# Описание генератора функций

## Введение. Текущие возможности и актуальные файлы

Сейчас (13.07.15) генератор умеет генерировать функции в таких случаях:

- одномерный случай, ни в одном уравнении нет производных порядка выше второго — генерируются **все** нужные функции;
- двумерный случай, ни в одном уравнении не содержится смешанных производных и производных порядка выше второго — тоже все функции;
- 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).

Для генерирования функций используются файлы, находящиеся в папке domainmodel:

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 Файл customOfficer.py и проверка входных данных

Файл 1 (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!

## 2 Файл equationParser.py и парсинг уравнений, начальных и граничных условий

Мозги парсера — библиотека pyparsing. 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.)
- **ParsePatternCreater**. Чтобы распарсить некое выражение средствами pyparsing, 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 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 матода **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 pyparsing. Это порождает целый дополнительный класс, который занимается проверкой.

## 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 которго описана в файле definition в папке doc.

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

- **FuncGenerator**. Оперируя генератором 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.

## 4 Файл rhsCodeGenerator.py и генерирование правых частей уравнений, заданных пользователем

Файл содержит класс **RHSCodeGenerator**, который отвечает за генерацию правой части одного уарвнения 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.

## 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**.

### 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).

### 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 обну строку в методе **callDerivGenerator()** класса **RHSCodeGenerator**.