

Описание генератора функций

Введение. Текущие возможности и актуальные файлы

Сейчас (07.08.15) генератор умеет генерировать функции в таких случаях:

- одномерный случай с соединениями, ни в одном уравнении нет производных порядка выше второго;
- двумерный случай с соединениями, ни в одном уравнении не содержится смешанных производных и производных порядка выше второго;
- двумерный случай, в уравнениях могут содержаться смешанные производные второго порядка, ни в одном уравнении нет производных порядка выше второго — правильно может быть обработана только ситуация, когда нет соединяющихся блоков (т.е. при генерировании функций не приходится приближать смешанные производные в местах соединений).
- трехмерный случай с соединениями, ни в одном уравнении не содержится смешанных производных и производных порядка выше второго.

Для генерирования функций используются файлы, находящиеся в папке domain-model:

1. customOfficer.py;
2. equationParser.py;
3. newFuncGenerator.py;
4. rhsCodeGenerator.py;
5. derivCodeGenerator.py;
6. someFuncs.py;
7. DerivHandler.py.

Файлы 6 и 7 используются для генерации, но функции, лежащие в них, могут использоваться и для других целей. Из файла 6 для генерации используются функции **NewtonBinomCoefficient()**, **generateCodeForMathFunction()**, **determineNameOfBoundary()**, **RectSquare()**, **determineCellIndexOfStartOfConnection2D()**, **getRanges()**. Функции **factorial()** и **getCellCountAlongLine()** из этого же файла используются функциями **NewtonBinomCoefficient()**, **determineCellIndexOfStartOfConnection2D()**, **getRanges()**. Файл 7 содержит функцию, которая находит порядок старшей производной.

1 Описание основных файлов

1.1 Файл customOfficer.py и проверка входных данных

Файл 2 (customOfficer.py) создан для проверки некоторых введенных в *.json данных на корректность перед запуском генерации функций. Этот файл содержит класс **Reviewer**, который связан с внешним миром методом **ReviewInput()**. В методе **createCPPandGetFunctionMaps()** класса **Model** создается экземпляр класса **Reviewer** и вызывается его метод **ReviewInput()**. Этот метод проверит правильность введения блоков и параметров.

Основные методы класса **Reviewer**:

- **ReviewParameters()**. Проверит, нет ли в списке параметров "Params" повторяющихся имен; совпадает ли множество ключей каждого из словарей в "ParamValues" с множеством элементов списка "Params"; правильно ли установлен "DefaultParamsIndex".
- **ReviewBlocks()**. Проверит каждый из блоков по таким критериям:
 - размеры блока ("Size") заданы неотрицательными числами;
 - индекс уравнения по умолчанию и индекс начального условия по умолчанию заданы корректно;
 - корректно заданы границы регионов уравнений, регионов начальных условий, регионов граничных условий (при этом нет контроля за наложением одного региона на другой, это остается на совести пользователя); этим будет заниматься метод **ReviewEqRegOrInitRegOrBoundReg()**;
 - одинаково количество уравнений в каждой системе, участвующей в задаче (система участвует в задаче, если она задана в списке "Equations" и при этом есть блок или часть блока, на котором задана именно эта система) (метод **ReviewEquations()**);
 - количество компонент для каждого начального условия совпадает с количеством уравнений в каждой системе (метод **ReviewInitials()**). То же самое и для граничных условий (метод **ReviewBounds()**).
- **ReviewInput()**. Вызывает методы **ReviewParameters()** и **ReviewBlocks()**.

Недостатки проверяльщика:

- Не определено до конца, какие данные нужно проверять на корректность, а какие нет. Поэтому некоторые проверки могут быть лишними, некоторые — отсутствовать.
- Корректность ввода интерконнектов пока что нигде не проверяется!

1.2 Файл equationParser.py и парсинг уравнений, начальных и граничных условий

Мозги парсера — библиотека `ruparsing`. Этот файл (equationParser.py) содержит три класса:

- **CorrectnessController**. Занимается проверкой уравнений и математических функций (например, используемых в качестве начальных или граничных условий) на корректность ввода (например, правильность расстановки скобок, правильность расстановки операторов $+$, $-$, $*$, $/$ и т.д.)

