desired output units, if omitted they are converted to SI

Returns

Return type A floating point value converted to the desired units.

Examples

```
>>> import apmapflow.unit_conversion as uc
>>> print(uc.convert_value(26.8, 'um'))
>>> print(uc.convert_value(26.8, 'um', 'cm'))
```

apmapflow.unit_conversion.**get_conversion_factor**(*unit_in*, *unit_out='SI'*)

Returns a conversion factor between the input unit and output unit or to SI if no output unit is provided.

Parameters

- **unit_in** (*string*) – the desired input units
- **unit_out** (*string, optional*) – the desired output units, if omitted the SI value is used

Returns

Return type A floating point value that can be used to convert between the two units.

Examples

```
>>> import apmapflow.unit_conversion as uc
>>> print(uc.get_conversion_factor('micron'))
>>> print(uc.get_conversion_factor('um', 'mm'))
```

apmapflow.unit_conversion.**register_voxel_unit**(*voxel_size*, *unit*)

Registers the 'voxel' unit with the unit_registry. voxel_size is the length of an edge of the voxel in the specified units.

Parameters

- **voxel_size** (*float*) – edge length of a voxel cube.
- **unit** (*string*) – units the voxel size is defined in.

Examples

```
>>> import apmapflow.unit_conversion as uc
>>> uc.register_voxel_unit(26.8, 'um')
>>> print(uc.unit_registry('voxel').to('m').magnitude)
```

Scripts

apm_bulk_run

Description: Parses a set of YAML formatted input files and creates a set of InputFiles to run through the LCL model in parallel. The –start flag must be supplied to run the simulations, otherwise a dry run is performed. Only the keyword args of the first YAML file are used to instantiate the bulk_run_class.

For usage information run: `apm_bulk_run -h`

Written By: Matthew stadelman
Date Written: 2016/08/04
Last Modfied: 2017/04/23

apmapflow.scripts.apm_bulk_run.**main**()
Driver function to handle parsing of command line args and setting up the bulk run

apm_combine_yaml_stat_files

Description: Recurses through a directory to find all YAML stat files based on the supplied pattern and combine them into a single CSV file. This script assumes all stat files have the same set of values.

For usage information run: `apm_combine_yaml_stat_files -h`

Written By: Matthew stadelman
Date Written: 2017/02/12
Last Modfied: 2017/04/23

apmapflow.scripts.apm_combine_yaml_stat_files.**determine_key_order**(*stat_file*)
reads the first file to determine key order

apmapflow.scripts.apm_combine_yaml_stat_files.**main**()
Driver program to handles combining YAML stat files into a single CSV file.

apmapflow.scripts.apm_combine_yaml_stat_files.**output_stat_data**(*outfile*, *key_order*, *data_list*)
Generates the combined stat output file. If all the units match then the unit will be moved up into the header otherwise it is output as an additional column

apmapflow.scripts.apm_combine_yaml_stat_files.**process_data_key**(*key*, *header*, *data_list*)
Checks the units for a data key and update lists in place. If all units match then it is moved up into the header. If not an additional column is output with the units

apm_convert_csv_stats_file

Description: Generates a YAML formatted file from a CSV stat file.

For usage information run: `apm_convert_csv_stats_file -h`

Written By: Matthew stadelman
Date Written: 2017/03/03
Last Modfied: 2017/04/23

class apmapflow.scripts.apm_convert_csv_stats_file.**StatFile**(*infile*)
Parses and stores information from a simulation statisitics file. This class helps facilitate data mining of legacy simulation results. If available the YAML formatted file should be used instead as it can be directly parsed into a dictionary by the yaml module.

parse_stat_file(*stat_file=None*)
Parses either the supplied infile or the class's infile and uses the data to populate the data_dict.

apmapflow.scripts.apm_convert_csv_stats_file.**main**()
Driver function to process the stat file.

apm_fracture_df

Description: Reads in a binary fracture image stack and calculates the fractal dimension (Df) along either the X and/or Z axis.

For usage information run: `apm_fracture_df -h`

Written By: Matthew stadelman
Date Written: 2016/11/17
Last Modfied: 2017/04/23

class apmapflow.scripts.apm_fracture_df.**FractureSlice**(*slice_data*)
Stores the fractal information for a single slice

Fractal
alias of `fractal`

bifurcation_frac
calculates fractional bifurcation

set_fractal(*key*, *data*)
Adds a named tuple containing fractal data to the fractals dict

slice_data
returns the slice data

zero_ap_frac
calculates fractional bifurcation

apmapflow.scripts.apm_fracture_df.**calculate_df**(*line_trace*)
Code initially written by Charles Alexander in 2013 Modified by Dustin Crandall in 2015 Rewritten into Python by Matt Stadelman 11/15/2016

Based on the methodology presented in Dougan, Addison & McKenzie's 2000 paper "Fractal Analysis of Fracture: A Comparison of Methods" in Mechanics Research Communications.

