PyFOAM wrapper for OpenFOAM and EOF-library

The program has been written by Ivan Smolianov

i.a.smolyanov@gmail.com

Программа предназначена для автоматизации ряд рутинных процедур настройки задач для решения в компьютерных средах OpenFOAM, Elmer и их совместного взаимодействия с использованием библиотеки EOF-library. Главным образом данная программа разрабатывалась для расчета гидродинамических, магнитогидродинамических и задач масса теплопереноса, но ее возможности этим не ограничиваются. С помощью кода PyFOAM можно проводить параметрические расчёты, расчеты для определения начальных условий с последующим переносом результатов в интересующий кейс, изменять любые настройки кейсов в одном текстовом файле и множество других процедур, которые отнимают существенное количество времени.

Основные цели и задачи кода PyFOAM:

* Сократить время на настройку задач OpenFOAM.
* Снизить количество рутинных операций и больше сфокусироваться на обработки и анализе результатов.
* Сделать возможным легко проводить параметрические исследования
* Возможность использования алгоритмов оптимизации

Структура программы:

* AdditionalFieles – хранит текстовые файлы с настройками различных моделей турбулентности.
* Examples – папка содержит ряд примеров по использованию данной программы
* Modules – основная папка программы содержащая различные модули для работы с кейсами OpenFAOM, Elmer и EOF.
* solvers – хранятся основные решатели EOF и OpenFOAM для работы с примерами из папки Examples.

Описания Modules.

На данном этапе разработки программа содержит 8 модулей представляющие собой 8 Python классов:

1. Модуль **Elmer** предназначен для автоматического редактирования текстового файла настроек Elmer моделей.
2. Модуль **InitialValue** служит для настройки начальных условий моделей OpenFOAM.
3. Модуль **Manipulations** предназначен для осуществления процедур копирования, дублирования, удаления информации и настроек моделей.
4. Модуль **Meshes** служит для работы с сетками OpenFAOM и Elmer.
5. Модуль **ParametricSweep** позволяет запускать расчеты в параметрическом исследовании.
6. **RunApplication** служит для запуска кейсов на расчет с заданными настройками.
7. **SetConstant** – для работы с папкой constant кейсов OpenFOAM.
8. **SetSystem** предназначен для настройки папки system кейсов OpenFOAM.

**Manipulations module.**

Основная идея работы с кодом PyFOAM – это создать основной кейс задачи (base case), настройки которого необходимо изменить, создать его копию, с помощью модулей PyFOAM настроить его по требованию пользователя и запустить на расчет в желаемом режиме исследования.

Основной кейс служит шаблоном для настраиваемой модели. В файлах настроек этого кейса расставляются ключевые слова (keys, ключи, тэги) в тех местах, которые должны будут изменяться. Приведем пример, необходимо в задаче изменять скорость потока и время расчета, тогда в папке ‘0’ файла ‘U’ необходимо указать уникальное ключевое слово, например, ‘Uin\_var’, тоже самое необходимо сделать для переменной времени, хранящейся в папке ‘system’ файла ‘contolDict’, используя другой уникальный ключ. Модули PyFOAM позволяют в последствии заменять эти ключи на заданные значения пользователем в самом основном кейсе или его копии. После настройки шаблонного кейса (baseCase) для удобства работы лучше сделать его копию и использовать ее для запуска и настроек модели, что позволит неоднократно изменять заданные вами ключи и менять настройки. Преобразовать выставленные ключевые слова в реальные значения с помощью модулей PyFOAM и запустить на рсчет.

Проведение манипуляций задания путей для основного и нового кейсов, копирование, удаление можно провести с помощью модуля Manipulations. Прежде, чем перейти к его описанию, отметим, что работа с базовыми кейсами позволяет безопасно работать без угрозы, что вы «сломаете» кейс и всегда вернуться к изначальным настройкам, но этим не исчерпывается. Можно настраивать и запускать уже настроенные кейсы без их дублирования.

Класс (модуль) **Manipulations** хранит три атрибута, другими словами, три переменные

* это имя узла (self.name);
* словарь с перечнем путей (self.pathes);
* словарь с именами кейсов (self.namesCases).

Атрибут self.name предназначен для задания переменной имени конкретному классу для собственного понимания с каким классом вы работаете. Иногда удобно при работе с большим количеством классов, чтобы их не перепутать между собой. На работу программ этот атрибут не влияет, а создан для удобства работы с модулями.