- **ParsePatternCreator**. Чтобы распарсить некое выражение средствами `ruparsing`, нужно составить подходящую под это выражение грамматику. Нам надо парсить уравнения (выделять их правые части) и отдельно функции. Для этого надо создать 2 разных (но очень похожих) грамматики. Еще надо получать из левых частей уравнений список компонент искомой функции. Надо поэтому уравнения парсить уже по-другому (выделять их левые части), т.е. есть необходимость создания третьей грамматики. Этот класс отвечает за создание этих грамматик.
- **MathExpressionParser**. С помощью созданной предыдущим классом грамматики либо парсит уравнение, либо математическую функцию, либо возвращает список компонент искомой функции.

Клиент использует только класс 1.2. Это происходит в методе `generateAllPointInitials()` класса `abstractGenerator` для того, чтобы парсить начальные условия; в методе `generateCentralFunctionCode()` того же класса для парсинга уравнений, необходимых для генерации центральных функций; в реализациях метода `generateBoundsAndIcs()` классов `generator1D` и `generator2D` (а в будущем и `generator3D`).

Недостатки парсера:

- Работает долго.
- Проверка на корректность ввода уравнений и математических функций (расстановка скобок и т.д.) сделана без использования средств `ruparsing`. Это порождает целый дополнительный класс, который занимается проверкой.

1.3 Файл `newFuncGenerator.py`, выполняющий основную работу

Класс `abstractGenerator` и его наследники `generator1D` и `generator2D` выполняют почти всю работу, связанную с генерацией (оставшуюся, но все же весьма значительную часть работы выполняют классы, находящиеся в файлах 4, 5).

В методе `createCPPandGetFunctionMaps()` класса `Model` создается экземпляр класса `FuncGenerator`, описание которого находится в рассматриваемом файле 3. В конструкторе этого класса в зависимости от размерности задачи выбирается конкретная реализация генератора (т.е. полем этого класса становится экземпляр одного из трех классов `generator1D`, `generator2D`, `generator3D`). В дальнейшем метод `generateAllFunctions()` класса `FuncGenerator` работает именно с этой конкретной реализацией. Также этот метод формирует список словарей `functionMaps`, по которому формируется матрица пересчета и структура которого описана в файле `definition` в папке `doc`.

Вот полный список классов файла `newFuncGenerator.py` с их описанием:

- **FuncGenerator**. Опираясь на генератор для нужной размерности, определенная в нем функция `generateAllFunctions()` генерирует:
 - определения всех констант (дефайнов);
 - функции для работы с начальными условиями и параметрами;
 - для каждого блока — набор центральных функций, набор граничных, угловых и соединительных (интерконнектов) функций.
- **abstractGenerator**. Содержит общие для всех трех генераторов поля и методы. Например, в нем определены методы генерирования констант (дефайнов) (метод `generateAllDefinitions()`), функций для начальных условий (`generateAllPointInitials()`), параметров (`generateParamFunction()`); методы генерирования

функций для граничных условий Дирихле (**generateDirichlet()**) или Неймана, а также интерконнектов (**generateNeumannOrInterconnect()**) (функция, генерирующая интерконнект, совпадает с функцией, генерирующей Неймана, просто ей передаются параметры с другим смыслом); метод генерирования центральных функций **generateCentralFunctionCode()**.

- **generator1D, generator2D, generator3D**. Содержат специфические методы и поля, нужные для обработки задачи указанной размерности. Генератор **generator3D** не работает и не изменялся очень очень давно. Важно то, что у этих классов есть общие методы **getBlockInfo()** и **generateBoundsAndIcs()**, которые вызываются в методе **generateAllFunctions()** класса **FuncGenerator**. В первом методе определяются все уравнения, заданные на блоке (т.е. копится информация для генерации центральных функций), и для каждой границы определяются граничные условия и интервалы в клетках, на которых они действуют. А также начинает формироваться словарь из списка **functionMaps**. Во втором методе генерируются граничные, угловые, соединительные функции и завершается формирование словаря из списка **functionMaps**.
- **InterconnectRegion**. Создан для того, чтобы унифицировать обработку границ двумерного (и может, трехмерного) блока, т.к. граничные условия и соединения с другими блоками для массива **functionMaps** надо представлять в одном и том же формате. Просто при создании экземпляра генератора для каждого блока создается список его соединительных регионов.
- **BoundCondition** и **Connection**. Метод **getBlockInfo()** в двумерном (и может, в трехмерном) случае составляет список элементов, которые есть экземпляры этих классов.