The Hurst exponent is defined as equal to 2 - the fractal dimension of a fracture profile

H = 2 - Df Df = 2 - H

The method calculates the Hurst Exponent of a fracture profile using the "Variable Bandwidth Method", wherein a window of size 's' is moved along the fracture profile and the standard deviation of the displacement of the profile at the ends of the window is calculated

Formulation:

```
                    N-s
sigma_s = [(1/N-s)*SUM((Profile_height(i) - Profile_height(i+s))^2)]^1/2
                    i=1
```

where sigma_s = incremental standard deviation N = number of data points Profile_height(x) = height of profile at x

H is determined as the slope of the plot of log10(sigma_s) against log10(s)

returns a range of standard deviations for each bandwidth useds

apmapflow.scripts.apm_fracture_df.**df_best_fit**(*df_data*)
With large window sizes the realtionship between the log10(window size) and the log10(standard deviation of change in height) does not appear to follow a linear trend, which we should see to calculate the Hurst Exponent

This function is designed to find where the R^2 value of a linear fit to the data is minimum

apmapflow.scripts.apm_fracture_df.**find_profiles**(*fracture_slice*)
Takes in a 2-D data slice and generates line traces for JRC and Df analysis.

Returns a dictionary of the top, bottom and midsurface traces as well as the fraction of bifurcations and zero aperture zones.

apmapflow.scripts.apm_fracture_df.**main**()
Driver program to load an image and process it to output hurst exponents

apmapflow.scripts.apm_fracture_df.**output_data**(*file_handle*, *traces*, *x_data=None*, *z_data=None*)
generates a tab delimited text file to store all of the data

apmapflow.scripts.apm_fracture_df.**process_slice**(*slice_data*, *traces*)
Processes slice data to measure the changes in the trace height along the fracture, using the variable bandwidth method, then find the best fit linear line to calculate the Hurst exponent.

apm_generate_aperture_map

Description: Generates a 2-D aperture map based on a binary image stack. You can add the format descriptor {image_file} in the aperture_map_name and it will be automatically replaced by the basename of the image file used.

For usage information run: `apm_generate_aperture_map -h`

Written By: Matthew stadelman
Date Written: 2016/09/13
Last Modfied: 2017/04/23

apmapflow.scripts.apm_generate_aperture_map.**gen_colored_image_stack**(*img_data*, *aperture_map*)
Handles producing a colored image

apmapflow.scripts.apm_generate_aperture_map.**main**()
Driver function to generate an aperture map from a TIF image.

apm_process_data_map

Description: Processes data maps using the desired data_processing class and either writes data to file or prints to screen.

For usage information run: `apm_process_data_map -h`

Written By: Matthew stadelman
Date Written: 2015/10/01
Last Modfied: 2017/04/23

apmapflow.scripts.apm_process_data_map.**main**()
Parses command line arguments and delegates tasks to helper functions for actual data processing

apmapflow.scripts.apm_process_data_map.**process_files**(*args*)
Handles processing of the input maps based on the supplied arguments

apm_process_image_stack

Description: Processes a binary tif stack, with the option to remove disconnected voxels based on an undirected graph. The number of clusters to retain can be specified and connectivity is defined on a 26 point basis, i.e faces, edges and corners. Standard outputs include the processed tif image stack, an aperture map and offset map based on the processed image. Offset maps are filtered based on gradient steepness to provide a smoother surface. Data gaps left by zero apeture zones or filtering are filled by linear and nearest interpolation methods to prevent artificial features.

For usage information run: `apm_process_image_stack -h`

Written By: Matthew stadelman
Date Written: 2016/08/30
Last Modfied: 2017/04/23

apmapflow.scripts.apm_process_image_stack.**calculate_offset_map**(*img_data*)
Handles calculation of an offset map based on image data

apmapflow.scripts.apm_process_image_stack.**filter_high_gradients**(*data_map*)
Filters the offset field to reduce the number of very steep gradients. The magnitude of the gradient is taken and all values less than or greater than +-99th percentile are removed and recalculated.

apmapflow.scripts.apm_process_image_stack.**generate_adjacency_matrix**(*conns*, *nonzero_locs*)

generates a ajacency matrix based on connectivity array

apmapflow.scripts.apm_process_image_stack.**generate_index_map**(*nonzero_locs*, *shape*)

Determines the i,j,k indicies of the flattened array

apmapflow.scripts.apm_process_image_stack.**generate_node_connectivity_array**(*index_map*, *data_array*)

Generates a node connectivity array based on faces, edges and corner adjacency

apmapflow.scripts.apm_process_image_stack.**main**()

Driver program to load an image and generate maps. Memory requirements when processing a large TIFF stack can be very high.

apmapflow.scripts.apm_process_image_stack.**patch_holes**(*data_map*)

Fills in any areas with a non finite value by taking a linear average of the nearest non-zero values along each axis

apmapflow.scripts.apm_process_image_stack.**process_image**(*img_data*, *num_clusters*, ***kwargs*)