Атрибут содержащий словарь с перечнем путей (self.pathes) предназначен для хранения требуемых в работе над кейсами путей. По умолчанию при инициализации класса создается словарь с тремя ключами. Если переменные newPath, basePath и runPath не заданы при инициализации класса **Manipulations**, они определятся как тип None. Количество путей в словаре не ограничено. Доступ к путям производится по ключу в словаре.

Последним атрибутом является словарь имен self.namesCases. Данный словарь удобно использовать для хранения имен существующих или будущих кейсов, которые можно получить по ключу в данном словаре.

Класс Manipulations можно создать простой командой

*Manipulations()*

тогда будет создан класс с настройками по умолчанию –

*Manipulations*(name='firts', runPath=None, newPath=None, basePath=None) или можно задать эти атрибуты самостоятельно.

Перейдем к описанию методов класса:

|  |  |
| --- | --- |
| Метод | Описание |
| getName(self, keyName) | Получить имя из словаря self.namesCases по ключу keyName. Если keyName в словаре нет, выдаст ошибку. Этот метод возвращает строку имени. |
| getPath(self, keyPath) | Получить путь из словаря self.pathes по ключу keyPath. Если keyPath в словаре нет, выдаст ошибку. Этот метод возвращает строку пути. |
| createYourPath(self, path, keyPath='testPath') | Добавляет путь path в словарь self.pathes с ключом keypath. |
| changePath(self, path, keyPath='newPath') | Изменяет путь в словаре self.pathes с ключом keyPath на значение path. |
| createNewPath(self, dirmame=None, newCaseName=None, keyPath='newPath') | Создаем путь на основе директории dirmame, в которой лежит требуемый кейс с именем newCaseName, в словаре self.pathes с ключом keypath. Данный метод возвращает строку пути. |
| generatorNewName(self, \*namesNewCase, baseNewName='', keyName='newName', splitter='\_') | Метод создает имя кейса в словаре self.namesCases с ключем keyName. из основы для имени baseNewName, присоединяя к нему по очереди переменные \*namesNewCase, разделяя их разделителем splitter. Удобно использовать для добавления переменных характеризующих данный кейс в его названием. Например, baseNewName ='MHDpump', а namesNewCases это значения это магнитная индукция Bv\_var. Тогда получим название. ‘MHDpump\_Bm\_1.5’ |
| dublicateCase(self, basePath=None, newPath=None, keyPathes=('basePath', 'newPath'),  keyNames=('baseName', 'newName'), mode='copy') | Создает копию кейса, находящегося в basePath в директории newPath. После выполнения процедуры копирования, создает или переписывает в словарях self.casesNames и self.pathes имена и пути основного и нового скопированного кейсов. Переменная mode характеризует, как будет производится копирование если в директории назначения существует папка с таким именем. Реализовано два режима (mode) – это copy и rewrite. Режим copy создает дубликат существующего кейса с добавлением даты дублирования в имя папки, а новый кейс копируется с текущим именем. Режим rewrite удаляет существующий кейс с таким же именем и на это место копирует новый. |
| checkExistence(self, basePath, newPath) | Вспомогательная функция класса, проверяющая корректность введенного пути. |
| priorityPath(self, basePath, newPath) | Вспомогательный метод класса служит для проверки приоритета пути для ряда методов, например, для dublicateCase. Если в методе не определен путь, тогда этот будет взят из словарей с ключами 'basePath’ и newPath. |

**Elmer module.**

Класс Elmer разработан для модификации настроек задачи Elmer, которые хранятся в текстовом файле с разрешением .sif. Основная идея модификации этого текстового файла заключается в задании ключевых переменных в sif файле, которые потом будут замены на соответствующее значение.

Инициализировать класс можно с помощью команды:

Elmer (self, pathCase=None, sifName='.sif')

Где, можно определить путь кейса pathCase в котором необходимо провести редактирование настроек sif файла с именем sifName. Если эти атрибуты не задать, а инициализировать класс как Elmer(), по умолчанию атрибуты будут None и ‘.sif’.

Метод setElmerVar(self, \*elmerDictioaries, pathCase=None, sifName=None) класса Elmer предназначен для поиска переменных с именами соответствующие ключам словарям \*elmerDictioaries в файле sifName, находящегося в директории pathCase и замене их на значения словарей \*elmerDictioaries по ключу. Если в этом методе не указан pathCase или sifName, метод возьмет имя из существующих атрибутов класса, если их там тоже нет, тогда выдаст ошибку.