1.4 Файл **rhsCodeGenerator.py** и генерирование правых частей уравнений, заданных пользователем

Файл содержит класс **RHSCodeGenerator**, который отвечает за генерацию правой части одного уравнения системы, заданной пользователем, с учетом краевых условий или соединений. Пример возврата такой функции — строка:

```
result[idx + 0] = params[0] * ((2.0 * DXM2 * (
    source[idx - Block0StrideX * Block0CELLSIZE + 0] -
    source[idx + 0] + (0.0) * DX)));
```

Этим занимается метод **generateRightHandSideCode()**. Методы **generateDirichlet()** и **generateNeumannOrInterconnect()** класса **abstractGenerator** используют именно этот метод. Отдельная задача — сгенерировать производную с нужным конечно-разностным приближением. Подготовкой к такой генерации занимаются методы **callDerivGenerator()** и **callSpecialDerivGenerator()**. А саму производную генерируют уже методы одного из двух классов, описанных в файле 5.

1.5 Файл **derivCodeGenerator.py** и генерирование производных

Т.к. в уравнении могут встретиться и чистые и смешанные производные, а смешанные производные в случае соединения блоков порождают много разных видов функций, которые необходимо сгенерировать, то удобно поручить генерирование чистых и смешанных производных разным классам. Эти классы:

- **PureDerivGenerator**;

- **MixDerivGenerator**.

1.5.1 PureDerivGenerator

Для центральной функции способен сгенерировать чистую производную любого порядка (потому что конечно-разностные приближения для них в этом случае содержат однотипные элементы, различно только их количество и коэффициенты перед слагаемыми). В зависимости от того, какими значениями был проинициализирован экземпляр этого класса, за генерацию производной отвечает одна из трех функций:

- **commonPureDerivativeAlternative()** — генерировать производную для центральной функции или для граничной в случае, когда граничное условие не влияет на производную (например, первая производная по y вдоль границы $x = 0$);
- **specialPureDerivativeAlternative()** — генерировать производную для граничной функции, когда граничное условие влияет на производную (например, производная по x вдоль границы $x = x_{max}$);
- **interconnectPureDerivAlternative()** — генерировать производную для функции-соединения (для интерконнекта).

1.5.2 MixDerivGenerator

Пока что способен генерировать смешанные производные только второго порядка для несоединенных блоков. Функции-генераторы:

- **commonMixedDerivativeAlternative()** — генерировать производную для центральной функции;
- **specialMixedDerivativeAlternative()** — генерировать производную для граничной функции.

Производные в углах двумерного блока генерируются по частям, а потом объединяются в одну строку в методе **callDerivGenerator()** класса **RHSCodeGenerator**.

2 Конечно-разностные формулы, используемые при генерировании производных

Расчетные функции (которые и генерирует генератор) отличаются друг от друга только выражениями, аппроксимирующими производные по пространственным переменным. Если рассматриваемая система уравнений n -мерна (т.е. искомая функция — это вектор-функция $u = (u_1, \dots, u_n)$, $u_k = u_k(t, x, y, z)$, $k = \overline{1, n}$) и ни одно из уравнений не содержит смешанных производных и производных порядка выше второго, то i -ое уравнение этой системы ($i = \overline{1, n}$) имеет вид:

$$\frac{\partial u_i}{\partial t} = f \left(u_1, \frac{\partial u_1}{\partial x}, \frac{\partial u_1}{\partial y}, \frac{\partial u_1}{\partial z}, \frac{\partial^2 u_1}{\partial x^2}, \frac{\partial^2 u_1}{\partial y^2}, \frac{\partial^2 u_1}{\partial z^2}, \dots, u_n, \frac{\partial u_n}{\partial x}, \frac{\partial u_n}{\partial y}, \frac{\partial u_n}{\partial z}, \frac{\partial^2 u_n}{\partial x^2}, \frac{\partial^2 u_n}{\partial y^2}, \frac{\partial^2 u_n}{\partial z^2} \right).$$