Processes a tiff stack on retaining voxels based on node connectivity. The clusters are sorted by size and the large N are retained.

apmapflow.scripts.apm_process_image_stack.**remove_isolated_clusters**(*conns*, *nonzero_locs*, *num_to_keep*, ***kwargs*)

Identifies and removes all disconnected clusters except the number of groups specified by "num_to_keep". num_to_keep=N retains the N largest clusters

apmapflow.scripts.apm_process_image_stack.**save_cluster_image**(*cs_ids*, *groups*, *counts*, *locs*, *img_shape*, *img_name*)

Saves an 8 bit image colored by cluster number

apm_process_paraview_data

Description: Generates 2-D data maps from OpenFoam data saved by paraview as a CSV file. The data has to be saved as point data and the following fields are expected p, points:0->2, u:0->2. An aperture map is the second main input and is used to generate the interpolation coordinates as well as convert the flow velocities into volumetic flow rates. This script assumes the OpenFoam simulation was performed on a geometry symmetric about the X-Z plane.

For usage information run: `apm_process_paraview_data -h`

Written By: Matthew stadelman
Date Written: 2016/09/29
Last Modfied: 2017/04/23

apmapflow.scripts.apm_process_paraview_data.**generate_coordinate_arrays**(*aper_map*, *para_data_dict*)

Generates the coordinate arrays to use in data interpolation for coverting paraview point data into a 2-D data map.

apmapflow.scripts.apm_process_paraview_data.**main**()

Processes commandline args and runs script

apmapflow.scripts.apm_process_paraview_data.**read_data_files**(*para_file*, *map_file*)

Reads in the paraview data file and aperture map file.

apmapflow.scripts.apm_process_paraview_data.**save_data_maps**(*map_coords*, *data_coords*, *aper_map*, *data_dict*, *density*)

Converts the raw paraview point data into a 2-D data distribution and saves the file by appending to the base_name.

apm_resize_image_stack

Description: Reduces the y-axis size of an image stack by cropping out the region above and below the fracture geometry. This can offer a significant filesize reduction if the y-axis significantly extends outside of the geometry.

For usage information run: `apm_resize_image_stack -h`

Written By: Matthew stadelman
Date Written: 2016/09/13
Last Modfied: 2017/04/23

apmapflow.scripts.apm_resize_image_stack.**main**()

Driver function to color the tif image.

apmapflow.scripts.apm_resize_image_stack.**resize_image**(*image_file*, *invert*)

Handles resizing the image y-axis

apm_run_lcl_model

Description: Runs each of the provided input files through the modified local cubic law model. Add the line ;EXE-FILE: (exec file) to the input file to use a different version of the local cubic law model.

For usage information run: `apm_run_lcl_model -h`

Written By: Matthew stadelman
Date Written: 2017/04/04
Last Modfied: 2017/04/23

apmapflow.scripts.apm_run_lcl_model.**main**()
Driver function to handle parsing command line args and running the model.

apm_subtract_data_maps

Description: Reads in an aperture map and then two data maps to subtract them. Any region of zero aperture is set to zero for the comparisons. The data maps can be normalized both before and after if desired and the final calculated data map is output.

For usage information run: `apm_subtract_data_maps -h`

Written By: Matthew stadelman
Date Written: 2016/10/27
Last Modfied: 2017/04/23

apmapflow.scripts.apm_subtract_data_maps.**main**()
Parses command line arguments and delegates tasks to helper functions for actual data processing

apmapflow.scripts.apm_subtract_data_maps.**output_percentile_set**(*data_field*, *args*)
Does three sets of percentiles and stacks them as columns: raw data, absolute value data, normalized+absolute value

apmapflow.scripts.apm_subtract_data_maps.**prepare_maps**(*args*)
loads the aperture map and data maps and then masks zero aperture zones as well as performs pre-subtraction normalization if desired.

apmapflow.scripts.apm_subtract_data_maps.**process_maps**(*aper_map*, *data_map1*, *data_map2*, *args*)
subtracts the data maps and then calculates percentiles of the result before outputting a final map to file.

Notes/ Tips/ Pitfalls:

- If the model is compiled using 32-bit compiler, running too large of a map can cause a memory overflow error.
- This guide assumes you install Anaconda3 locally. If you choose to install it system wide you will need to run some commands with `sudo` in unix systems or in an elevated command prompt in Windows.
- Running `./bin/build_model debug` will recompile the model using additional flags, code coverage and profiling
- Using Anaconda inside a Babun prompt is tricky and takes some effort to get fully functional.

- Your `$PATH` variable will need to be manually adjusted so the conda version of Python will shadow the default version used in Babun.
- Direct use of the conda Python interpreter doesn't work and it instead needs to be called with `python -i`.

Python Module Index

a

Index

Symbols

R

S

U

V

W

Z