Модуль **setConstant** содержит один класс под названием SetConstantParam. Основное назначение данного класса проводить манипуляции над файлами в папке constant кейсов OpenFOAM. Атрибуты этого класса это путь где лежит кейс pathCase и путь до библиотеки PyFOAM pathLib. В будущем атрибут pathLib будет удален.

Методы:

setTransportProp(self, \*lists) заменяет значения, имеющие название такое же как ключе словарях \*lists на значения этого словаря для данного ключа, в файле transportProperties.

setTurbModel2(self, typeTurbModel='kEpsilon', pathCase=None) копирует из библиотеки нужный файл с настройками модели турбулентности в кейс, лежащий в директории pathCase. Если путь не задан в методе, он берется из атрибута класса. На данный момент реализованы следующие модели турбулентности LES, kEpsilon, realizableKE, kOmega, kOmegaSST, laminar, LESSmag.

setAnyConstantFiles(self, \*listsVar, files=['controlDict'], pathCase=None) изменяет значение совпадающие с ключами в словарях \*listsVar на значения этого словаря в файлах files, расположенный в папке constant кейса, лежащего в директории pathCase. Если путь не задан, тогда он берется из атрибута класса.

Модуль **SetSystem** содержит класс SetSystem() предназначенный для настройки файлов в папке system кейсов OpenFOAM. При инициализации класса можно задать путь кейса, в котором планируется проводить данные манипуляции.

Метод setControlDict(self, \*listsControlDicts, pathCase=None) заменяет совпадающие переменные с ключами словарей \*listsControlDicts на значение словаря для данного ключа в файле controlDict указанного кейса в директории pathCase.

Метод setfvSolution(self, \*listsfvSolution, pathCase=None) заменяет совпадающие переменные с ключами словарей \* listsfvSolution на значение словаря для данного ключа в файле fvSolution указанного кейса в директории pathCase.

Метод setfvSchemes(self, \*listsfvSchemes, pathCase=None) заменяет совпадающие переменные с ключами словарей \* listsfvSchemes на значение словаря для данного ключа в файле fvSchemes указанного кейса в директории pathCase.

Метод setAnyFiles(self, \*listsVar, files=['controlDict'], pathCase=None) заменяет совпадающие переменные с ключами словарей \* listsVar на значение словаря для данного ключа в файлах, указанных в списке files, заданного кейса в директории pathCase.

Модуль **RunApplication** cодержит класс Runner() предназначенный для автоматизации процедур запуска кейсов OpenFOAM и EOF. При инициализации класса Runner(name='test', pathCase=None ) можно присвоить ему имя для этого класса name и путь до кейса, который необходимо запустить.

Метод setPathCase(self, pathCase) позволяет задать путь для кейса или изменить существующий для этого класса.

Метод setCores(self, coreOF=4, CoreElmer=4) устанавливает значение требуемых ядер в параллельном вычислении для OpenFOAM с помощью переменной coreOF и Elmer с помощью переменной CoreElmer.

Метод setCoresOF(self, coreOF=4) устанавливает количество ядер для параллельного расчета кейсов OpenFOAM.

Метод seCoresElmer(self, coreElmer=4, meshName='') устанавливает количество ядер для параллельного расчета задач Elmer и имени сетки meshName, что бы ее разбить при необходимости.

Метод setCoresEOF(self, coreOF=4, coreElmer=4, elmerMeshName='') устанавливает значение требуемых ядер в параллельном вычислении для OpenFOAM с помощью переменной coreOF и Elmer с помощью переменной CoreElmer, а также имя для сетки Elmer, что бы можно было ее разбить.

Метод setDecomposeParDict(self, coreOF=None, nameVar='core\_OF', pathCase=None) устанавливает значение ядер для распараллеливания в файле decomposeParDict, заменяя значение nameVar в этом файле.

Метод setNameSolver(self, solverName='pimpleFoam') определяет имя решателя для запуска кейса.

Метод setModeRunner(self, mode='common') определяет режим запуска. На данном этапе реализовано три типа запуска. Режим ‘common’ запуск только OpenFOAM решателей без распараллеливания. Режим ‘parallel’

Метод setPyFoamSettings(self, pyFoam=False) задает настройку запускать во время расчета отображение сходимости. Работает пока только для режимов ‘parallel’ и ‘common’.