Для простоты записи можно считать, что система одномерна, т.е. есть всего одно уравнение:

$$\frac{\partial u}{\partial t} = f \left(u, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial u}{\partial z}, \frac{\partial^2 u}{\partial x^2}, \frac{\partial^2 u}{\partial y^2}, \frac{\partial^2 u}{\partial z^2} \right). \quad (1)$$

Надо аппроксимировать все производные, стоящие в правой части (1). Нужны аппроксимации для:

1. центральных функций;
2. функций для расчета границ блока;
3. функций для расчета вершин блока (в двумерном и трехмерном случаях);
4. функций для расчета ребер блока (в трехмерном случае).

В 2. - 4. надо учитывать краевые условия (в случае краевых условий Неймана как раз и возникает необходимость аппроксимировать производные, а в случае условий Дирихле вместо правой части уравнения (1) просто подставляется значение производной функции, являющейся этим условием), соединения блоков и то, что на разных частях одного блока могут быть заданы разные уравнения. Для приближения производных во всех функциях используются центральные конечные разности.

Пусть на рассматриваемом блоке введена сетка с шагами $\Delta x, \Delta y, \Delta z$ по пространственным переменным x, y, z и введен шаг Δt по времени t . Значения функции u в узлах сетки далее обозначаются обычным образом:

$$u_{ijk}^\ell := u(\ell \Delta t, i \Delta x, j \Delta y, k \Delta z),$$

а значения производных в узлах сетки пишутся похожим образом, например:

$$\left(\frac{\partial^2 u}{\partial y^2} \right)_{ijk}^\ell := \frac{\partial^2 u}{\partial y^2}(\ell \Delta t, i \Delta x, j \Delta y, k \Delta z).$$

Здесь $\ell = \overline{0, L}$, $i = \overline{0, M}$, $j = \overline{0, N}$, $k = \overline{0, P}$.

Замечание: т.к. генератор все-таки умеет генерировать функции для задачи, когда в уравнении есть смешанные производные и при этом рассматриваются блоки без соединений, то для этой ситуации ниже также описаны формулы, используемые генератором для аппроксимации смешанных производных. Для такой задачи уравнение (1) выглядит по-другому:

$$\frac{\partial u}{\partial t} = f \left(u, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial u}{\partial z}, \frac{\partial^2 u}{\partial x \partial y}, \frac{\partial^2 u}{\partial x \partial z}, \frac{\partial^2 u}{\partial y \partial z}, \frac{\partial^2 u}{\partial x^2}, \frac{\partial^2 u}{\partial y^2}, \frac{\partial^2 u}{\partial z^2} \right). \quad (2)$$

2.1 Формулы для производных в центральных функциях

В центральных функциях для приближения производных используется самый обычный вид конечно-разностных формул.

Первая производная (например, по z):

$$\left(\frac{\partial u}{\partial z}\right)_{ijk}^{\ell} \approx \frac{u_{ijk+1}^{\ell} - u_{ijk-1}^{\ell}}{2\Delta z}$$

Вторая чистая производная (например, по yy):

$$\left(\frac{\partial^2 u}{\partial y^2}\right)_{ijk}^{\ell} \approx \frac{u_{ij+1k}^{\ell} - 2u_{ijk}^{\ell} + u_{ij-1k}^{\ell}}{(\Delta y)^2}$$

Для центральных функций смешанные производные генерировать очень легко. Смешанная производная (например, по xz):

$$\left(\frac{\partial^2 u}{\partial x \partial z}\right)_{ijk}^{\ell} \approx \frac{u_{i+1jk+1}^{\ell} - u_{i-1jk+1}^{\ell} - u_{i+1jk-1}^{\ell} + u_{i-1jk-1}^{\ell}}{4\Delta x \Delta z}$$

2.2 Формулы для производных в функциях расчета границ блока

Рассмотрим какую-нибудь границу блока, например, $y = 0$. Пусть на этой границе задано условие Неймана:

$$\frac{\partial u}{\partial y}(t, x, 0, z) = \varphi(t, x, z). \quad (3)$$