Метод setFields(self, pathCase=None) вызывает команду setFields в папке pathCase.

Метод decomposeElmer(self, decomposeElmer) производит распараллеливание сетки Elmer если значение decomposeElmer является True.

Метод decompose(self, decomposeOF) производит распараллеливание сетки OpenFOAM если значение decomposeOF является True.

Метод runCase(self, pathCase=None, decomposeOF=True, decomposeElmer=False) производит запуск решения с заранее заданным решателем и режимом решения кейса, находящегося в директории pathCase. Если путь не задан берется из атрибутов класса. Переменные decomposeOF и decomposeElmer являются логическими и описывают производить процедуру распараллеливания сетки перед запуском кейса.

Класс Mesh() предназначен для работы с настройками сеток OpenFOAM и Elmer. При инициализации класса можно задать путь кейс Mesh(pathCase=’None’).

Метод setBlockMesh(self, meshList, pathCase=None) заменяет значения переменных, которые совпадают с ключами в словаре meshList на значения этого словаря для найденного ключа в файле blockMeshDict кейса расположенного в директории pathCase.

Метод runBlockMesh(self, pathCase=None) выполняет команду запуска утилиты blockMesh в кейсе с путем pathCase.

Метод settingsElmerMesh(self, meshList, pathCase=None, elmerMeshName='') позволяет редактировать файл настроек сетки ельмера, например с разрешением geo.

Метод runElmerMesh(self, pathCase=None) запускает преобразование формата сетки geo в формат Elmer.

Метод **ParametricSweep()** позволяет запускать решение с разными значениями переменных или настроек кейса последовательно. Для этого необходимо определить перебираемый словарь. Ключи у этого словаря должны соответствовать переменным, которые должны быть варьироваться. Значение этого словаря должны быть листы с набором значений. Листы для каждого ключа должны иметь одинаковый размер. Затем необходимо задать в каких словарях искать варьируемые переменные. Следующим шагом является задать функцию, содержащую перечень процедур, которые должны быть запущены в параметрическом исследовании. Эта функция будет вызываться столько раз, сколько переменных в перебираемом листе.

Метод setFun(self, fun1=None) задает python функцию, которая будет выполнятся в параметрическом исследовании.

Метод getSweepDict(self, dict={'keys': [1, 2, 3]}) определяет варьируемый словарь.

Метод getDicts(self, dicts=[]) определяет в каких словарях искать варьируемые переменные.

Метод run(self, generatorName=False) запускает параметрическое исследование. Если generatorName=True, тогда будет генерироваться дополнительная переменная строка name, содержащая название варьируемых переменных и их значение на текущем исследовании. При активации этой опции необходимо в запускаемую функцию добавить, как минимум одну входную переменную. Эту переменную удобно иногда использовать в генераторе названия кейса, что бы показать какие переменные соответствуют этому кейсу.

|  |
| --- |
| Приложение 1. Модуль Manipulations |
| class Manipulations():  def \_\_init\_\_(self, name='firts', runPath=None, newPath=None, basePath=None):  self.name = name  self.pathes = {'newPath': newPath,  'basePath': basePath,  'runPath' : runPath}  self.namesCases = {'newName': None}  def \_\_repr\_\_(self):  return f"Name of manipulation node ({self.name}, runpath {self.pathes['runPath']}, basepath " \  f"{self.pathes['basePath']}, newPath {self.pathes['newPath']})"  def \_\_str\_\_(self):  return f"Name of manipulation node ({self.name}, runpath {self.pathes['runPath']}, basepath " \  f"{self.pathes['basePath']}, newPath {self.pathes['newPath']})"  def dublicateCase(self, basePath=None, newPath=None, keyPathes=('basePath', 'newPath'),  keyNames=('baseName', 'newName'), mode='copy'):  """The function creates copy of the base case.  pathBaseCase is the path of base case that will be copied by the function  pathNewCase is the path of new case that will be created by the function  mode defines how the procedure of copying will be done.  a) rewrite mode is the mode when folder of new case already being existed, then the folder  will delited by the function and base case folder will be copied to the folder being the same name  b) copy mode is the mode when folder of new case already being existed, then the folder will be copied  to the folder being old name with prefix of current time of copying. And new case will be copied  to folder being path of pathNewCase variables."""  basePath, newPath = self.priorityPath(basePath, newPath)  self.checkExistence(basePath, newPath)  if mode == 'rewrite':  print(f'The folder {os.path.basename(newPath)} is exist. The script run the rewrite mode')  shutil.rmtree(newPath)  shutil.copytree(basePath, newPath)  elif mode == 'copy':  print(f'The folder {os.path.basename(newPath)} is exist. The script run the copy mode')  now = datetime.datetime.now()  old\_file = newPath + '\_' + 'old' + '\_' + now.strftime("%d-%m-%Y %H:%M")  try:  shutil.move(newPath, old\_file)  self.oldNameCase = os.path.basename(old\_file)  except shutil.Error:  print('You run the script is often. There is exception old case')  shutil.copytree(basePath, newPath)  else:  shutil.copytree(basePath, newPath)  self.pathes[keyPathes[0]] = basePath  self.pathes[keyPathes[1]] = newPath  self.namesCases[keyNames[0]] = os.path.basename(basePath)  self.namesCases[keyPathes[1]] = os.path.basename(newPath)  def generatorNewName(self, \*namesNewCase, baseNewName='', keyName='newName', splitter='\_'):  """The function serves to create two variables of base and new case paths.  The name of new case is generated by special algorithm realized in the fucntion.  The name will be created by adding variables of the list namesNewCase to base case folder name. The variables  will be seprated by special sympol (spliter) to name of the folder.  Variables:  \*namesNewCase is a number of variables, which will be added to name of new case  baseCase is the folder name of base case  splitter is the variables defending the for separation in folder name of new case """  for addName in namesNewCase:  baseNewName += splitter + str(addName)  self.namesCases[keyName] = baseNewName  return self.namesCases[keyName]  def createNewPath(self, dirmame=None, newCaseName=None, keyPath='newPath'):  """The function is used for creating new path  Variables  dirname is the path of directory where new folder of case put  newCaseName is the name of new case  newPath is the path of new case  """  self.pathes[keyPath] = os.path.join(dirmame, newCaseName)  return self.pathes[keyPath]  def changePath(self, path, keyPath='newPath'):  """The function is used for changing existent path by name  Input variables  path is new given path  key is the name of variables of key for dictionary of addtionals pathes"""  if keyPath in self.pathes.keys():  self.pathes[keyPath] = path  else:  print('Error the key of path is not exist!')  def createYourPath(self, path, keyPath='testPath'):  """The function is used to create your own path  The created path will be written into dictionary self.addtionaldictionary with key = name  Input path is path of your new given path  name is the key of dictionary storaged all addtional pathes"""  self.pathes[keyPath] = path  def getPath(self, keyPath):  """The methods gives back path acording givven name or key  Input:  key is the name of class variables consisting pathes or key of dictionary with pathes """  if keyPath in self.pathes.keys():  return self.pathes[keyPath]  else:  print('Error: The given name of key with pathes is not exist!')  def getName(self, keyName):  """The methods gives back path acording givven name or key  Input:  key is the name of class variables consisting pathes or key of dictionary with pathes """  if keyName in self.namesCases.keys():  return self.namesCases[keyName]  else:  print('Error: The given name of key with pathes is not exist!')  def checkExistence(self, basePath, newPath):  """The method supports to find out existing gotten pathes  If one of the gotten pathes is not exist, program is interupted  """  if not os.path.exists(basePath):  sys.exit('Error: The base case is not exist in the directory!!!')  elif not os.path.exists(newPath):  dirname, newCaseName = os.path.split(newPath)  if os.path.exists(dirname):  os.mkdir(f'{newPath}')  else:  sys.exit(f'Error: The new path {dirname} is not exist !!!')  def priorityPath(self, basePath, newPath):  """The method is used for selection of given path  the first priority is given path by methods  the second priority is given path by class constructor  If both path is None, the program is interupted  Input :  basePath, newPath is checkoing pathes  Output:  retrunBasePath, returnNewPath is selected pathes acording priority  """  if basePath == None:  if self.pathes['basePath'] != None:  retrunBasePath = self.pathes['basePath']  else:  sys.exit('Error: You do not enter the base path!!!')  else:  retrunBasePath = basePath  if newPath == None:  if self.pathes['newPath'] != None:  returnNewPath = self.pathes['newPath']  else:  sys.exit('Error: You do not enter the new path!!!')  else:  returnNewPath = newPath  return retrunBasePath, returnNewPath |